EEE 569 DIGITAL IMAGE PROCESSING
HOMEWORK #3

ABINAYA MANIMARAN

MANIMARA@USC.EDU

SPRING 2018

SUBMITTED ON 03/28/2018

PROBLEM 1
TEXTURE ANALYSIS AND SEGMENTATION

A) TEXTURE CLASSIFICATION:

➢ ABSTRACT AND MOTIVATION:

Image texture can be defined as one important parameter of an image that gives information about spatial arrangement of pixel intensities i.e. RGB or grayscale levels. They are usually naturally found in an image during its capture or they can be artificially created on an image. Given a set of images, they can be classified/clustered into classes/clusters based on its texture. Since texture gives most information about the arrangement of pixel intensities, if those arrangement are captured by feature extraction methods, these features become one of the important information for image classification or clustering. In this part of the question, 12 images are given for classification. The motivation is to cluster these images by extracting features that captures texture information using Laws Filters.

➢ APPROACH AND PROCEDURES:

This problem is approached by applying Laws Filters on the Images. There are 5 Laws Filters available. They are listed below.

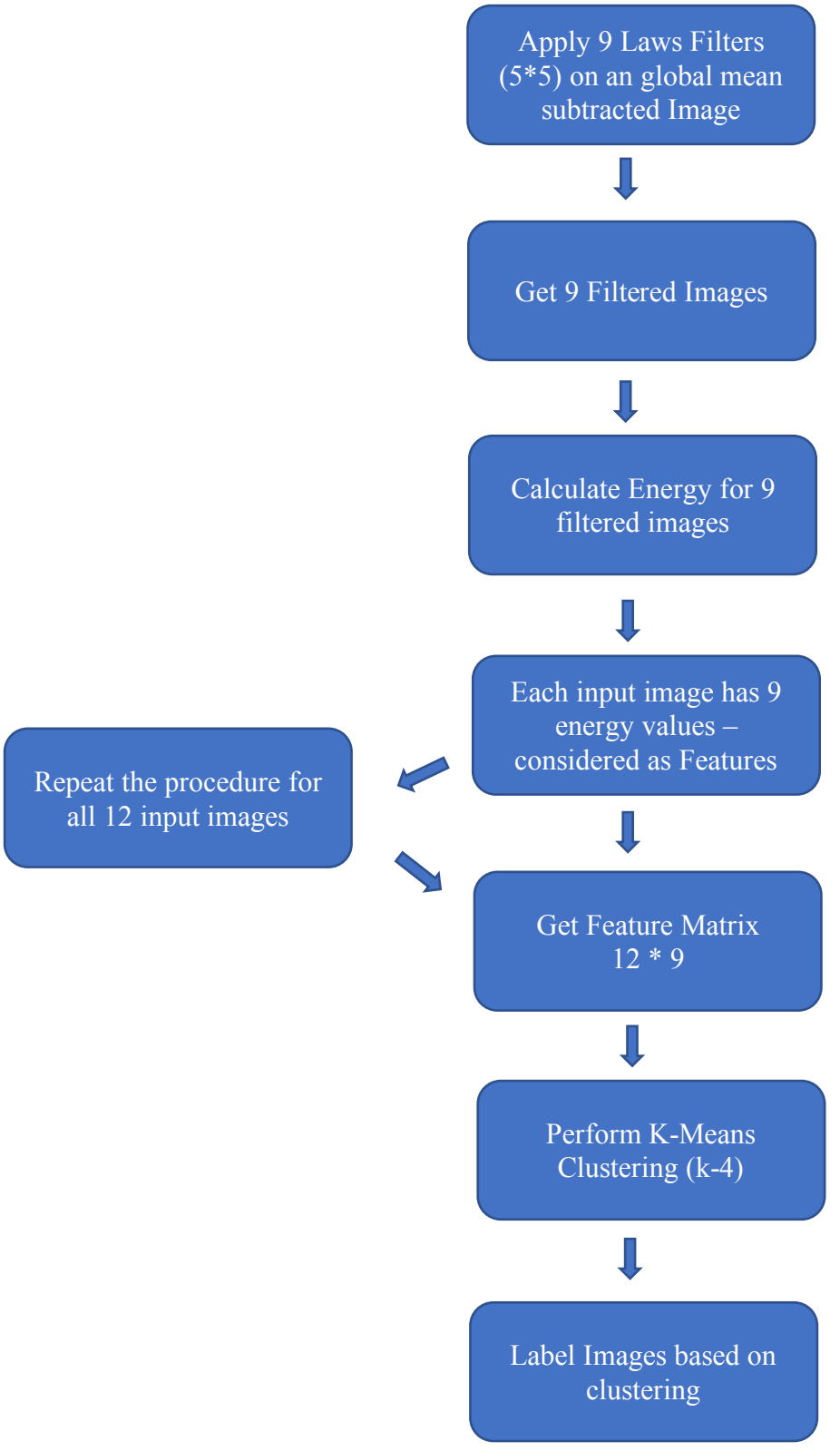
Filter	Kernel
Level (L5)	[1 4 6 4 1]
Edge (E5)	[-1 -2 0 2 1]
Spot (S5)	[-1 0 2 0 -1]
Wave (W5)	[-1 2 0 -2 1]
Ripple (R5)	[1 -4 6 -4 1]

A 5*5 filter for image texture filtering can be done by tensor product of any 2 filters above. Therefore, using the above five 1D filters, 25 masks can be obtained.

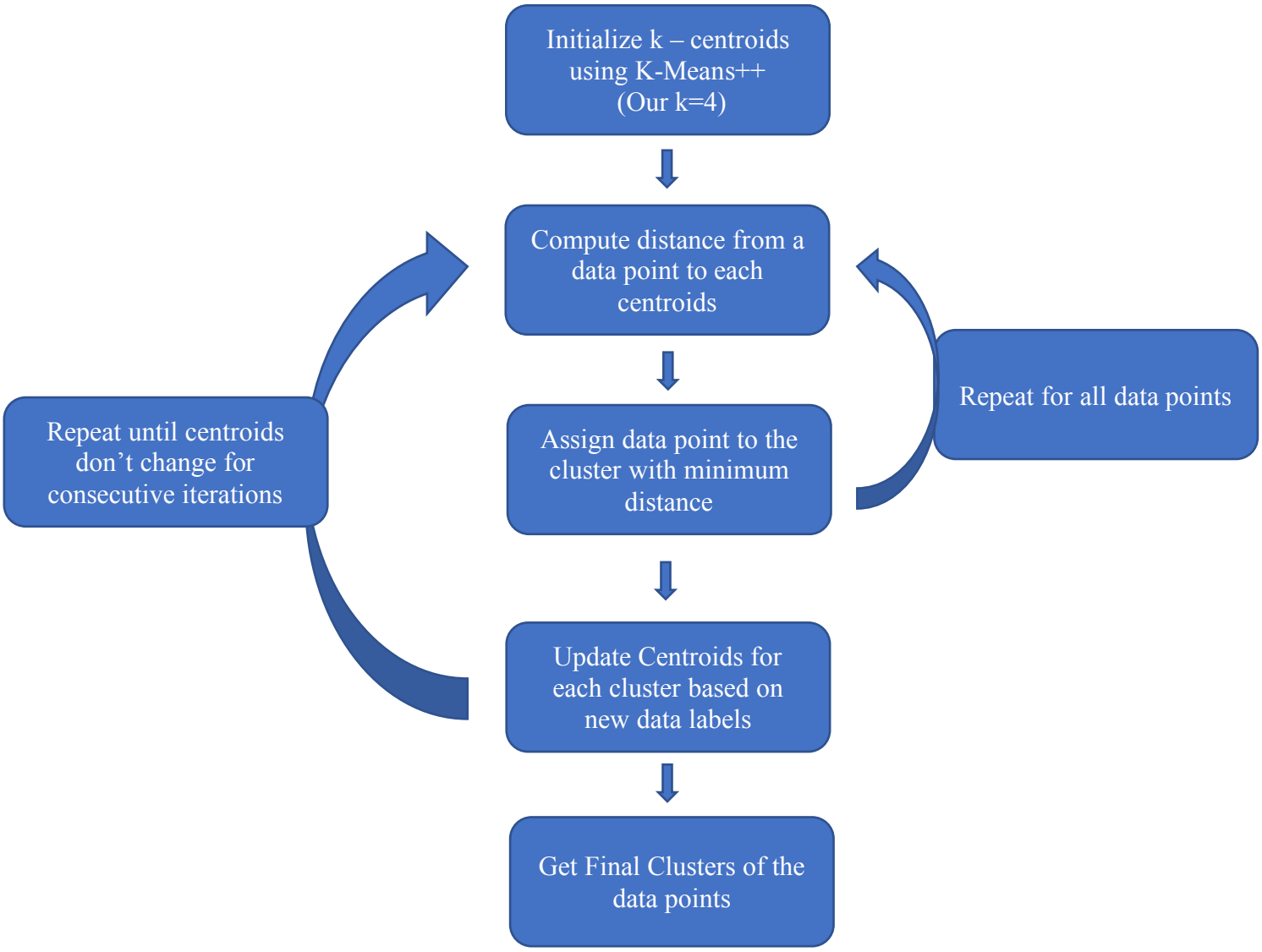
Sample Filters are shown below: (E5 tensor product with L5)

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

Texture in an image can be categorized in Medium and High Frequencies. For texture classification problem, E5, S5 and W5 filters were considered, since they are the ones that can filter Medium and High frequencies from an image. They were obtained by taking tensor product with each of the other filters to get nine 5*5 filters.



Flow chart for Texture Classification



K-Means Algorithm implemented for Clustering

Energy of an Image was Calculated by using the following formula:

$$\text{Energy} = \sum_{x=1}^{Row} \sum_{y=1}^{Col} (\text{abs}(f(x,y)))\text{^}2$$

Where, f(x,y) = Pixel Intensity,
Row = Number of Rows,
Col = Number of Columns

Algorithm Implemented (C++):

main() Function:

- Read given *Texture1.raw to Texture12.raw* images using *fileRead()* function
- Convert 1D images to 2D using *image1Dto2D()* function
- Define E5, S5 and W5 1D Masks
- Find Tensor Products to get 5*5 masks
- Stack all given 2D images to get 3D using *stackImages()*
- Declare object for *textureClassification Class*
- Call *subtractMean()* method to subtract global mean from all images
- Call *energyFeatureExtraction()* method
- Call *initializeKmeansCentroids()* method
- Call *kmeansClustering()* method
- Print the cluster labels for all images

textureClassification Class:

- Initialize all the required data structures
- Define Constructor and Destructor for the class

subtractMean() Function:

- Find mean of every image using two nested for loops
- Subtract image pixel intensity values from mean value

energyFeatureExtraction() Function:

- Apply 9 masks on every image
- Get energy values for every image
- Energy for an image as calculated as the summation of square of absolute values of all pixel values
- 9 energy values will be obtained for every image
- Repeat this for all 12 images
- Get 12*9 energy feature matrix

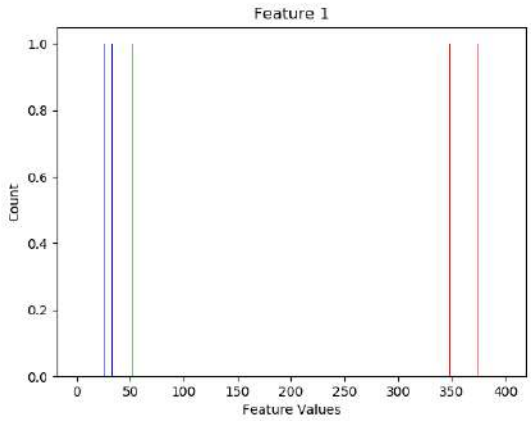
initializeKmeansCentroids() Function:

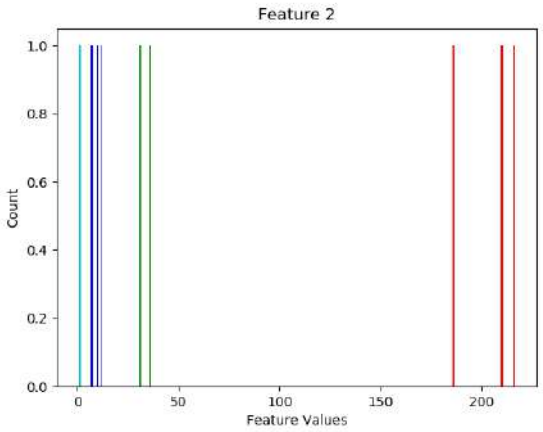
- Initialize using kmeans++ algorithm
- One centroid is initialized for every cluster

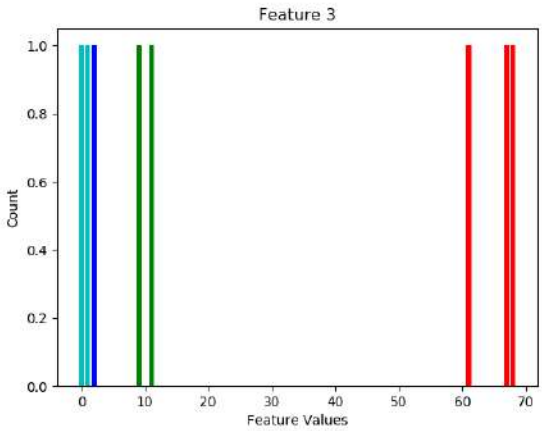
kmeansClustering() Function:

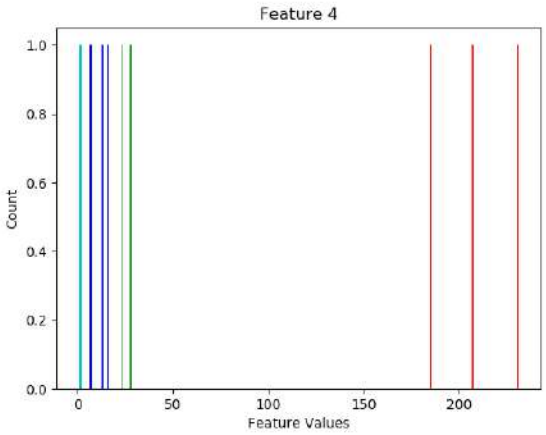
- Find Euclidian distance from every data point to every cluster
- Find the cluster which is closer to the data point
- Repeat this for all data points
- Assign labels to that cluster
- Update centroids
- Repeat for 20 iterations

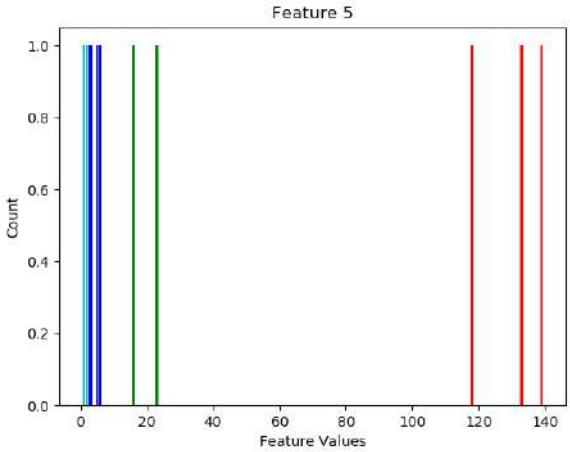
➢ EXPERIMENTAL RESULTS:

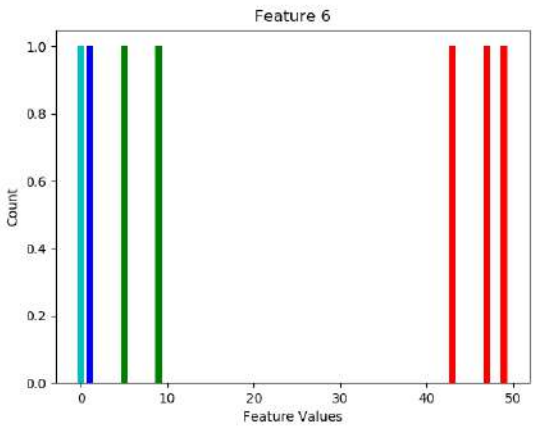


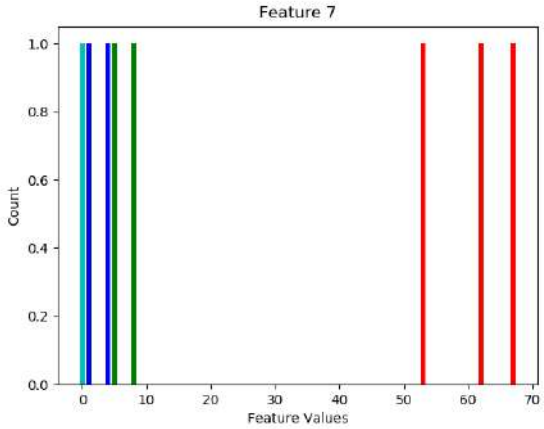


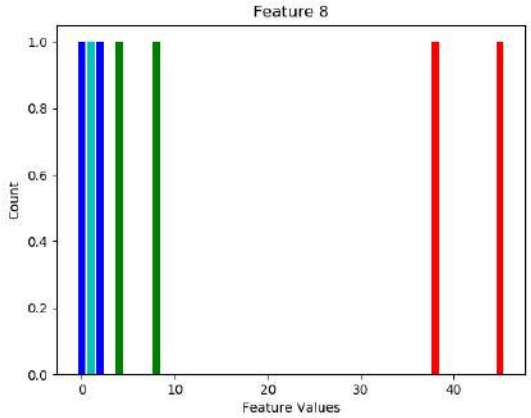


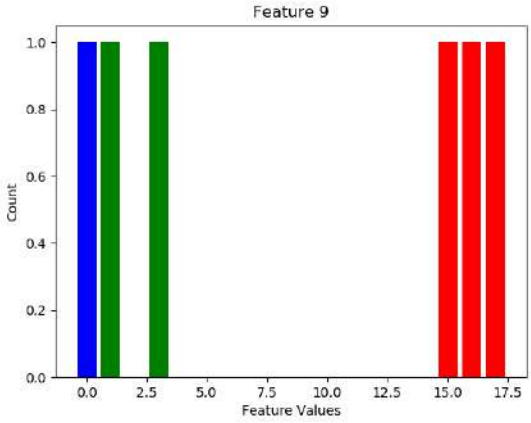




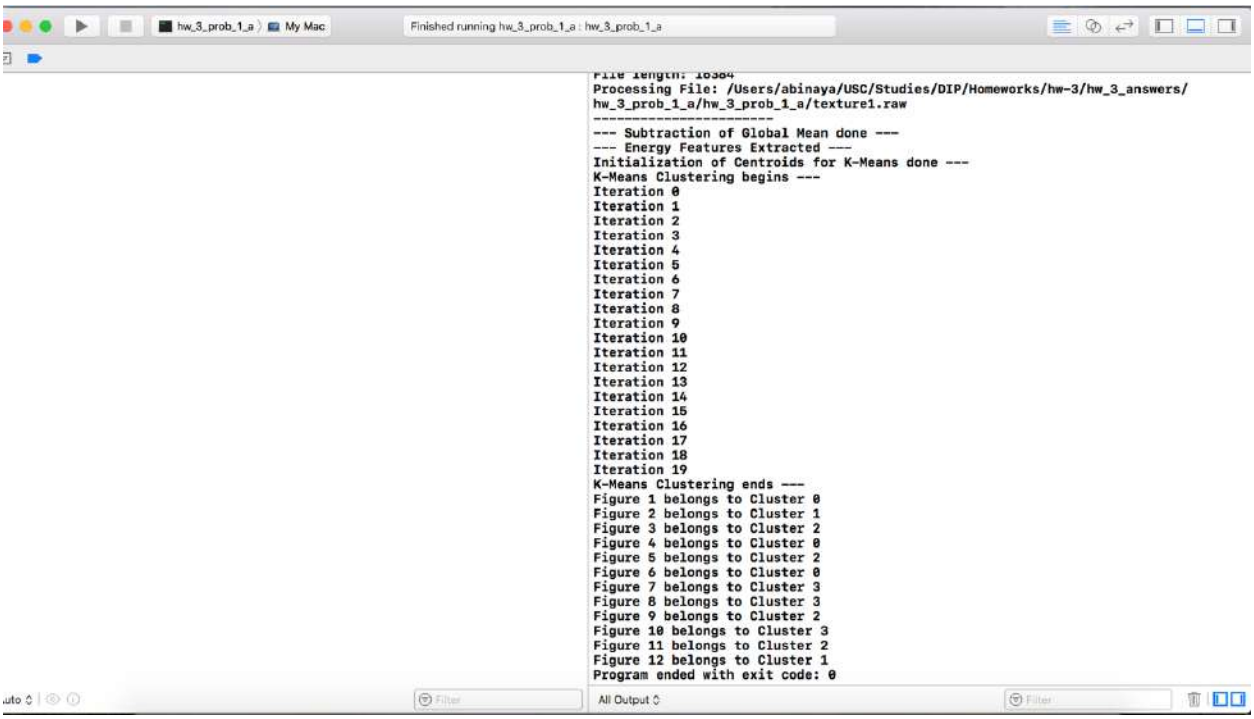








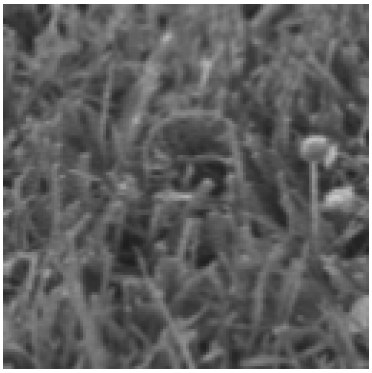
The above histograms are for 9 energy features and colors depend on ground truth clusters. Feature 1 (E5E5) has good discriminant power. Feature 2(E5S5) and 4(S5S5) is also good. Feature 8 is the least good one, since it mixes cluster 1 and 2 information. A joint pdf would show better results



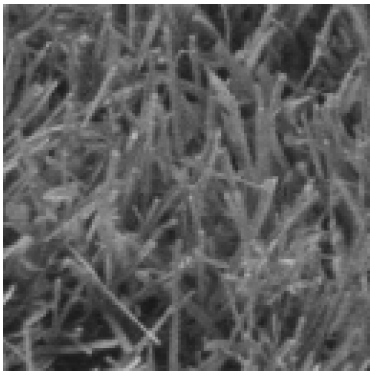
Cluster	Figure Nos
0	1,4,6
1	2,12
2	3,5,9,11
3	7,8,10

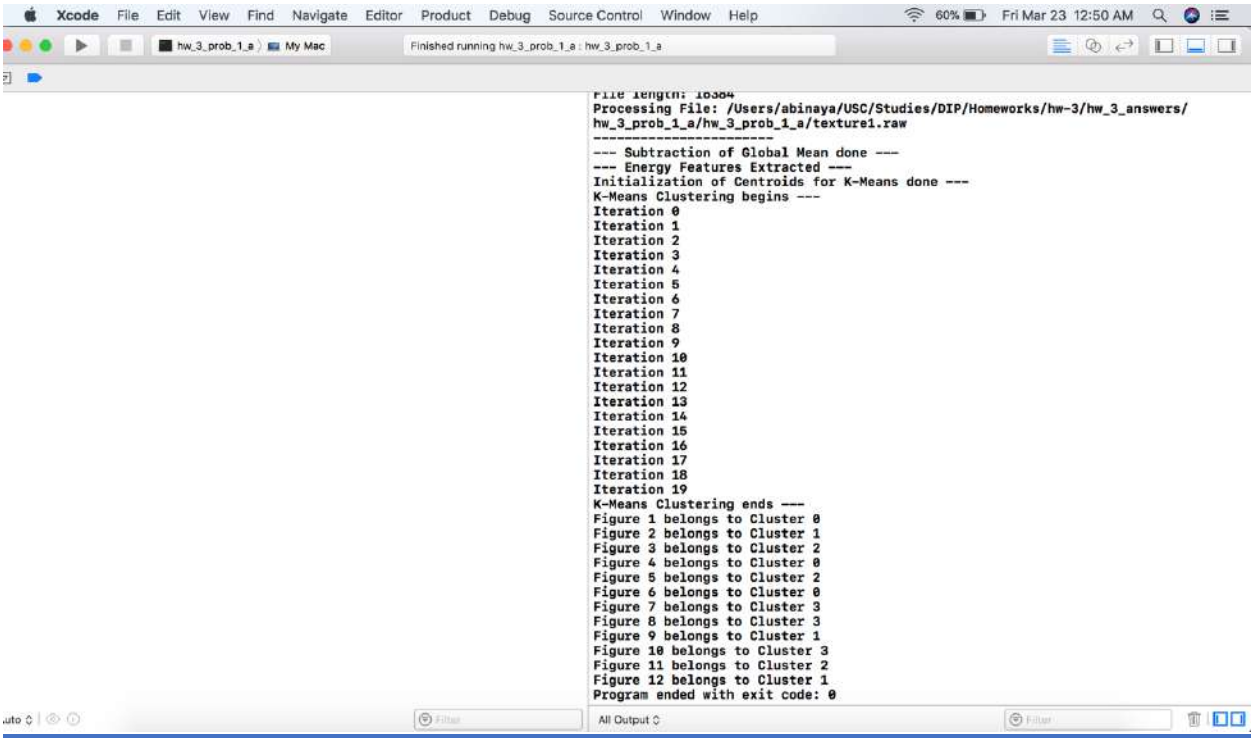
Output Table shows Figure 9 is misclassified. This is because Figure 9 is noisy. Noiseless of the same image is given to us as Figure 14. So, replaced Figure 9 with 14th image and same procedure was repeated.

Texture 9 Image - Noisy



Texture 14 Image - Noiseless image





Cluster	Figure Nos
0	1,4,6
1	2,9,12
2	3,5,11
3	7,8,10

➢ DISCUSSION:

- Thus, using the above procedure, energy feature vectors were obtained for every image using Laws Filters
- It is proved that Laws Filters help in Texture Feature Extraction from the image from the histograms provided. Every histogram has some information about clusters
- All the images were clustered based on K-Means Clustering
- As we can see from above, all images are clustered correctly
- Each cluster has 3 images clustered
- The above experiment results show that Texture feature extraction is very sensitive to Noise in an image
- Once noiseless image was considered, all images were clustered correctly

B) TEXTURE CLASSIFICATION

➢ ABSTRACT AND MOTIVATION:

The previous question gave us some hands-on experience on texture feature extraction using Laws Filters. Using the same concept, but instead of classifying images, we can classify pixels. Given pixel, after applying Laws Filters, we can use them for texture segmentation. Segmenting an image into different textures, is similar to classifying each pixel of an image to different cluster. The given image has complex mixed textures. Applying similar concept of Laws texture feature extraction and K-Means, we can see how good the given complex image gets segmented.

➢ APPROACH AND PROCEDURES:

This problem is approached by applying Laws Filters on the Images. The 3 Laws Filters considered for this question are given below.

Filter	Kernel
Level (L3)	[1 2 1] * 1/6
Edge (E3)	[-1 0 1] * 1/2
Spot (S3)	[-1 2 -1] *1/2

A 3*3 filter for image texture filtering can be done by tensor product of any 2 filters above. Therefore, using the above three 1D filters, 9 masks can be obtained.

(Source: Digital Image Processing, William K.Pratt)

$$\frac{1}{36}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$ Laws 1

$$\frac{1}{12}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$ Laws 2

$$\frac{1}{12}\begin{bmatrix} -1 & 2 & -1 \\ -2 & 4 & -2 \\ -1 & 2 & -1 \end{bmatrix}$$ Laws 3

$$\frac{1}{12}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$ Laws 4

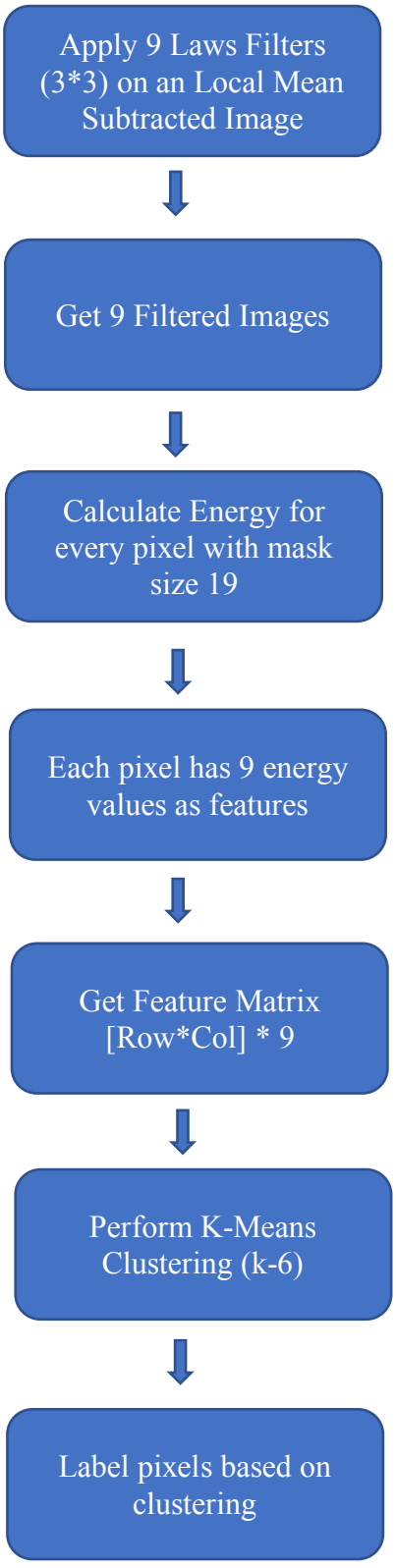
$$\frac{1}{4}\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$ Laws 5

$$\frac{1}{4}\begin{bmatrix} -1 & 2 & -1 \\ 0 & 0 & 0 \\ 1 & -2 & 1 \end{bmatrix}$$ Laws 6

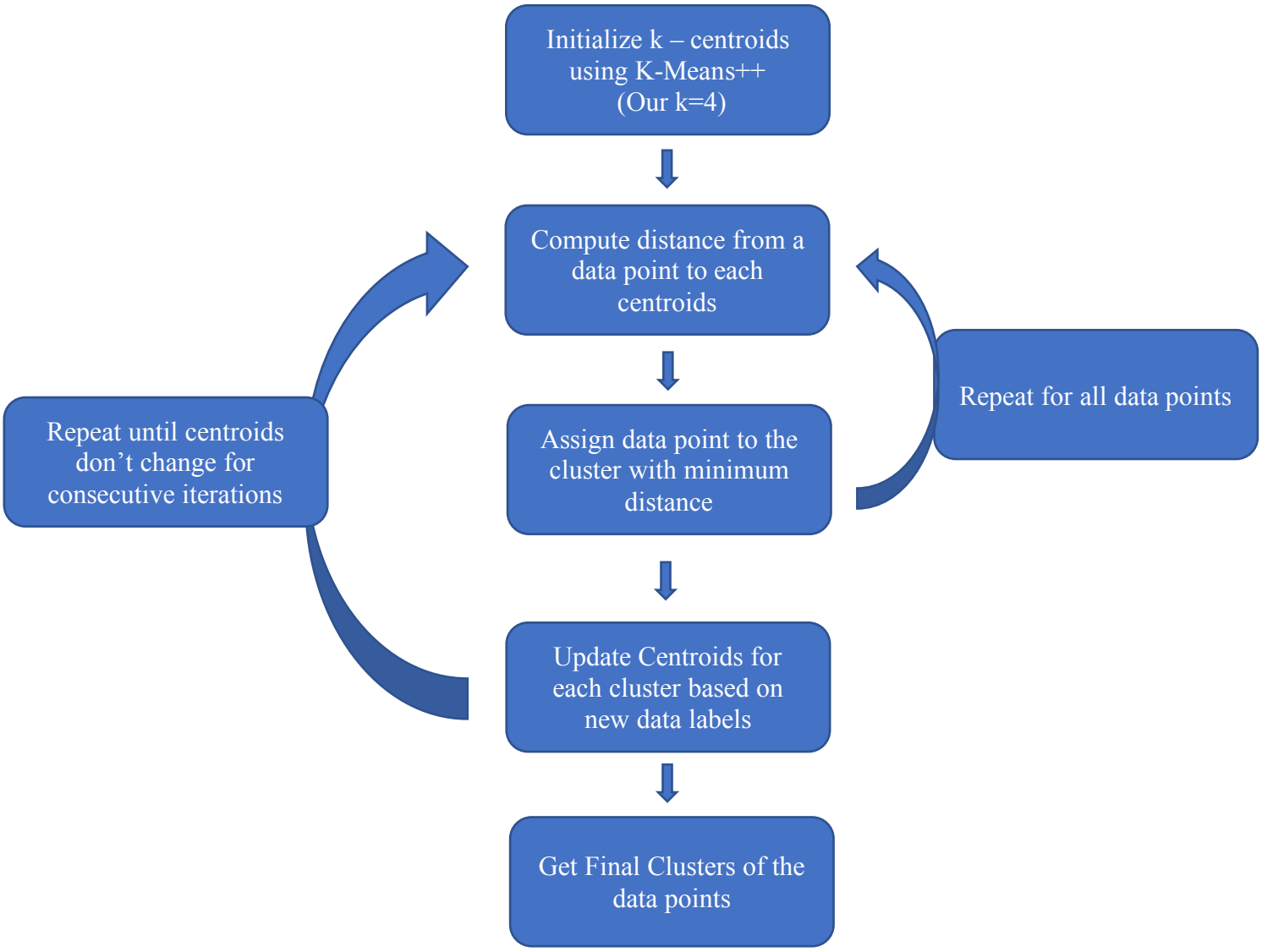
$$\frac{1}{12}\begin{bmatrix} -1 & -2 & -1 \\ 2 & 4 & 2 \\ -1 & -2 & -1 \end{bmatrix}$$ Laws 7

$$\frac{1}{4}\begin{bmatrix} -1 & 0 & 1 \\ 2 & 0 & -2 \\ -1 & 0 & 1 \end{bmatrix}$$ Laws 8

$$\frac{1}{4}\begin{bmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$ Laws 9



Flow chart for Texture Segmentation



K-Means Algorithm implemented for Clustering

Energy of an Image was Calculated by using the following formula:

$$\text{Energy} = \sum_{x=1}^{N}\sum_{y=1}^{N}(\text{abs}(f(x,y)))\text{^}2$$

Where, f(x,y) = Pixel Intensity,
N = mask size

Algorithm Implemented (C++):

main() Function:

- Read given *Composite.raw* image using *fileRead()* function
- Convert 1D images to 2D using *image1Dto2D()* function
- Define L3, E3 and S3 1D Masks
- Find Tensor Products to get 3*3 masks
- Declare object for *textureSegmentation Class*
- Call *subtractLocalMean()* method to subtract global mean from all images
- Call *energyFeatureExtraction()* method
- Call *initializeKmeansCentroids()* method
- Call *kmeansClustering()* method
- Call *clustersToGrayLevels()* method
- Convert 2D segmented image to 1D and write to file using *fileWrite()*
- Delete all allocated memories

textureClassification Class:

- Initialize all the required data structures
- Define Constructor and Destructor for the class

subtractMean() Function:

- Find mean of every pixel using two nested for loops with desired mask size
- Subtract image pixel intensity values from mean value

energyFeatureExtraction() Function:

- Apply 9 masks on every image
- Get energy values for every pixel
- Energy for an pixel as calculated as the summation of square of absolute values of all surrounding N*N pixel values
- 9 energy values will be obtained for every pixel
- Repeat this for all (Row*Col) pixels
- Get (Row*Col)*9 energy feature matrix

initializeKmeansCentroids() Function:

- Initialize using kmeans++ algorithm
- One centroid is initialized for every cluster

kmeansClustering() Function:

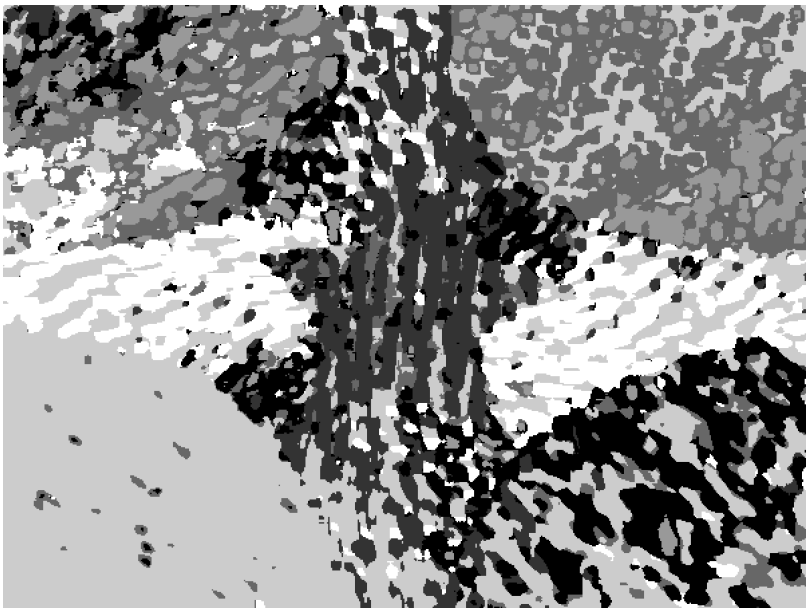
- Find Euclidian distance from every data point to every cluster
- Find the cluster which is closer to the data point
- Repeat this for all data points
- Assign labels to that cluster
- Update centroids
- Repeat for 20 iterations

clustersToGrayLevels() Function:

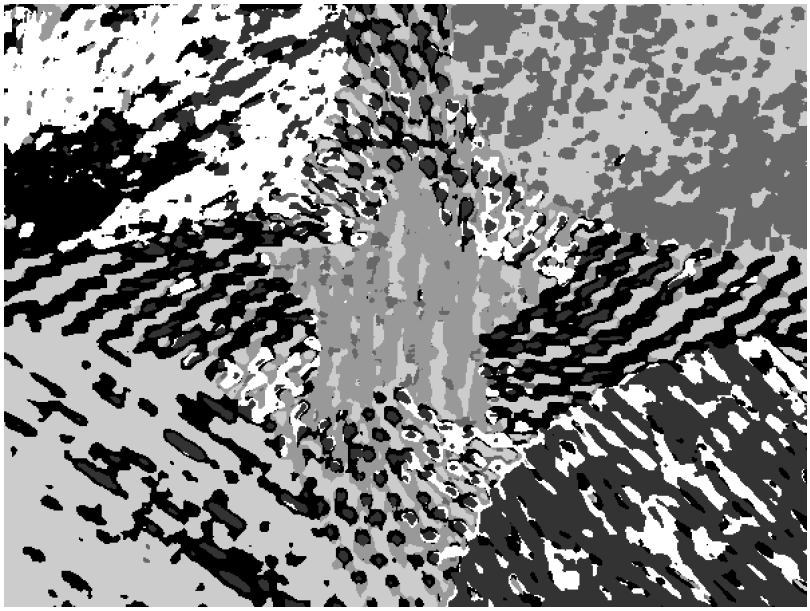
- There are 6 labels and 6 gray levels are chosen for viewing purposes.
- (0,51,102,153,204,255) are chosen and converted using two nested-for loops

➢ EXPERIMENTAL RESULTS:

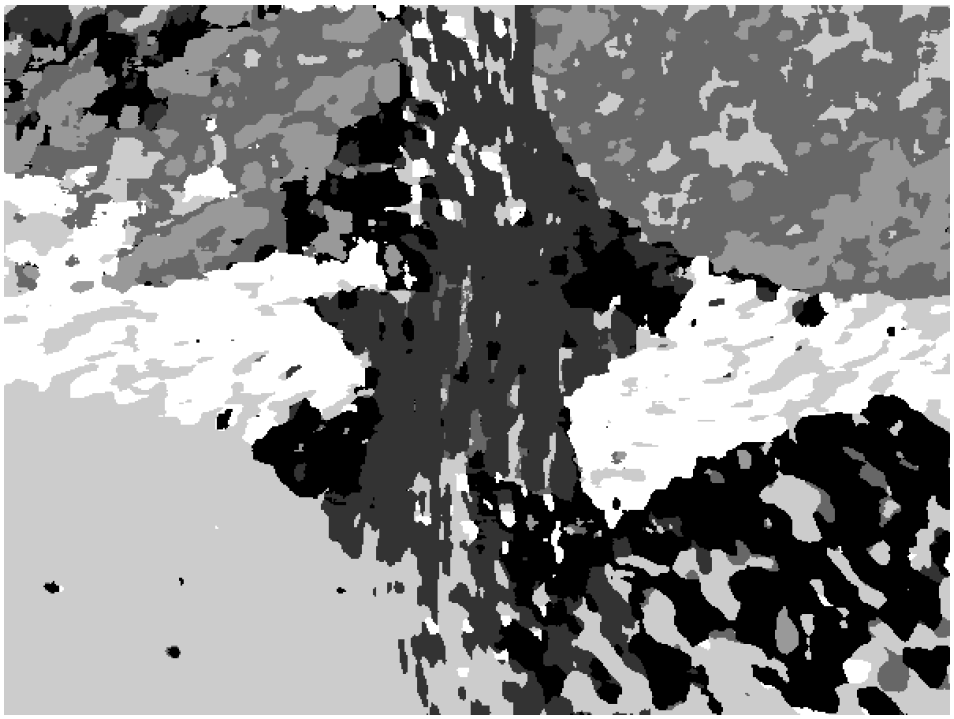
Mask Size = 9 - Normalized



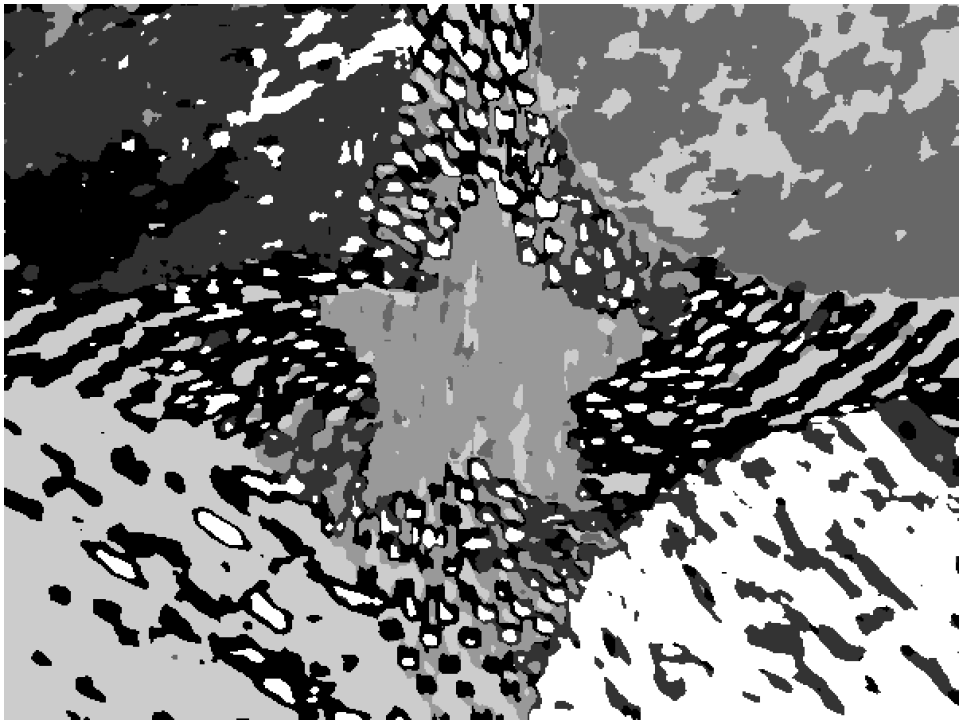
Mask Size = 9 - Without Normalization



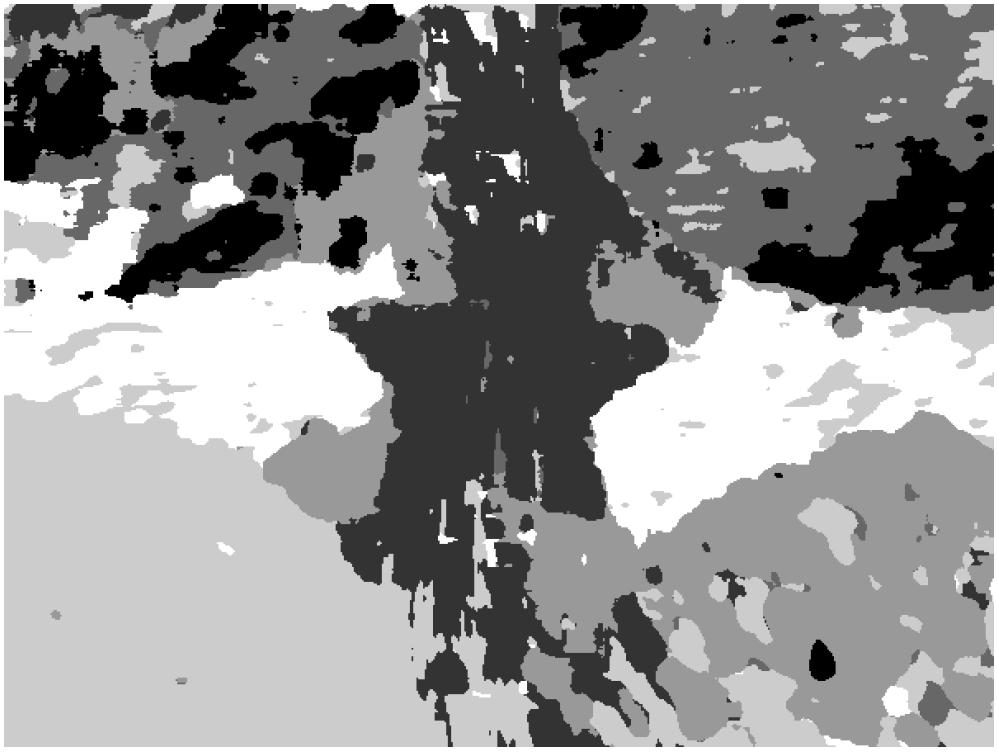
Mask Size = 13 - Normalized



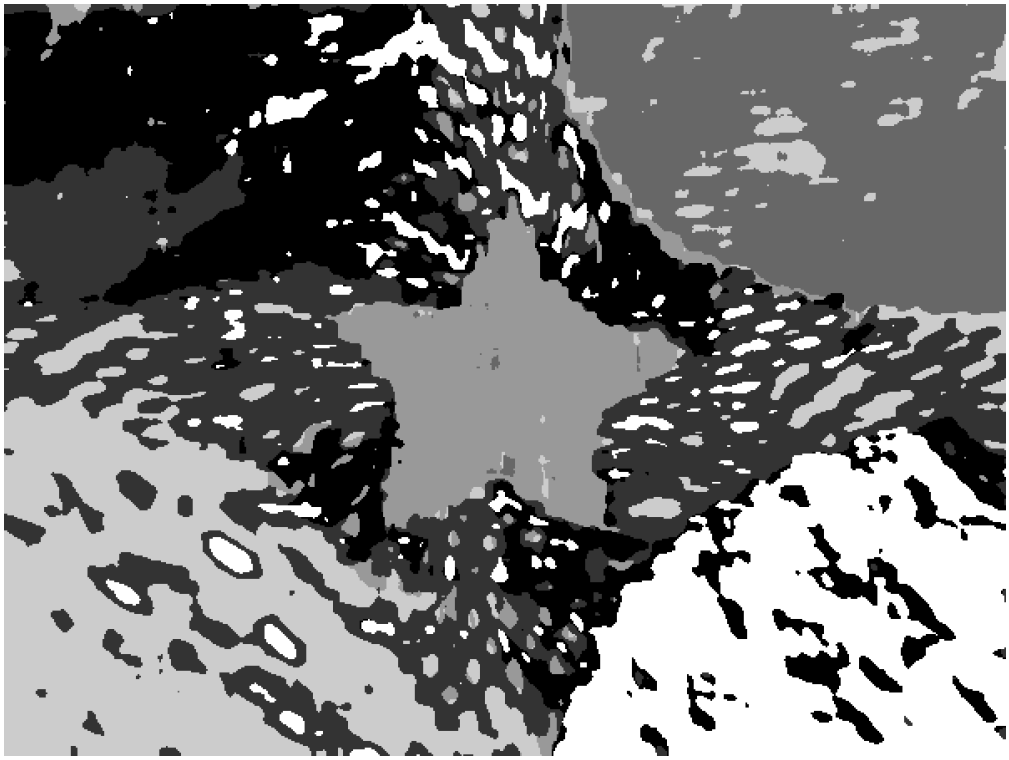
Mask Size = 13 - Without Normalization



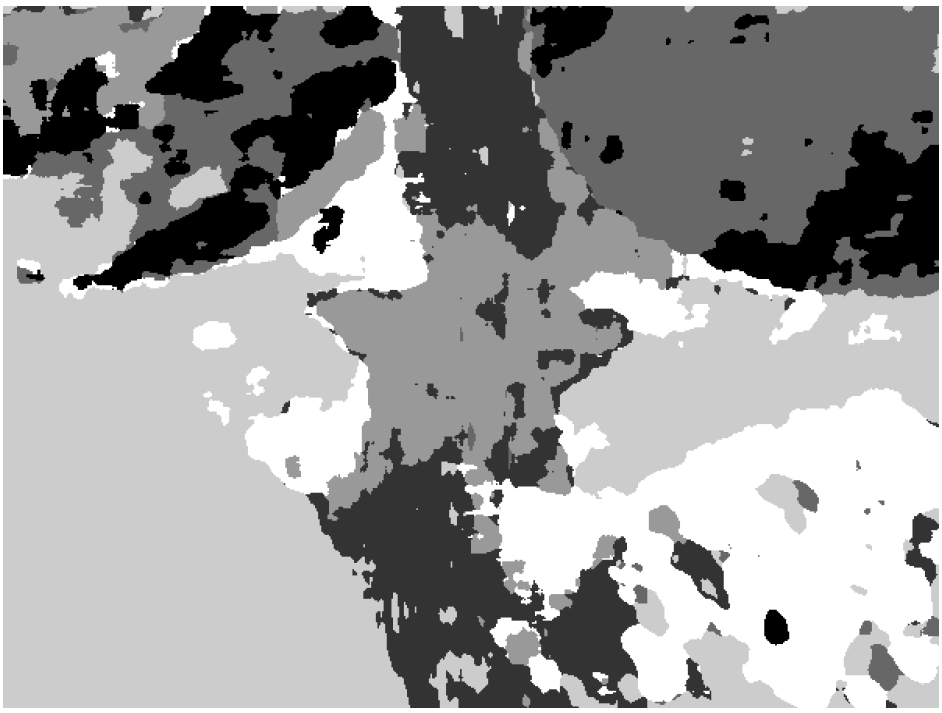
Mask Size = 17 - Normalized



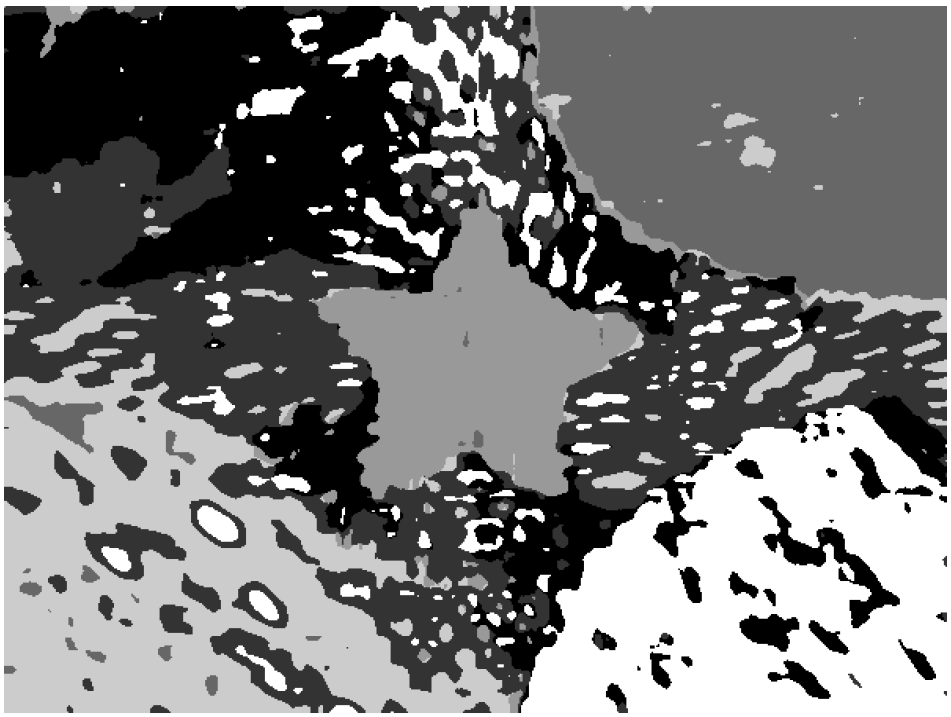
Mask Size = 17 - Without Normalization



Mask Size = 19 - Normalized



Mask Size = 19 - Without Normalization



➢ DISCUSSION:

- Thus, using the above-mentioned procedure using Laws Filter for texture feature extraction, I segmented the given input Composite.raw image into 6 segments.
- Various mask sizes were done on trial and error basis for:
    - Local Mean Subtraction
    - Energy Feature Extraction for a pixel
- Upon trial and error basis mentioned for the above two process I figured out that for Mask Size = 13, the results have clear boundaries, but distorted regions inside
- Also, based on the above results, normalization of features based on L3L3 is actually not giving good results. Without Normalization, the results are far better
- As I increased the Mask Size, the inside regions of every segment were less distorted, but the boundaries are averaged out (i.e. boundaries are mixed and not clear). This is seen clearly in the last image of Mask Size = 19. The regions are clear when compared to Mask Size 13, but the boundaries are mixed especially the top left one
- Thus, it is hard to say which result is better, but one can use Mask Size 13 or 19, depending upon the application needs

C) IMPROVED TEXTURE SEGMENTATION:

➢ ABSTRACT AND MOTIVATION:

This part of the question is exploratory, and we were allowed to try different Texture Segmentation improvement techniques. I have tried Dimensionality Reduction – Principal Component Analysis to improve K-Means clustering performance. The same procedure was carried out until feature extraction. Laws Texture Feature Extraction method was used to get energy feature values for every pixel. Then PCA was done on the extracted features using MATLAB. The dimension reduced features were used for K-Means Clustering. Detailed algorithm and experimentation results are given below.

➢ APPROACH AND PROCEDURES:

This problem is approached by applying Laws Filters on the Images. There are 5 Laws Filters available. They are listed below.

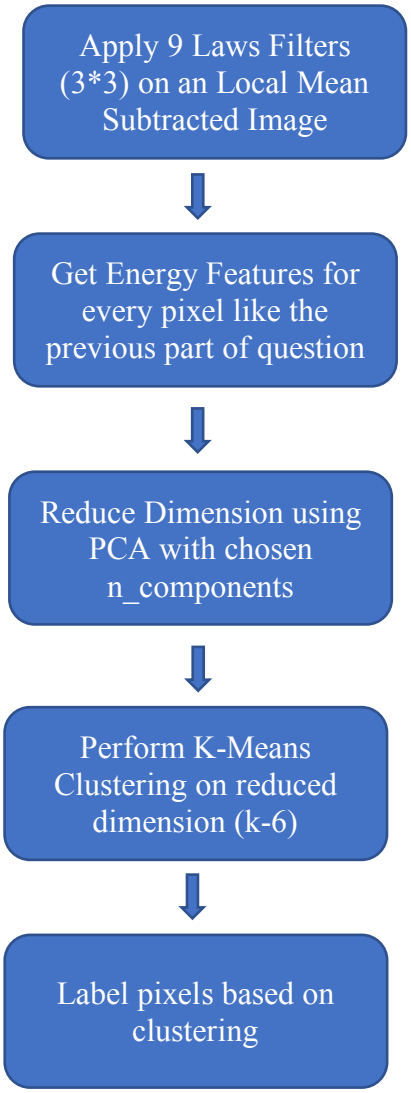
Filter	Kernel
Level (L5)	[1 4 6 4 1]
Edge (E5)	[-1 -2 0 2 1]
Spot (S5)	[-1 0 2 0 -1]
Wave (W5)	[-1 2 0 -2 1]
Ripple (R5)	[1 -4 6 -4 1]

A 5*5 filter for image texture filtering can be done by tensor product of any 2 filters above. Therefore, using the above five 1D filters, 25 masks can be obtained.

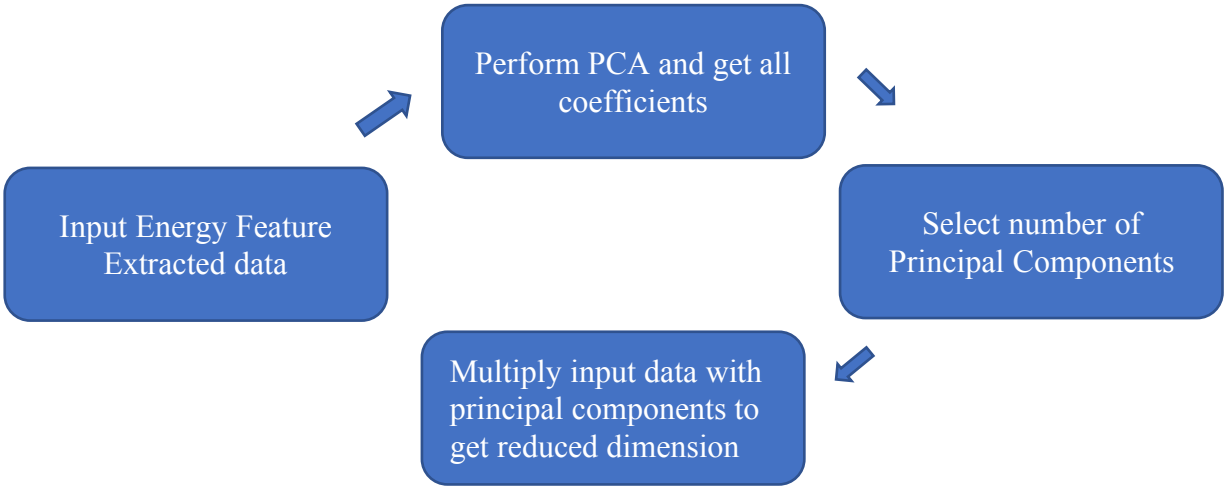
Sample Filters are shown below: (E5 tensor product with L5)

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

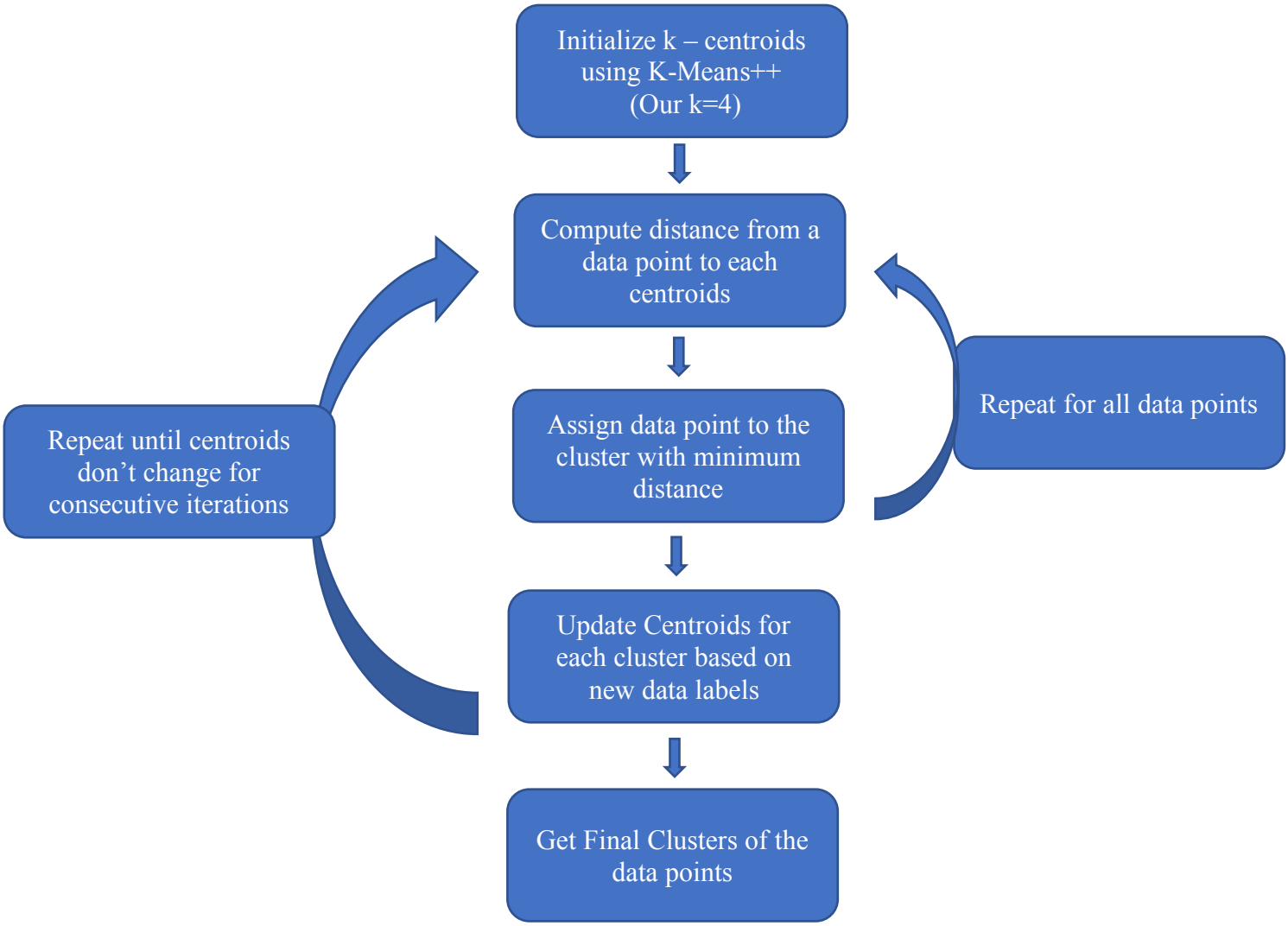
Texture in an image can be categorized in Medium and High Frequencies. For this part of the problem, all 5 filters were considered. They were obtained by taking tensor product with each of the other filters to get nine 5*5 filters.



Flow chart for Texture Segmentation with PCA



PCA Algorithm for Dimensionality reduction



K-Means Algorithm implemented for Clustering

Algorithm Implemented (C++):

main() Function:

- Read given *Composite.raw* image using *fileRead()* function
- Convert 1D images to 2D using *image1Dto2D()* function
- Define L3, E3 and S3 1D Masks
- Find Tensor Products to get 3*3 masks
- Declare object for *textureSegmentation Class*
- Call *subtractLocalMean()* method `to subtract global mean from all images
- Call *energyFeatureExtraction()* method
- Write energy feature 2D pointer to a text file using *fileWriteEnergy()*
- Read back reduced dimenstion array using *fileReadEnergy()*
- Call *initializeKmeansCentroids()* method
- Call *kmeansClustering()* method
- Call *clustersToGrayLevels()* method
- Convert 2D segmented image to 1D and write to file using *fileWrite()*
- Delete all allocated memories

textureClassification Class:

- Initialize all the required data structures
- Define Constructor and Destructor for the class

subtractMean() Function:

- Find mean of every pixel using two nested for loops with desired mask size
- Subtract image pixel intensity values from mean value

energyFeatureExtraction() Function:

- Apply 9 masks on every image
- Get energy values for every pixel
- Energy for an pixel as calculated as the summation of square of absolute values of all surrounding N*N pixel values
- 9 energy values will be obtained for every pixel
- Repeat this for all (Row*Col) pixels
- Get (Row*Col)*9 energy feature matrix

initializeKmeansCentroids() Function:

- Initialize using kmeans++ algorithm
- One centroid is initialized for every cluster

kmeansClustering() Function:

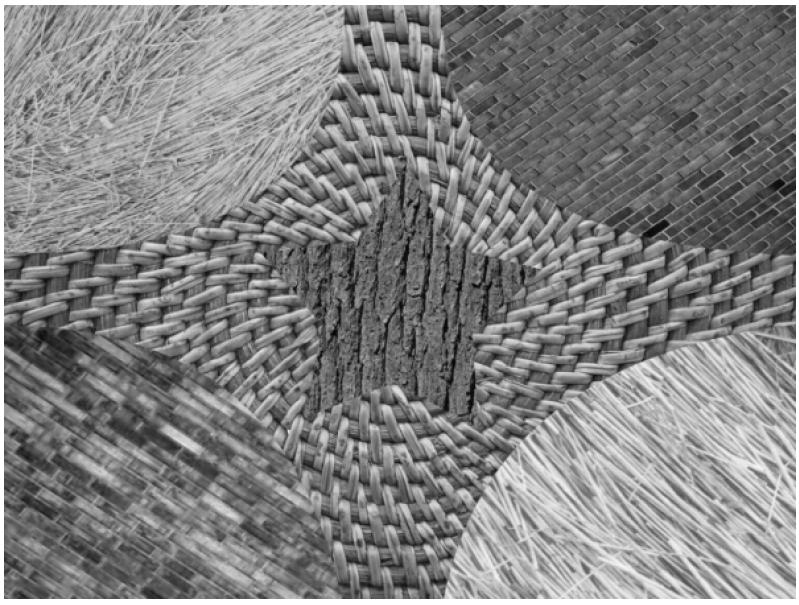
- Find Euclidian distance from every data point to every cluster
- Find the cluster which is closer to the data point
- Repeat this for all data points
- Assign labels to that cluster
- Update centroids
- Repeat for 20 iterations

clustersToGrayLevels() Function:

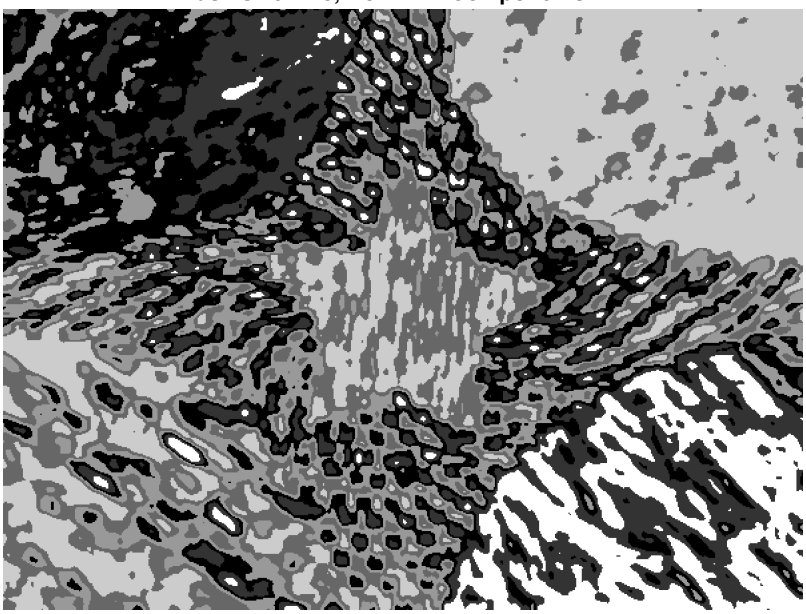
- There are 6 labels and 6 gray levels are chosen for viewing purposes.
- (0,51,102,153,204,255) are chosen and converted using two nested-for loops

➢ EXPERIMENTAL RESULTS:

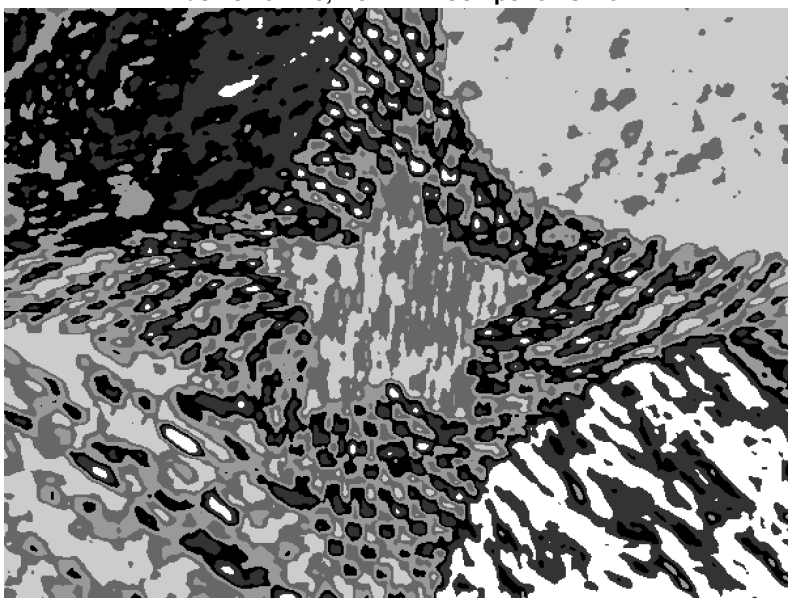
Composite Image - Input



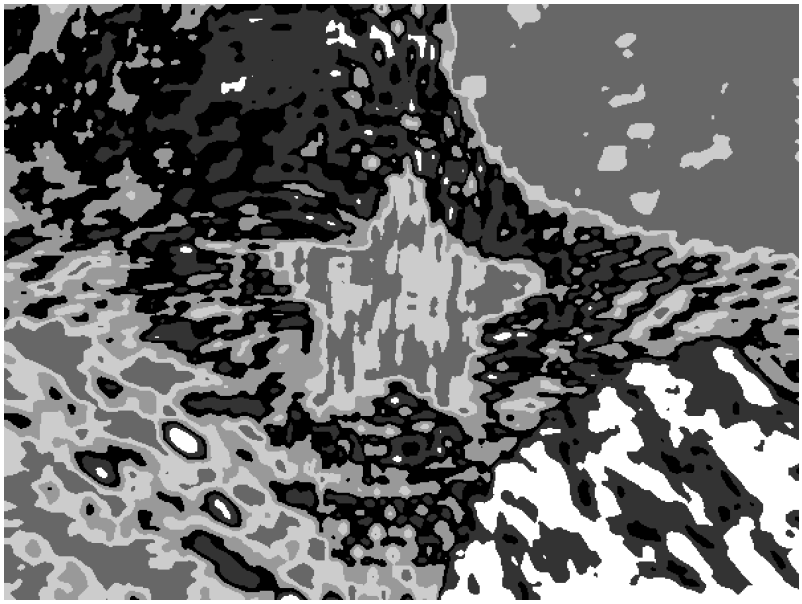
Mask Size = 13, PCA with Components = 2



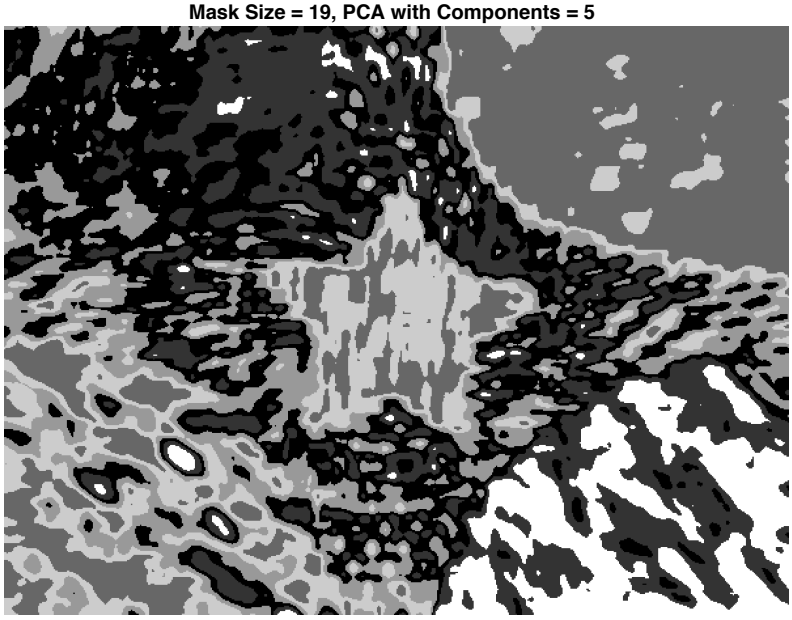
Mask Size = 13, PCA with Components = 5



Mask Size = 19, PCA with Components = 2



Mask Size = 19, PCA with Components = 5



- **DISCUSSION:**

Why PCA before Clustering should improve the results:

- Reduces dimension of data
- Data gets transformed using orthogonal transformation and given a set of highly uncorrelated variables called principal components suppressing correlated variables in the data
- The final data with selected principal components are the ones with high variance and possibly should have clear clusters than data with correlated variables
- Also, since the dimension is less, computation time for K-Means clustering will be drastically reduced

Texture Segmentation using PCA and Clustering:

- Various Outputs are given with different Principal Components chosen. I chose n_components = 2 and 5
- The above number of components were applied on Mask Size = 13 and 19
- I believe the output with Mask Size = 13 and Principal Components = 2 is the best output overall
- It has boundaries well defined and regions also less distorted
- Even the patterns inside the regions are well visible
- And, as expected computation time for K-Means clustering with PCA reduced dimension of energy features was very less
- Though the improvement is not very great, the computational time was brought down. A comparative study of various methods likeGabor filters (GF), Gaussian Markov random fields (GMRF), run-length matrix (RLM) and co-occurrence matrix (GLCM) is shown here. We can adopt those methods to improve segmentation. (Source: https://arxiv.org/abs/1601.00212)

PROBLEM 2
EDGE DETECTION

D) BASIC EDGE DETECTION:

➢ ABSTRACT AND MOTIVATION:

Edge detection is the most basic and frequently used Computer Vision Algorithm. Edge detection is determining the boundaries of objects in a given image. Although, image processing algorithms have developed a lot over the years, edge detection is still a vital research area. Many algorithms have been developed to detect edges from noisy and noiseless images effectively. From basic feature extraction methods by studying the image to finding edges using Convolutional Neural Networks exist and people use them in different applications. The whole purpose of this question is to detect edges from an image using different methods and understand the difference between the algorithms and performances. In this part of the question, I have implemented Sobel and Zero Crossing Edge detector and have compared the performances.

➢ APPROACH AND PROCEDURES:

Sobel Edge Detection:

Sobel Edge detection is done by differentiating the pixel values along X and Y direction using Sobel Edge Masks. The magnitude and direction of the pixel gradient is found from X and Y gradient values. Sobel Edge Masks along X and Y direction are given below.
(Source: EE 569 Discussion 9)

-1	0	+1
-2	0	+2
-1	0	+1

Gx

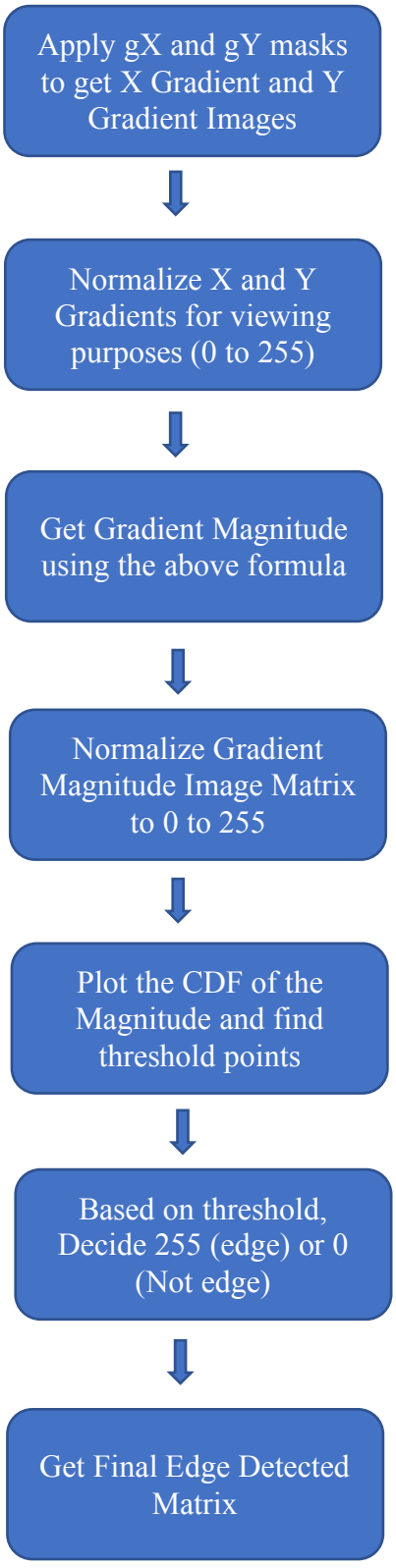
+1	+2	+1
0	0	0
-1	-2	-1

Gy

Magnitude and Orientation

$$\sqrt{g_y^2 + g_x^2} \qquad \theta = \tan^{-1}\left[\frac{g_y}{g_x}\right], \qquad \nabla f = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

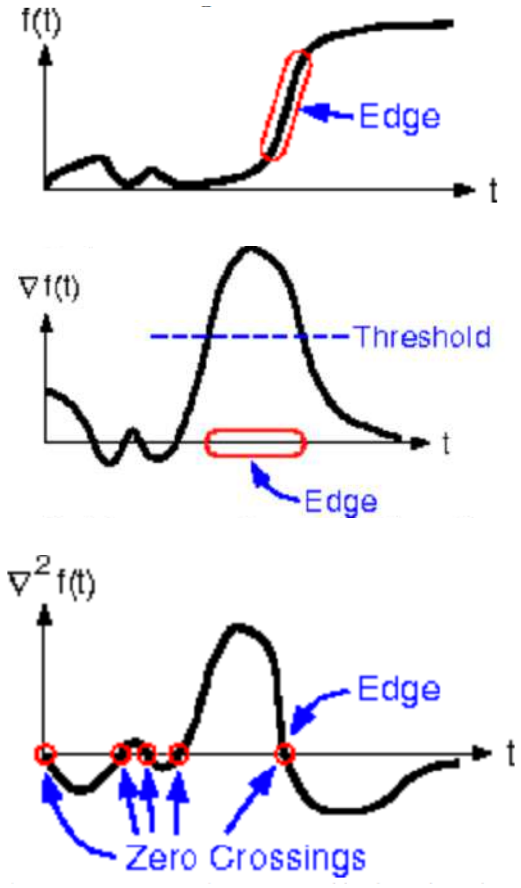
- After finding the magnitude of gradient every pixel, they are thresholded to find edges. Thresholding is done on scaled values of Magnitude between 0 to 255
- The thresholds are chosen by trial and error basis by looking into the Cumulative Histogram plot
- Top tail of values is considered as edge since the magnitude of gradient would be high for edges



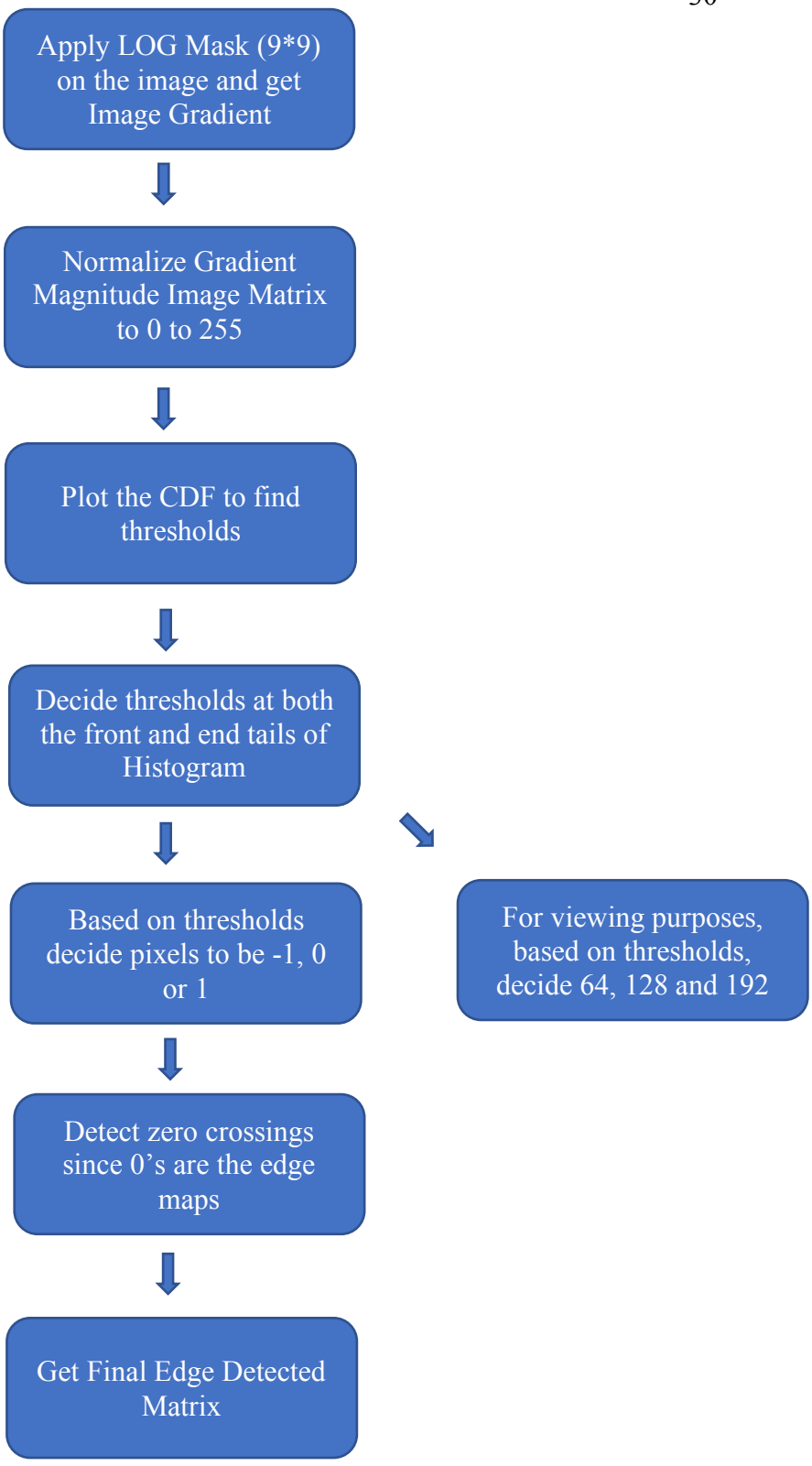
Sobel Edge Detection Flow Chart

Zero Crossing Edge Detection:

- Edge detection as done in previous Sobel Edge Detection, the first order derivation would have to be thresholded. Choosing of an optimal threshold would be very difficult. It will have to be done manually to get a good edge map based on the application
- Zero Crossing Edge detection uses the concept that a second order derivative of an edge would cross zero and it is easier to detect. An illustration is shown below (Source: http://www.owlnet.rice.edu/~elec539/Projects97/morphjrks/laplacian.html)



- A Gaussian Mask is used before taking second order derivative to remove noise if any in the image. Laplacian Mask is used for taking second order derivative of the image
- We can apply both of the mask separately or we can apply a single mask by convolving the 2 masks



Zero Crossing based on LOG Edge Detection Flow Chart

The Mask that I have tried for this question is given below:
(Source: http://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm?utm_source=www.uoota.com)

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

The above Mask is an LOG Mask of 9*9 with Gaussian sigma = 1.4

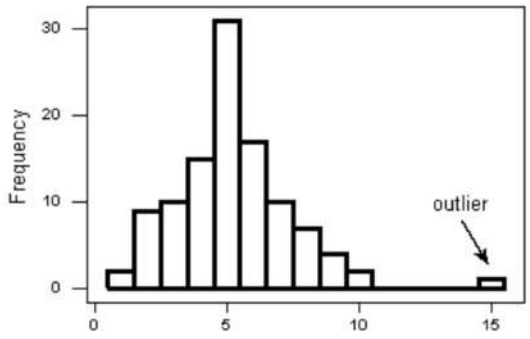
To normalize/scale the gradient magnitudes, the below method is used:

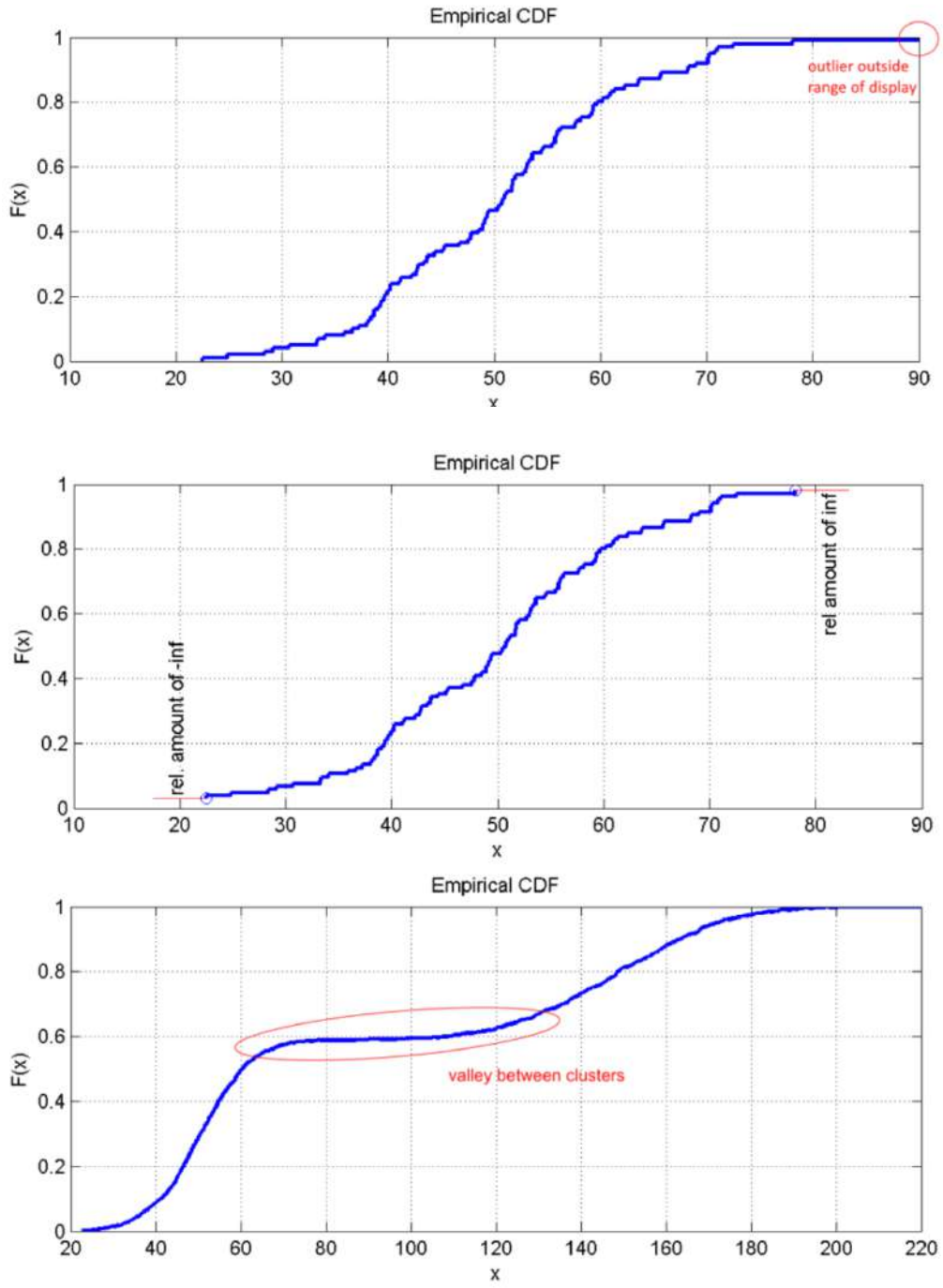
$$g(x,y) = ((f(x,y) - min\_value) / (max\_value - min\_value)) * 255$$

where, f(x,y) = pixel value at x, y location
min_value = minimum value of gradient magnitudes
max_value = maximum value of gradient magnitudes

Analyzing Histogram: (Algorithm to determine thresholds)

- In a distribution, for example Gaussian, knee points are points that lie in both the sides of the histogram, where the slope of the CDF curve changes rapidly.
- These Knee points are typically considered as thresholds to detect outliers.
- I have shown a few histograms below to understand what a knee point is and how points beyond them are considered as outliers.





(Source: http://www.andata.at/en/software-blog-reader/why-we-love-the-cdf-and-do-not-like-histograms-that-much.html)

- The first image shows that there are outliers in the PDF.
- These outliers get mapped after Knee points in CDF and hence they can be detected from CDF knee points
- In out Edge Detection, Edge points are considered as outliers and this algorithm is what I have followed to find both the thresholds
- I have tried various knee points like, 5% and 95%, 8% and 92%, 10% and 90%

Algorithm Implemented (C++) – Sobel Edge Detection:

main() Function:

- Read given *Boat,raw and Boat_noisy.raw* images using *fileRead()* function
- Convert 1D images to 2D using *image1Dto2D()* function
- Define gX and gY Mask
- Call convertToGrayScale*()* Function
- Declare object for *sobelEdgeDetection Class* for boat.raw image
- Call *getGradients()* method
- Call *getGradientMagnitude()* method
- Call *normalizeGradients*() method
- Call *thresholdForEdges()* method
- Repeat the same procedure for boat_noisy.raw image
- Write all the outputs
- Free all allocated memories

sobelEdgeDetection Class:

- Initialize all the required data structures
- Define Constructor and Destructor for the class

convertToGrayScale() Function:

- Loop through every r,g,b pixel using 2 nested for loops
- Calculate the luminosity of grayscale using Luminosity method

getGradients() Function:

- Loop through every pixel image using 2 nested for loops
- Get the 3*3 surrounding pixels for a given pixel using another 2 nested for loops
- Multiply the acquired pixels using gX and gY masks
- Get X and Y gradient value for every pixel

getGradientMagnitude() Function:

- Loop through every pixel using 2 Nested for loops
- Find the magnitude of the pixel using the above given formula from X gradient and Y gradient values

normalizeGradients() Function:

- Find minimum and maximum values for X Gradient, Y Gradient and Gradient Magnitude images
- Normalize all the pixel values using the above formula

thresholdForEdges() Function:

- Get the threshold value to be used
- Loop through every pixel using 2 nested for loops
- Based on the threshold value decide, the value to be 0 or 255

Algorithm Implemented (C++) – Zero Crossing Edge Detection:

main() Function:

- Read given *Boat,raw and Boat_noisy.raw* images using *fileRead()* function
- Convert 1D images to 2D using *image1Dto2D()* function
- Define LOG Mask
- Declare object for *zeroCrossingEdgeDetection Class* for boat.raw image
- Call convertToGrayScale*()* method
- Call *getGradientMagnitude()* method
- Call *normalizeGradients()* method
- Call *getDoubleThresholdedImage*() method
- Call *detectZeroCrossing()* method
- Repeat the same procedure for boat_noisy.raw image
- Write all the outputs and Free all allocated memories

zeroCrossingEdgeDetection Class:

- Initialize all the required data structures
- Define Constructor and Destructor for the class

convertToGrayScale() Function:

- Loop through every r,g,b pixel using 2 nested for loops
- Calculate the luminosity of grayscale using Luminosity method

getGradientMagnitude () Function:

- Loop through every pixel image using 2 nested for loops
- Get the 3*3 surrounding pixels for a given pixel using another 2 nested for loops
- Multiply the acquired pixels using LOG mask
- Get LOG gradient value for every pixel

normalizeGradients() Function:

- Find minimum and maximum values for LOG Gradient image
- Normalize all the pixel values using the above formula

getDoubleThresholdedImage() Function:

- Get two threshold values to be used
- Loop through every pixel using 2 nested for loops
- Based on the threshold value decide, the value to be -1, 0 or 1

detectZeroCrossing() Function:

- Traverse through every pixel using 2 nested for loops
- Check if the pixel is not zero
- Get its surrounding 3*3 pixels using 2 nested for loops
- Even if one element of the surrounding pixel is not zero, assign the pixel as 255

➢ EXPERIMENTAL RESULTS:

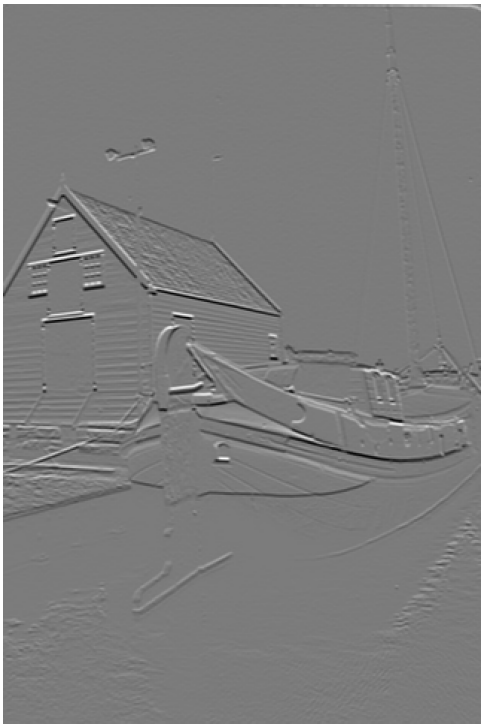
Input Boat Image

Input Noisy Boat Image

Sobel Edge Detection (Boat.raw image)

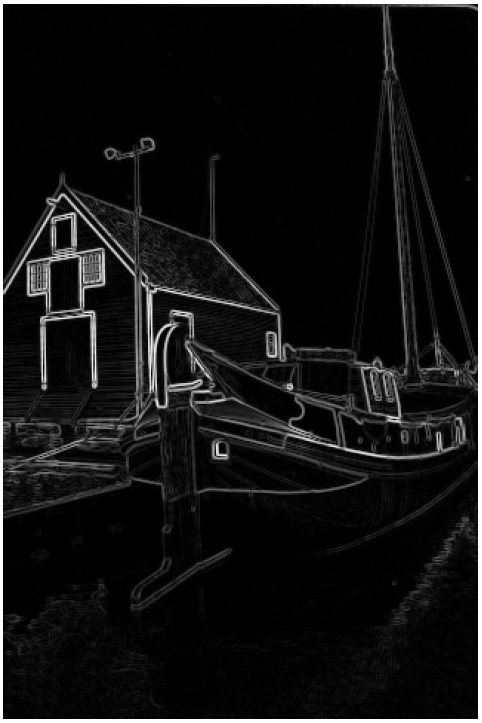
X Gradient - Normalized

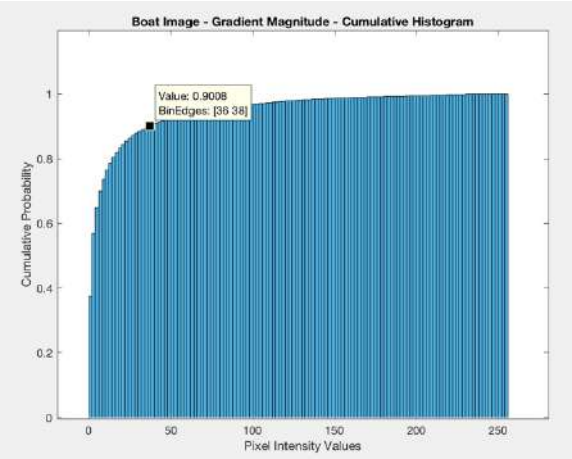
Y Gradient - Normalized



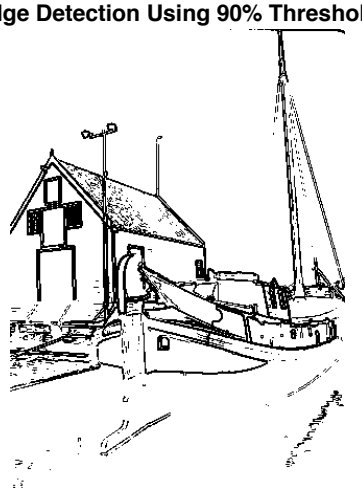
X Gradient Image Shows Vertical Edges and Y Gradient Image Shows Horizontal Edges

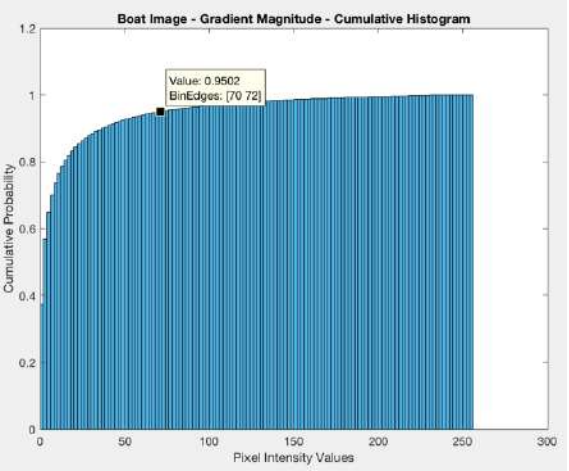
Gradient Magnitude - Normalized



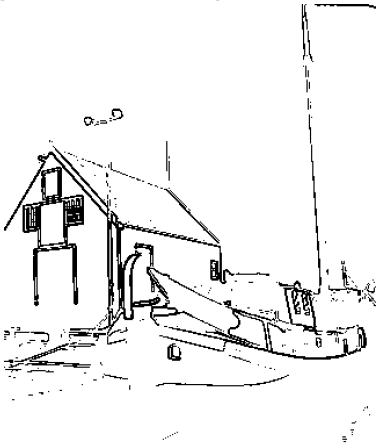


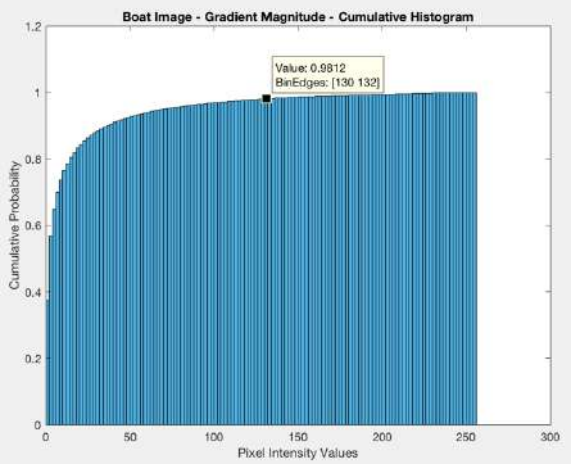
Edge Detection Using 90% Threshold





Edge Detection Using 95% Threshold



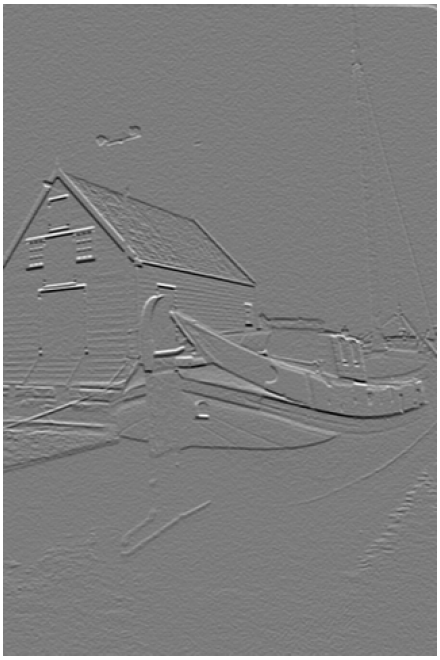


Edge Detection Using 98% Threshold

Sobel Edge Detection (Boat_noisy.raw image)

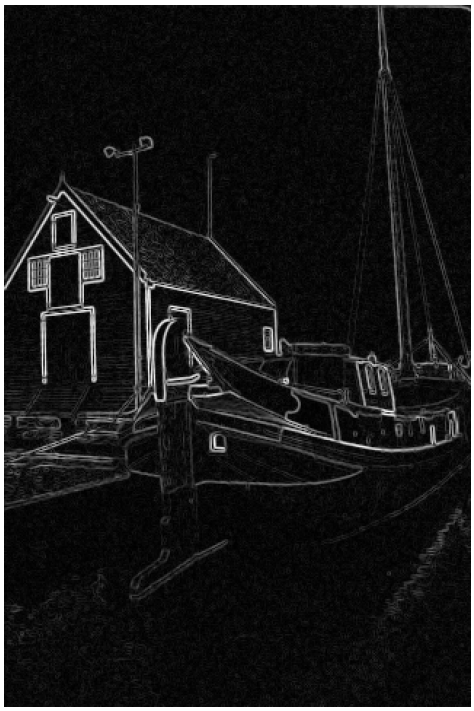
X Gradient - Normalized

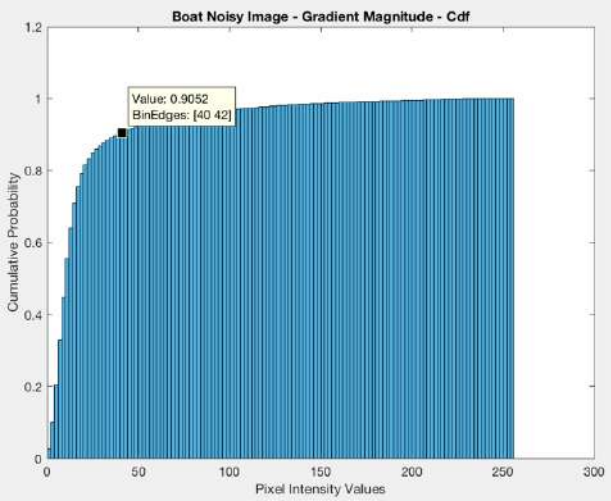
Y Gradient - Normalized



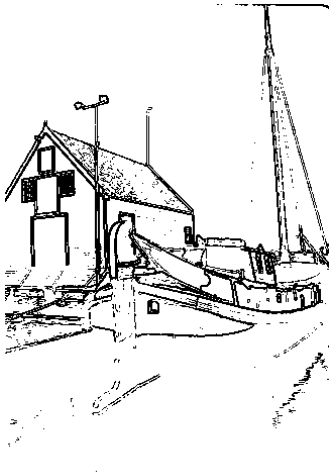
X Gradient Image Shows Vertical Edges and Y Gradient Image Shows Horizontal Edges

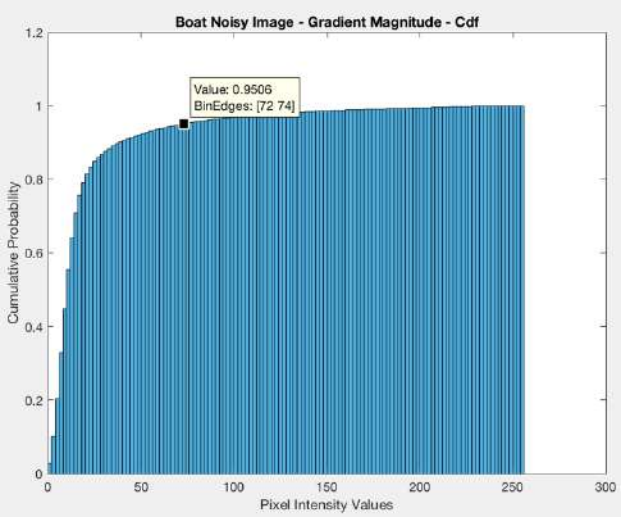
Gradient Magnitude Normalized



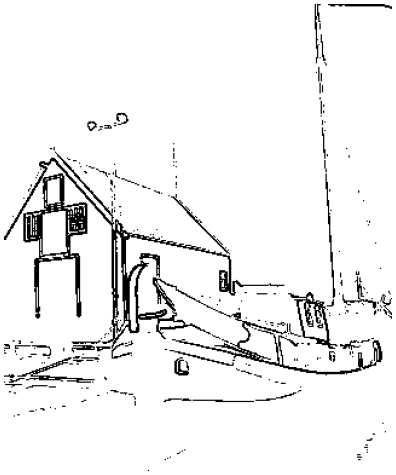


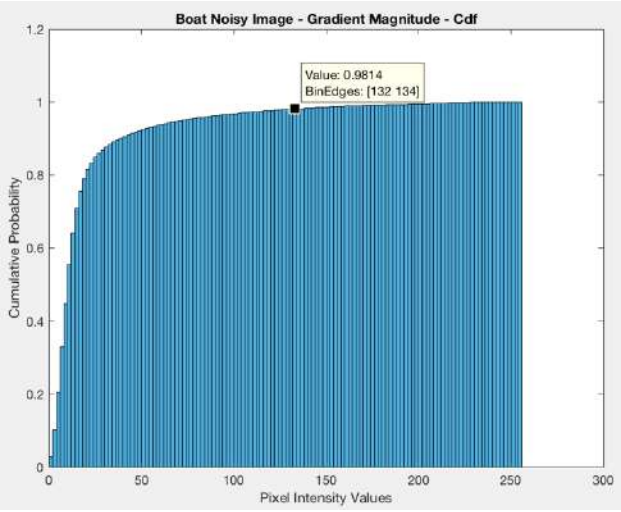
Edge Detection using 90%



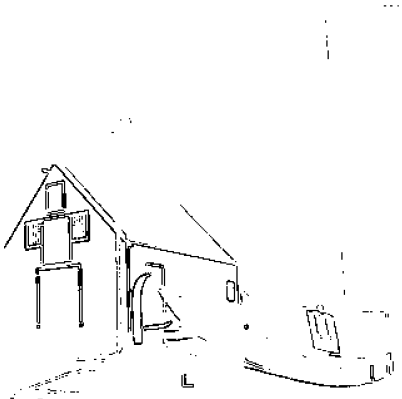


Edge Detection using 95%



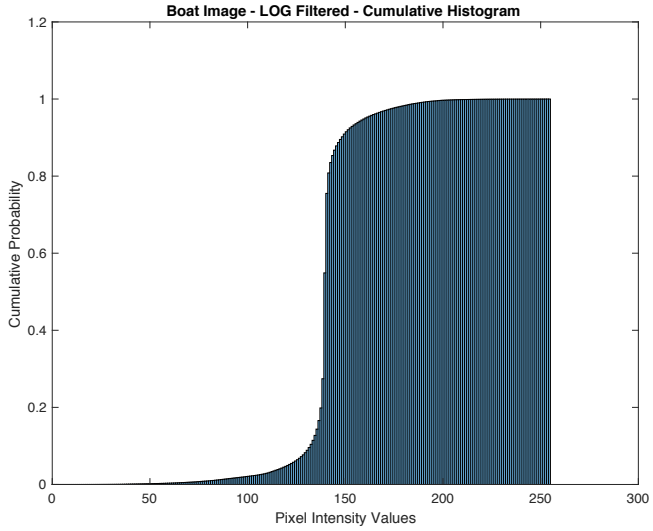


Edge Detection using 98%



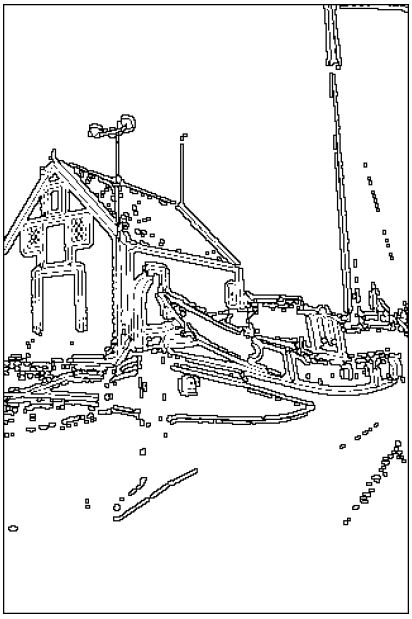
Zero Crossing Edge Detection (Boat.raw image)

LOG Gradient - Normalized



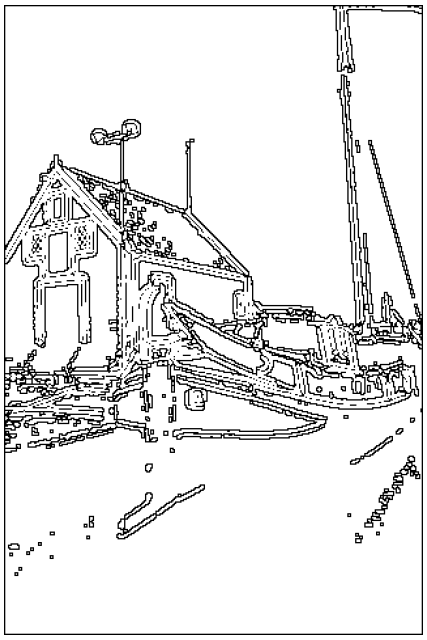
Thresholded at 5% and 95%

Edge Detection at 5% and 95%



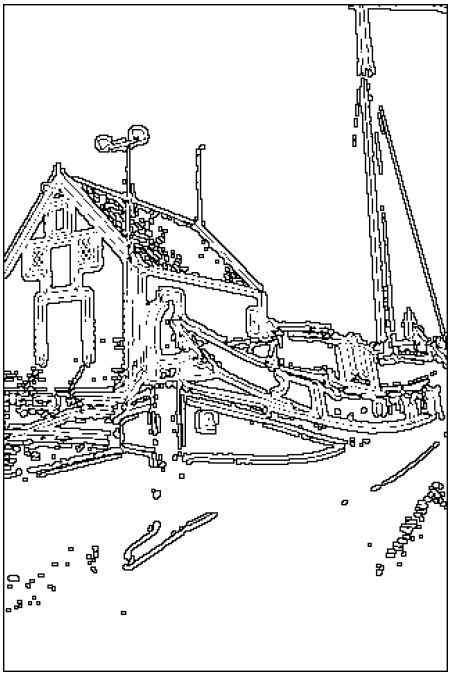
Thresholded at 6% and 94%

Edge Detection at 6% and 94%



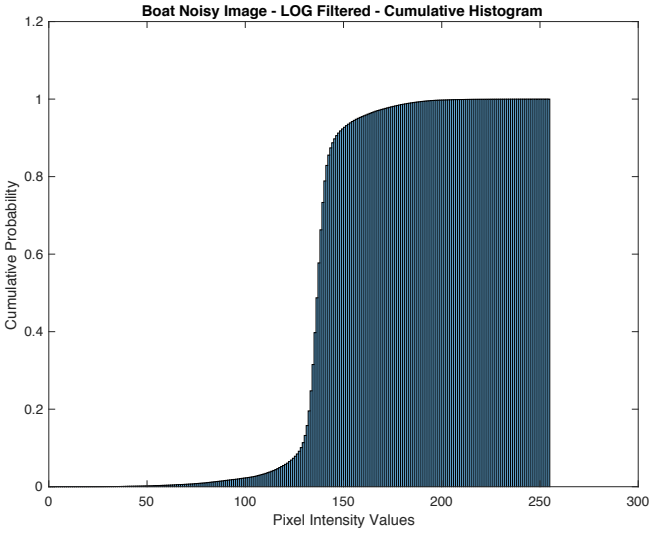
Thresholded at 7% and 93%

Edge Detection at 7% and 93%



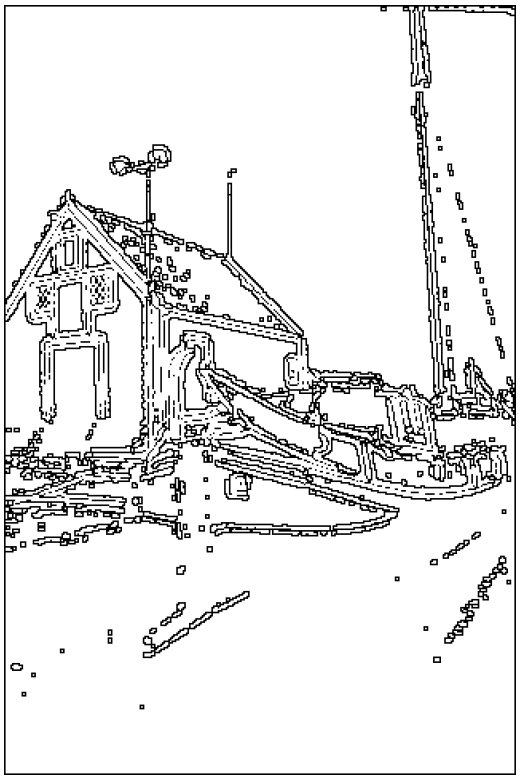
Zero Crossing Edge Detection (Boat_noisy.raw image)

LOG Gradient - Normalized



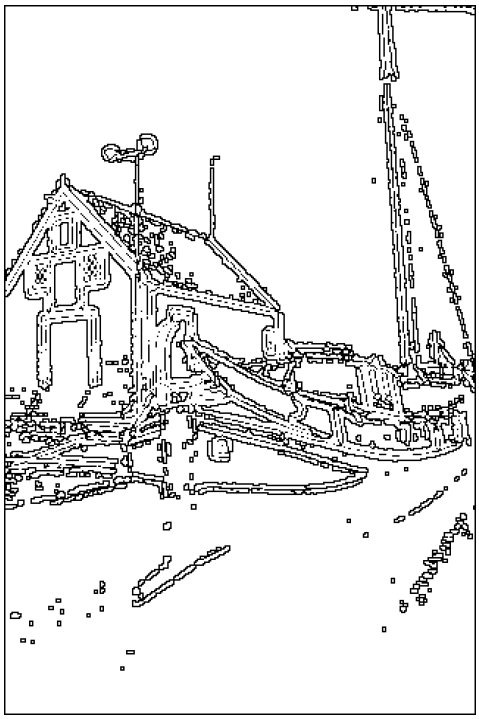
Thresholded at 5% and 95%

Edge Detection at 5% and 95%



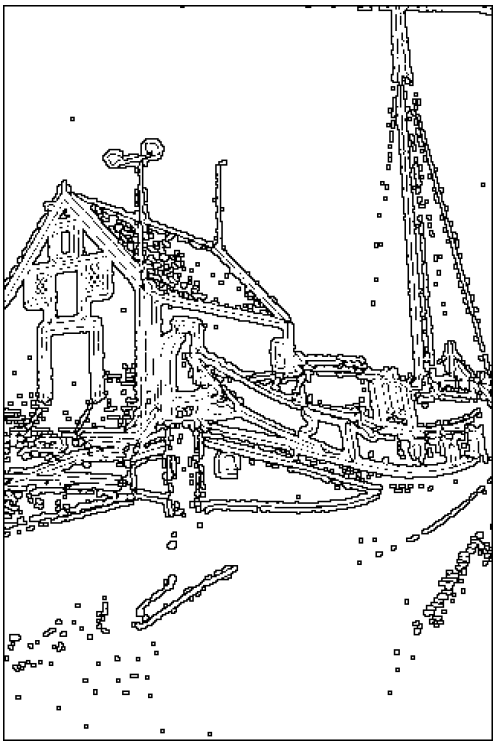
Thresholded at 6% and 94%

Edge Detection at 6% and 94%



Thresholded at 7% and 93%

Edge Detection at 7% and 93%



➢ DISCUSSION:

Theoretical Comparison between Sobel and Zero Crossing Edge Detection:

- The main difference between Sobel and Zero Crossing Edge Detection is the derivatives
- Sobel detects edge after single derivative and decides edges based on threshold
- Zero Crossing Edge Detection detect edge after double derivative, and hence no threshold is needed. Edges are decided based on zero crossing
- Threshold is the main difference between both the methods, and hence Sobel is very sensitive to the threshold we select
- Also, Sobel Edge Detection is very sensitive to Noise. Whereas, LOG detection implements Gaussian Filter before taking derivatives. Hence most of the noise is removed by the Gaussian Filter

Experimental Results Comparison between Sobel and Zero Crossing Edge Detection:

- Sobel Edge detection was done using different thresholds at 90%, 95% and 98%
- On performing edge detection on both noise free and noisy images, 90% threshold seems to give the best output. The edges are sharp and clear
- As I kept on increasing the thresholds, edges were disappearing. As I reduced the thresholds, the image was becoming noisy
- Noisy and Noise free image have similar edge detected

- For LOG Edge detection, I chose and experimented with 5%, 6% and 7% thresholds at both the tails of CDF
- As I kept on increasing the thresholds, the number of discontinuous points on the image were increasing. To avoid that I had to choose low thresholds such that not much of the information is lost as well as high threshold such that they do not have many discontinuous points
- For Noise Free image, 6% and 94% thresholds give the best output
- For Noisy image, 5% and 95% thresholds give the best output

- Comparing Sobel and Zero Crossing Edge Detection, the later has the best performance on both noise and noise free images
- For example, notice the top left and right part of the house door. Sobel Edge detection gives shaded black region. But LOG edge detection gives proper vertical and boxed lines
- Also notice the boat, double super clear edges are seen in the LOG Zero Crossing edge detection than Sobel Edge Detection

- On the whole, LOG seems to be the advanced and better edge detection algorithm when compared to Sobel Edge Detection. The main advantage is because of extra Gaussian Filter and double derivative so that zero crossing can be done

E) STRUCTURED EDGE:

➢ ABSTRACT AND MOTIVATION:

In the previous part of the question, Sobel Edge Detection and Laplacian of Gaussian Edge Detection were used to detect edges. Although these techniques are considered to be good in terms of performances when compared to previous days Edge Detection methodologies, time has come for improvement. This part of the question is one where I have experimented an advanced Edge Detection methodology called Structured Edge Method, a sophisticated and improvised technique for Edge Detection. The Theory part of the question explains the algorithm of Structured Edge (with a flowchart) and Random Forest Classifiers. In the experiment section of the question, I have used the downloaded Edge detection method using Structured Edge codes to perform analysis.

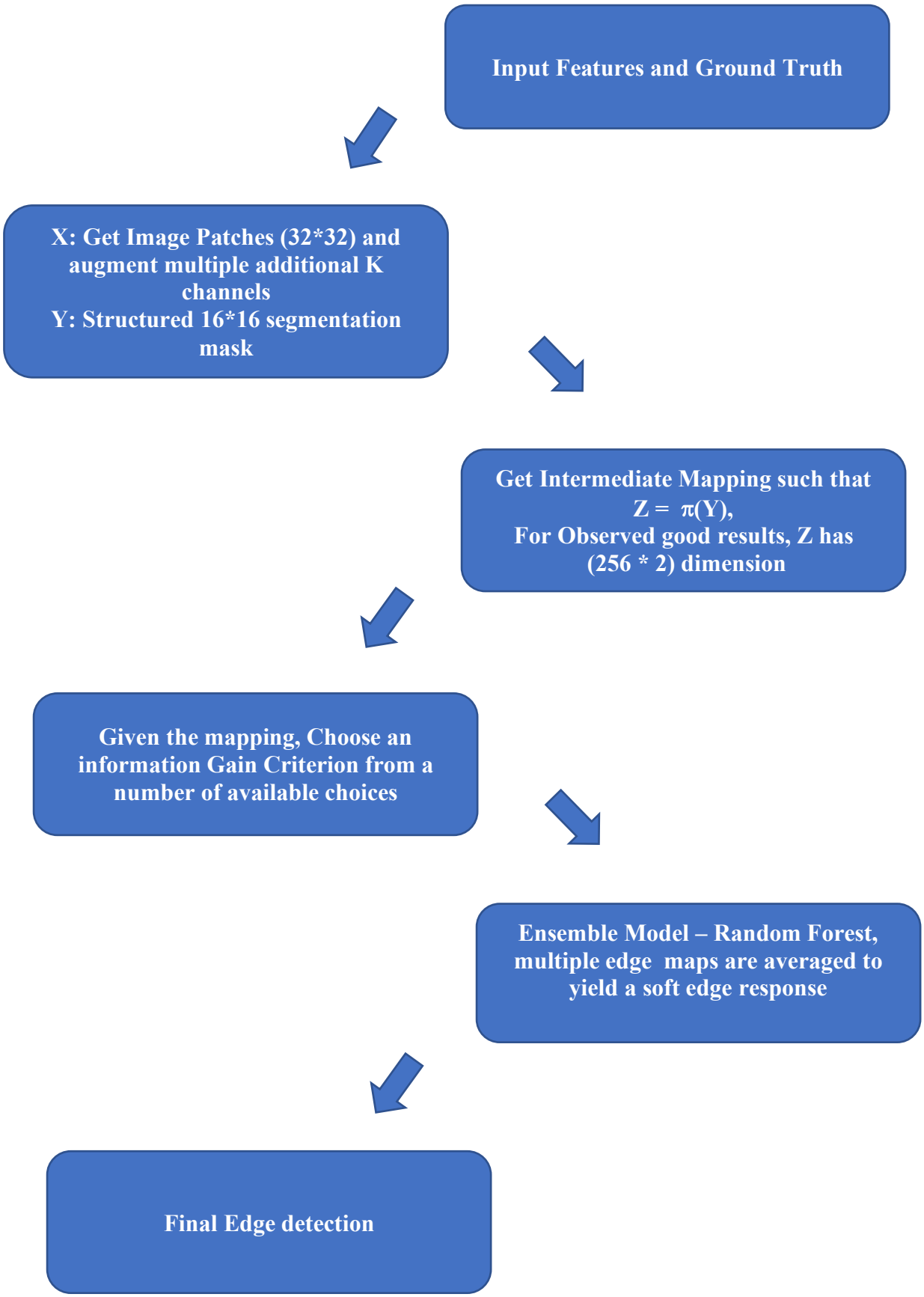
➢ APPROACH AND PROCEDURES:

1. Structured Edge Detection Algorithm:

Structured Forests for Fast Edge Detection was a research work originally published by Microsoft Research. The success of this algorithm was in the fact that they proposed predictions of local edge masks in a structured learning framework on the basis of Random Decision Forests. I believe the main advantage of this kind of edge detection is that

- In real time the performance is faster than any state-of-the-art approaches
- Contour detection is very less noisy and efficient

- The below flow chart explains Structured Edge algorithm in a systematic way. Important step to notice is that the algorithm extends Random Decision Forests to general output spaces.
- For calculating Information Gain, the algorithm relies on measuring similarity over Y, where Y are the structured output spaces.
- Sometimes the similarity over Y is not well defined and hence mapping of Y to Z space is done, so that distance is easily measured for finding similarity
- Thus, with this understanding with two stages are followed basically. They are:
    a) Y to Z intermediate mapping to get a better estimate of similarity
    b) Z to C straight forward mapping where the labels are predicted
- The Z to C mapping is done by Random Forests Decision Tree Classifier which combines the output of many Decision trees
- This sort of ensemble training is always proved to be the best in any Data Science field, not limited to Image Processing techniques
- Following the flow chart, there is a detailed explanation on Decision Trees Implementation also.
- The performance of such a robust algorithm can be evaluated by various metrics like Precision, Recall and F Measure. When comparison with the other state of art Edge detection methodologies, they would always be better because of its efficiency and advantages. This is proved in the next part of the question.



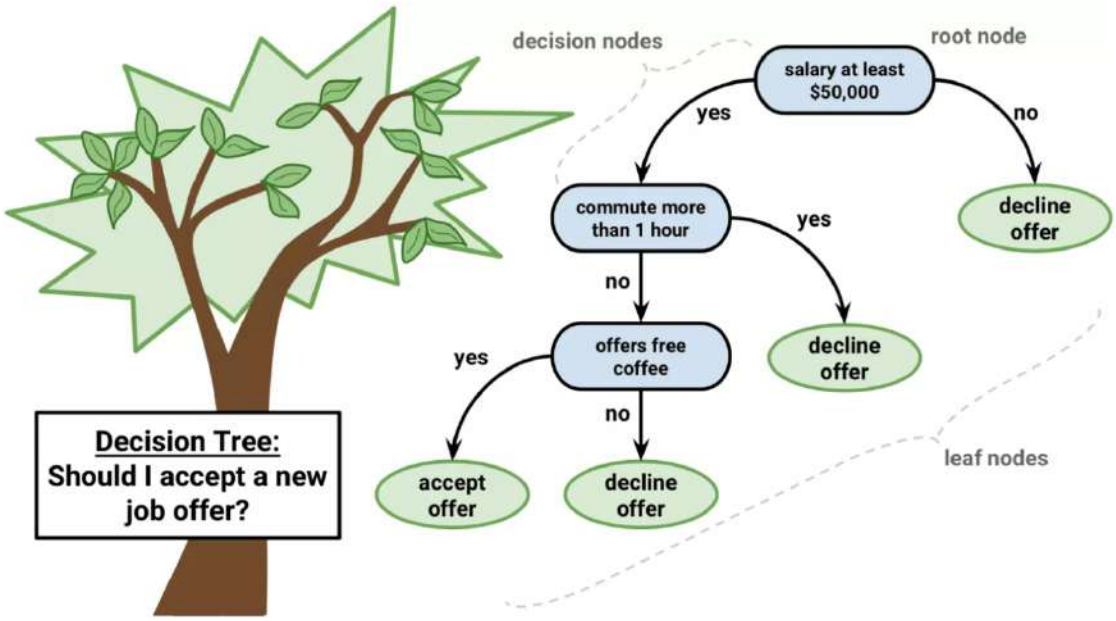
Flow Chart for Structured Edge Detection

2. Random Forest Classifier:

Random Forest Classifier is becoming to be widely used in all applications because of its various advantages and applications. Some of them are:

- It is very generic and can be used for both classification and regression
- Random Forest Classifier by itself can handle missing values
- Even when more decision trees are added to the classifier, there will be no problem of overfitting. People from various industries suffer with overfitting problem.
- Again, Random Forest Classifier can handle both Categorical and Continuous Numerical values.

As the name suggest, Random Forest Classifier builds a forest with many number of trees (i.e.) decision trees. Higher the number of decision trees, the accuracy is high. As mentioned above there is no problem of overfitting. First, let us understand the concept of decision trees using a simple example given below:
(Source: https://dataaspirant.com/2017/01/30/how-decision-tree-algorithm-works/)



The Algorithm of Decision Trees can be simply understood as below:

- The best attribute of the dataset as the root node for the tree to be built
- Training dataset is split into subsets, such that each subset contains data with the same value for an attribute
- For this step, best feature and best feature value has to be chosen and many Criteria are available. One such common criteria is Information gain
- Calculate Information Gain for every feature and for every possible feature value. Select the feature and a value such that information gain is maximum
- Repeat all the steps above on every subset until you get a leaf node in all the subsets

Algorithm of Random Forest based on Decision Trees:

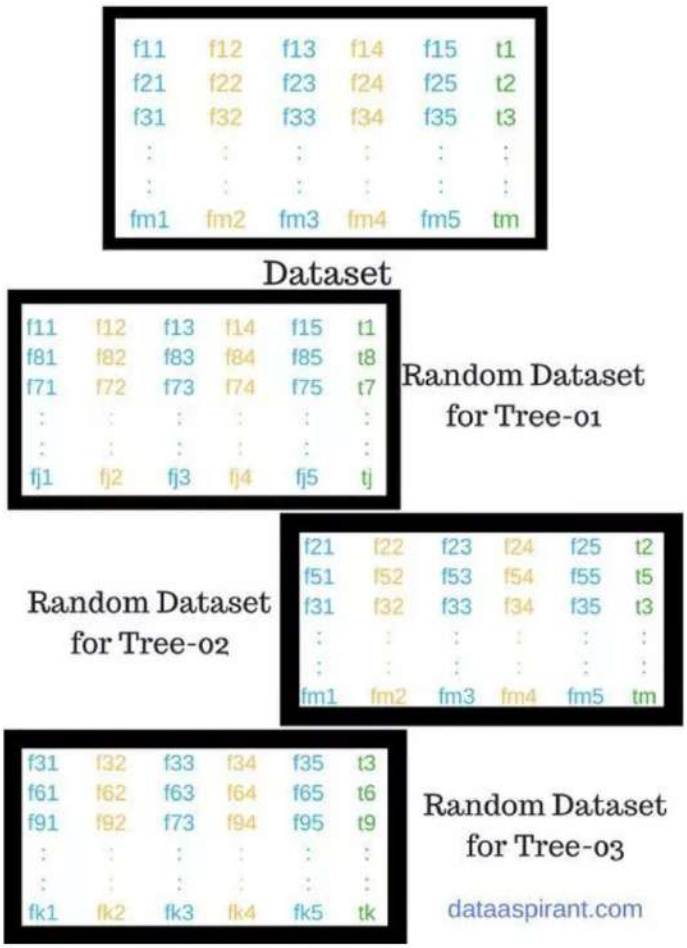
Training:

- Randomly select "K" features from total of "M" features such that K << M
- From the selected K features, calculate the node "d" using the best split point criteria
- Split the parent node into child nodes based on best split point
- The above steps are repeated until there is only one node is left in all child nodes
- The above steps are repeated n – number of times such that a forest is built with many number of decision trees

Testing:

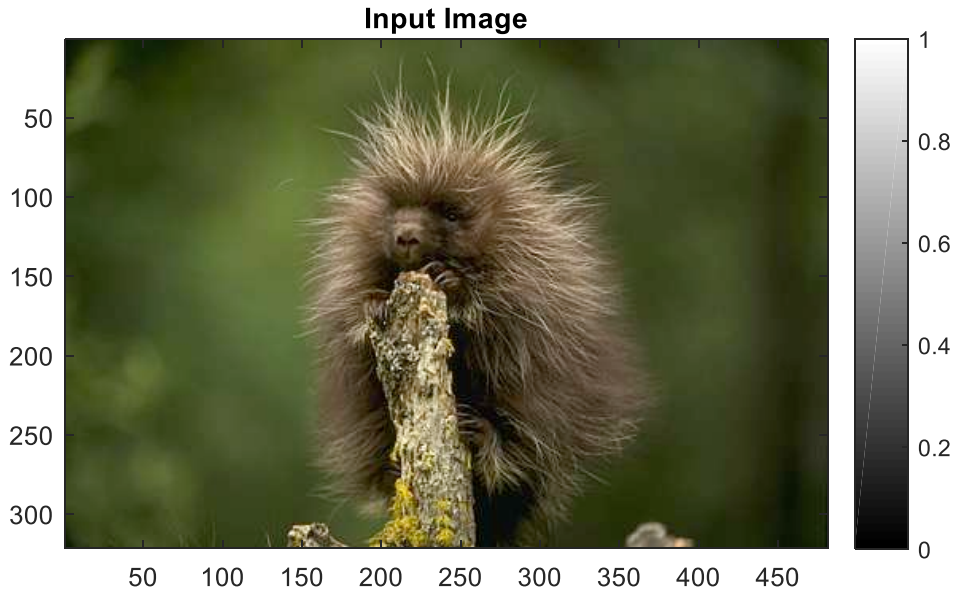
- Get the test feature and use the decision rules developed for every decision tree to predict the outcome
- Any criteria can be used to combine the predictions from every decision tree to get a final output
- For example, can calculate votes for each predicted target and consider highly predicted target as the final prediction
- Also, can find the means of the outputs if they are continuous and consider them as the final prediction

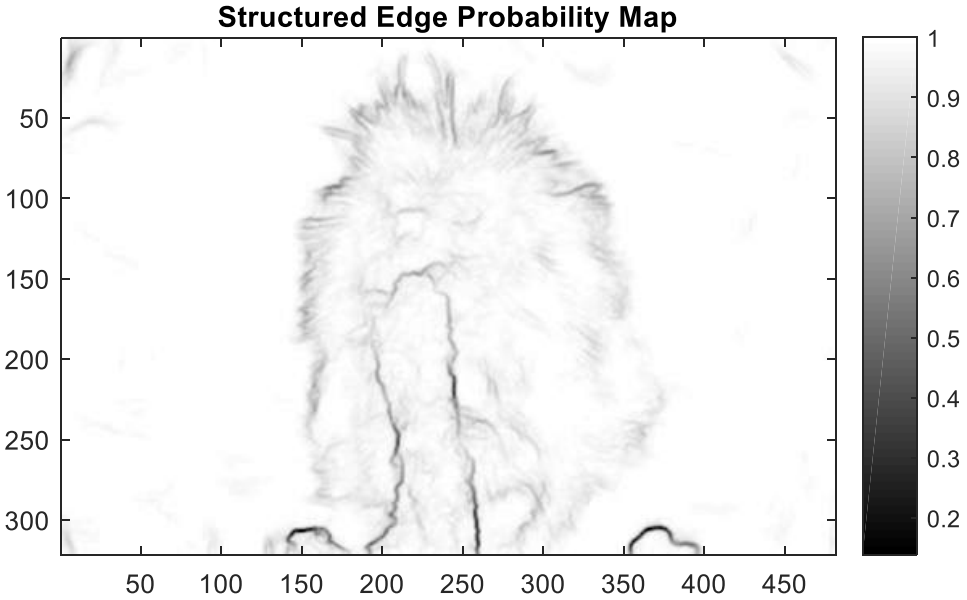
The figure shown below is for better understanding of how Random Forests Decision Trees work: (Source: http://dataaspirant.com/2017/05/22/random-forest-algorithm-machine-learing/)



➢ EXPERIMENTAL RESULTS:

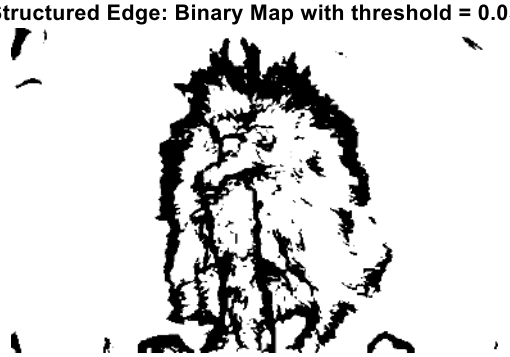
ANIMAL IMAGE – STRUCTURED EDGE WITH BAD PARAMETERS





Chosen Parameters:
Multiscale = 0
Sharpen = 2
nTreesEval = 4
nThreads = 4
nms = 0

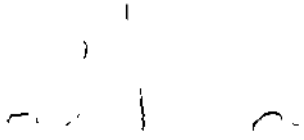
Structured Edge: Binary Map with threshold = 0.05



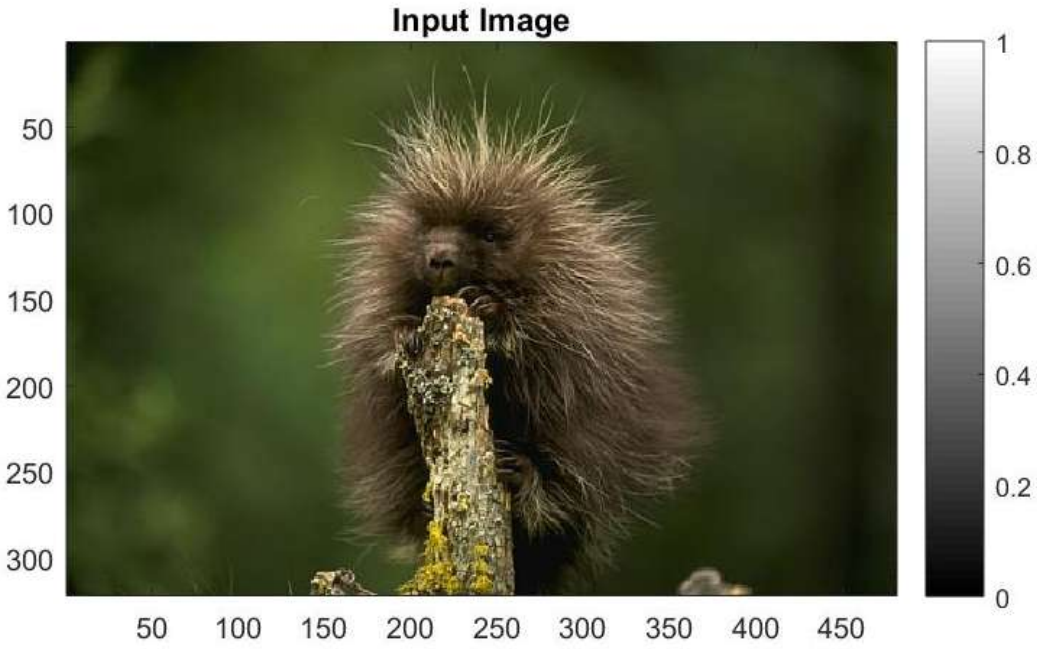
Structured Edge: Binary Map with threshold = 0.1

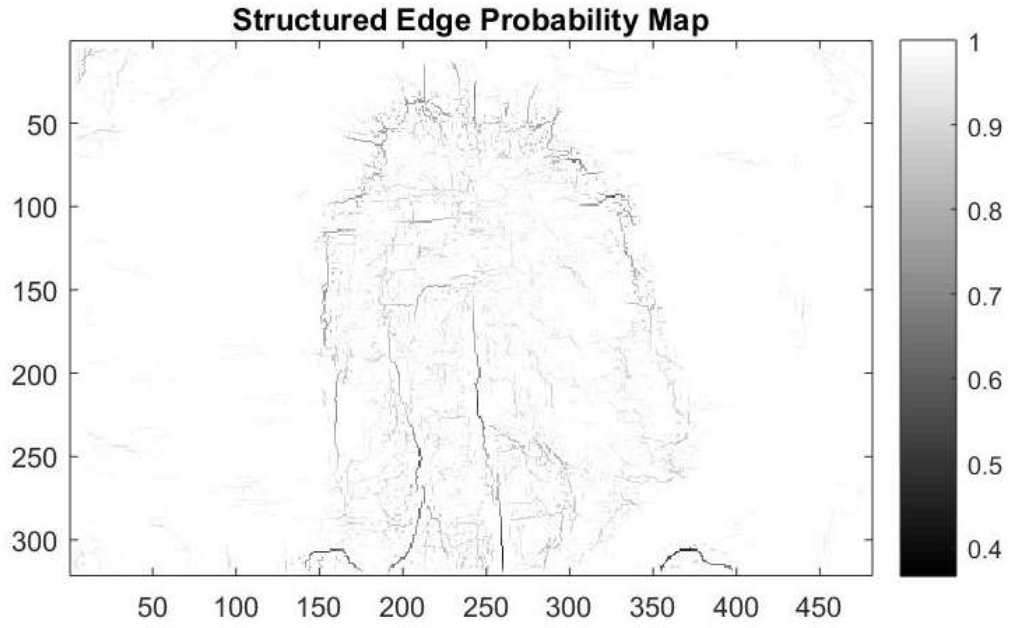
Structured Edge: Binary Map with threshold = 0.2

Structured Edge: Binary Map with threshold = 0.5



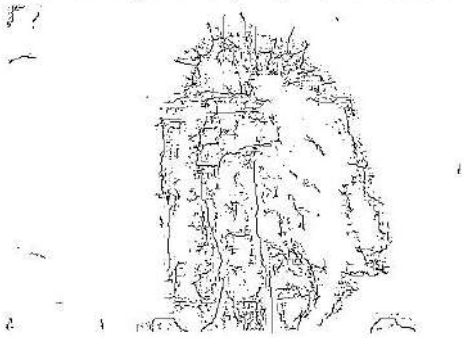
ANIMAL IMAGE – STRUCTURED EDGE WITH BEST PARAMETERS



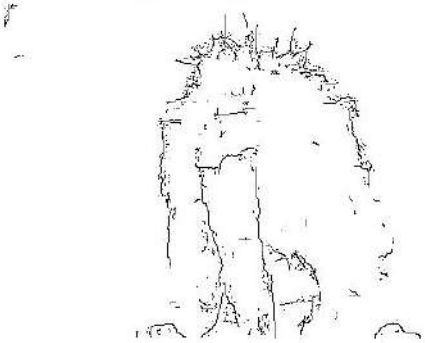


Chosen Parameters:
Multiscale = 1
Sharpen = 0
nTreesEval = 1
nThreads = 4
nms = 1

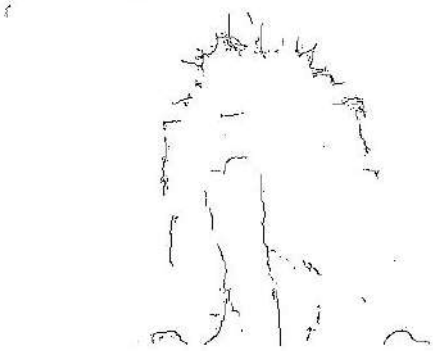
Structured Edge: Binary Map with threshold = 0.05



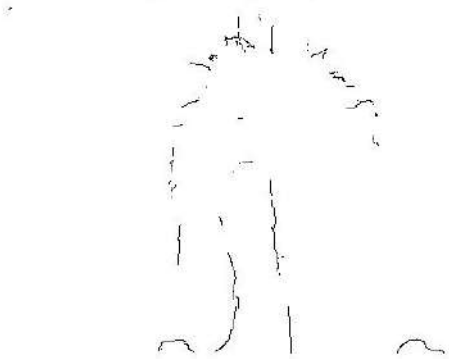
Structured Edge: Binary Map with threshold = 0.1



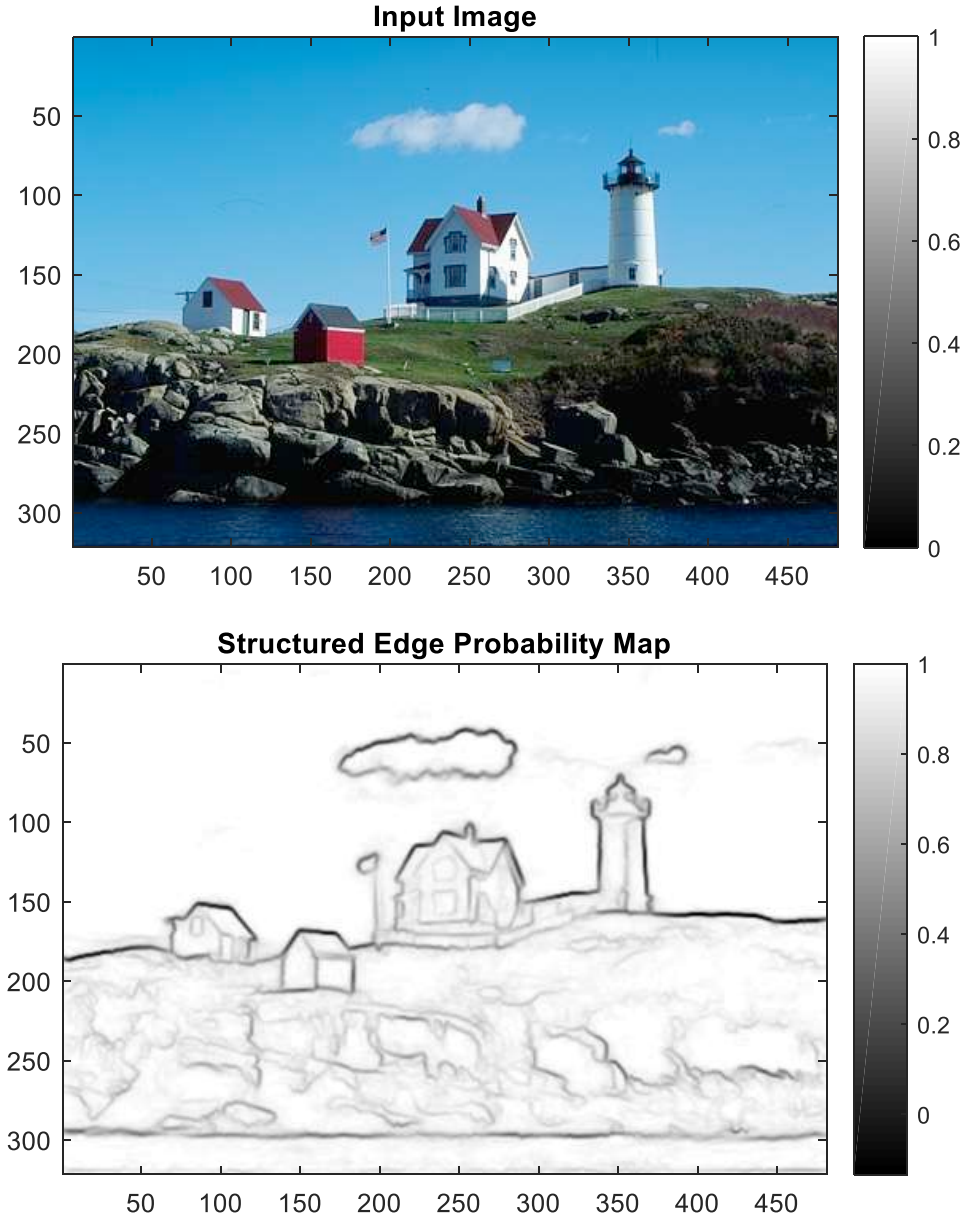
Structured Edge: Binary Map with threshold = 0.15



Structured Edge: Binary Map with threshold = 0.2



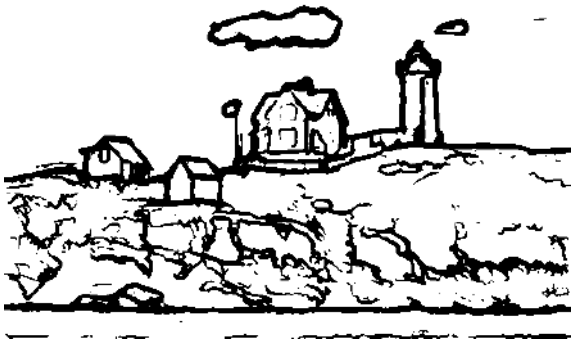
HOUSE IMAGE – STRUCTURED EDGE WITH BAD PARAMETERS



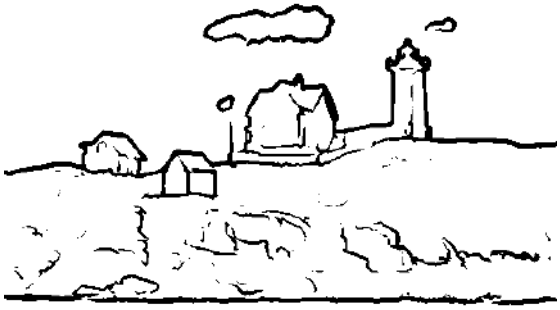
Chosen Parameters:
Multiscale = 0
Sharpen = 2
nTreesEval = 4
nThreads = 4
nms = 0

Structured Edge: Binary Map with threshold = 0.05

Structured Edge: Binary Map with threshold = 0.1

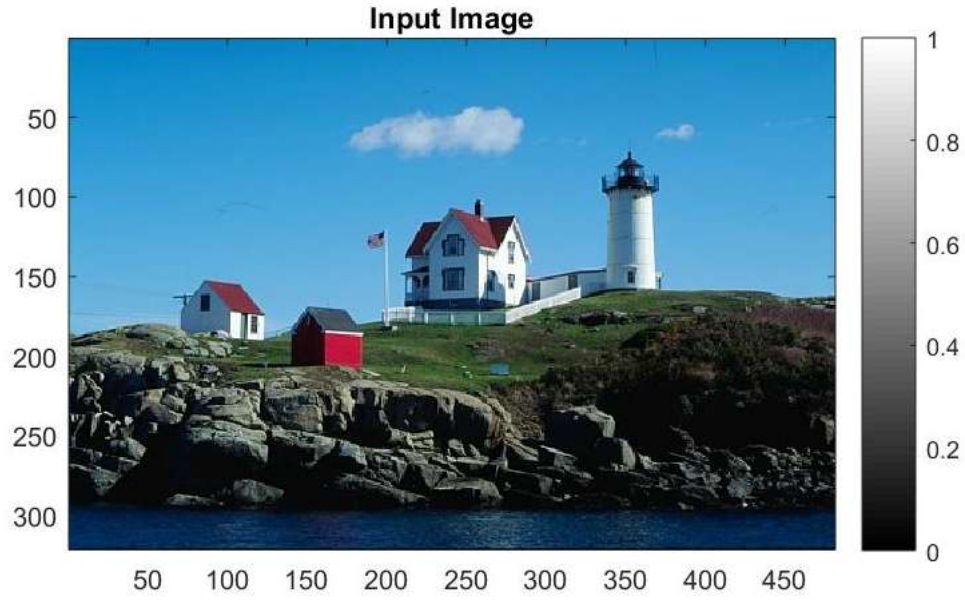


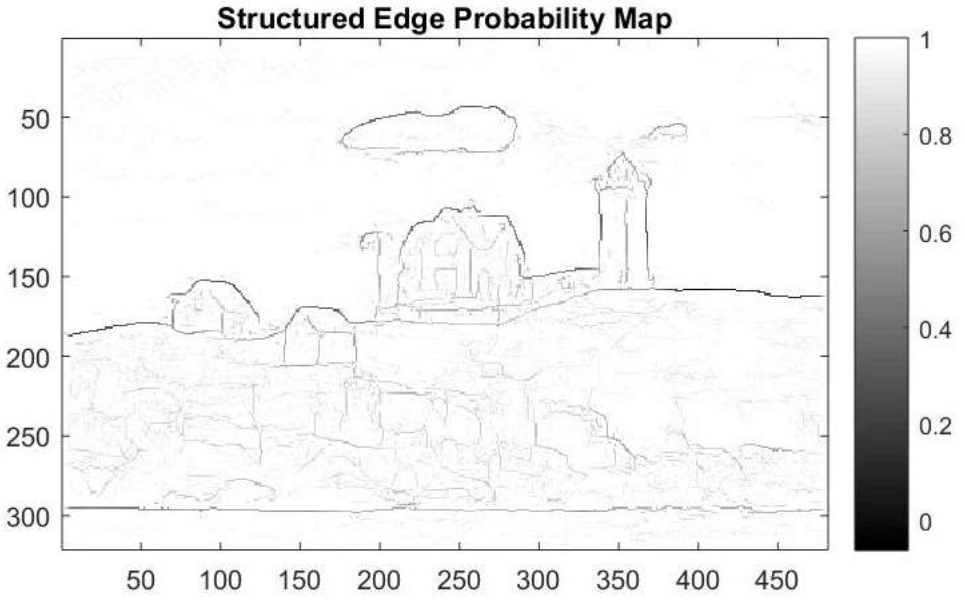
Structured Edge: Binary Map with threshold = 0.2



Structured Edge: Binary Map with threshold = 0.5

HOUSE IMAGE – STRUCTURED EDGE WITH BEST PARAMETERS

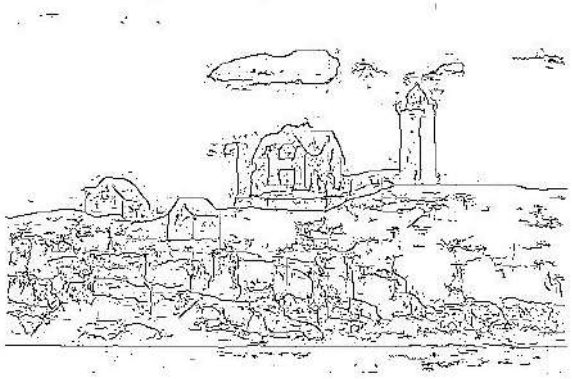




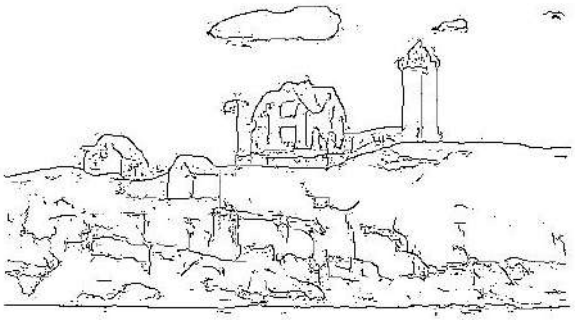
Chosen Parameters:

Multiscale = 1
Sharpen = 0
nTreesEval = 1
nThreads = 4
nms = 1

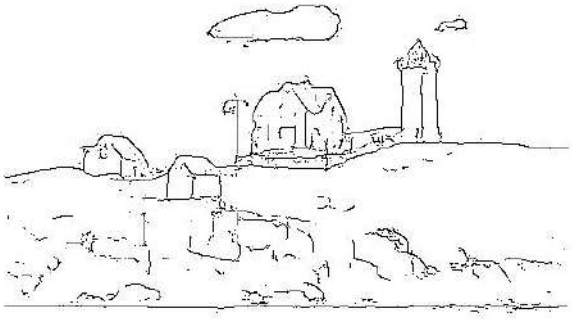
Structured Edge: Binary Map with threshold = 0.05



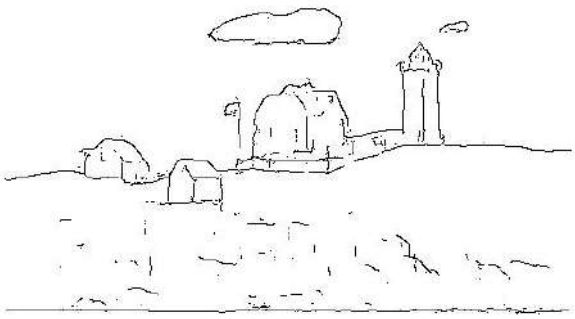
Structured Edge: Binary Map with threshold = 0.1



Structured Edge: Binary Map with threshold = 0.15

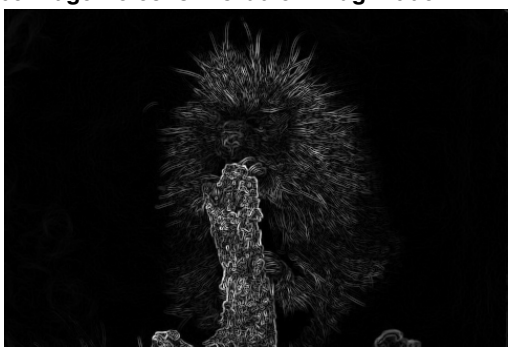


Structured Edge: Binary Map with threshold = 0.2

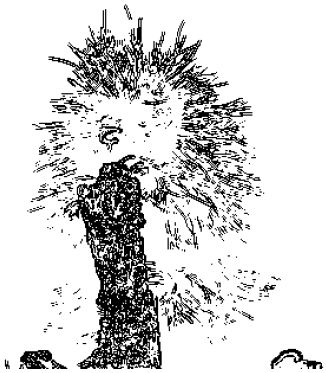


SOBEL EDGE DETECTION OF ANIMAL AND HOUSE IMAGES

Sobel Edge Detection- Gradient Magnitude - Animal



Sobel Edge Detection - Edges at 90% Threshold

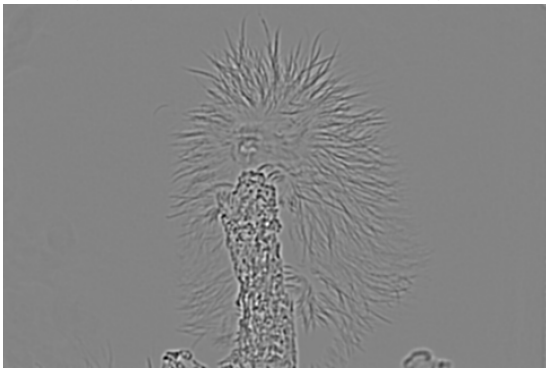


Sobel Edge Detection- Gradient Magnitude - House

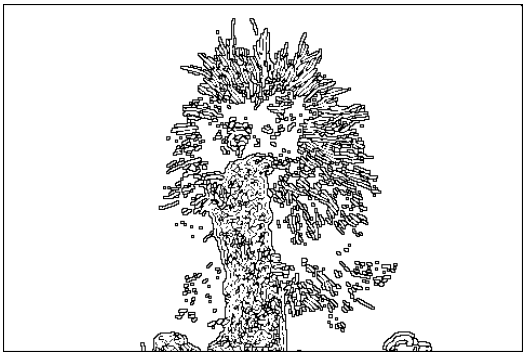
Sobel Edge Detection - Edges at 90% Threshold

ZERO CROSSING LOG EDGE DETECTION OF ANIMAL AND HOUSE IMAGES

Zero Crossing Edge Detection- Gradient Magnitude - Animal

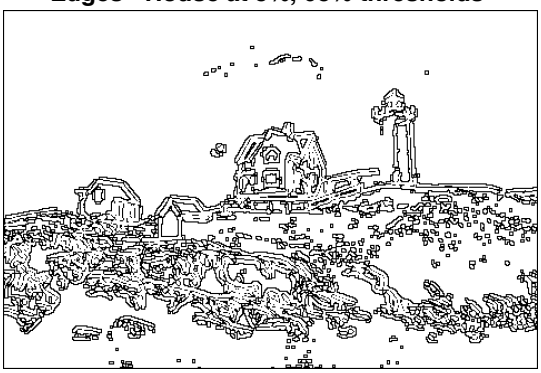


Edges - Abimal at 5%, 95% thresholds



Zero Crossing Edge Detection- Gradient Magnitude - House

Edges - House at 5%, 95% thresholds



➢ DISCUSSION:

- Thus, this part of the question, helped us understand Structured Edge Detection, which is now one of the most frequently used algorithm in real world
- An overall outline is that training structured labels with random forest is difficult and had two main challenges:
    - Many a times structured output spaces are high dimensional and complex
    - Information gain over the structured labels may not be well defined and training decision trees would be difficult

  These problems were overcome by reducing the structured output spaces to a different reduced dimension. And, this was done by the Intermediate Mapping that was described above.
- From above results, Structured edge proves to be more robust, less Noisy and has very less computation time.
- I also tried changing various parameters of Structured Edge to understand the online code and compare results
  (Source: https://arxiv.org/pdf/1406.5549.pdf)
    - This paper is also from Microsoft, with experimentation on various parameters of the model
        - Multiscale: Prediction is done at a multi scale level – two enhancements are done (Value = 1 gives good output)
        - Sharpening: Sharpening of the image to improve accuracy (Value = 1 to improved accuracy and Value = 0 for high speed)
        - nTreesEval: Number of Decision Trees (Value = 1 for high speed)
        - nThreads: Maximum number of threads for evaluation
        - nms: Non-Maximal Suppression to be done or not (Value = 1 for improved accuracy definitely)
        - Thresholds: Tried on values = 0.05, 0.1, 0.15, 0.2 and 0.5 for above parameters
- The above results have been shown based on Best and Bad parameters. Best parameters were chosen keeping accuracy and computational time in mind. There can always be a trade-off between the two performance measures.
- Thus, for the best parameters,
    - Animal Image shows best output when Threshold = 0.1
    - House Image shows best output when Threshold = 0.1
- On comparison with Sobel and Zero Crossing Edge Detection, as expected the Structured Edge Output is less noisy and clean. The edges are very sharp in Structured Edge Output whereas, the other two have edges that are either thick or looks False-positive (Which need not be necessarily an edge).
- Because of the texture in House and Animal Images, High Frequency components by Sobel and LOG are misunderstood as Edges, while they are not.

F) PERFORMANCE EVALUATION:

➢ ABSTRACT AND MOTIVATION:

In the first part of the question, I had shown two state-of-art Edge detection methodologies like Sobel Edge Detection and Zero Crossing LOG based Edge Detection. In the second part of the question, I showed an advanced and robust method of Edge Detection called Structured Edge Detection based on Machine Learning Concepts instead of designing our own algorithm. I claimed that later is has good performance on efficiency and noiseless results. As it always goes without saying, every claim is to be proved for a better understanding. Hence, in this part of the question, I have done some commonly used Performance Evaluation Techniques to get various values for certain evaluation metrics. Detailed theoretical explanation of the evaluation metrics is given below. Also, in the experimental results section I have shown tables of comparisons between Structured Edge and previous techniques of Edge Detection.

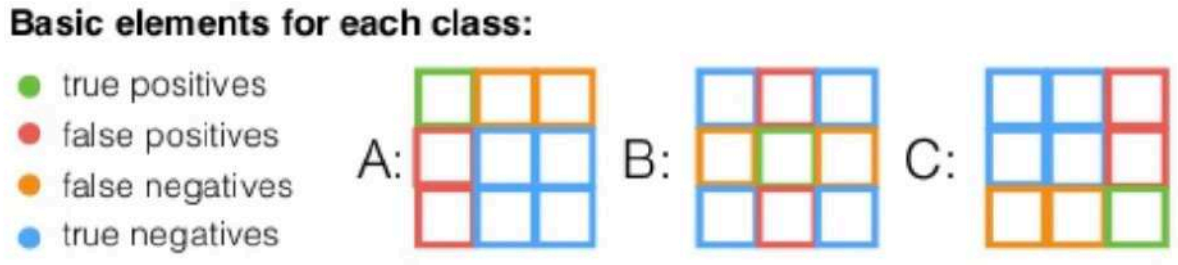
➢ APPROACH AND PROCEDURES:

The concept of Performance Metrics explained below is based on "Confusion Matrix". This focuses on how good the predictive capability of a model developed is. For better understanding, this confusion matric does not have anything to do with the computational speed or scalability or etc. of a model. An example for better understanding is shown below:

		prediction		
		A	B	C
ground truth	A	1022	13	144
	B	300	542	24
	C	12	55	132

Let's take the example above,

- The above shown square boxes is an example of "Confusion Matrix" of a predictive model
- The rows indicate the ground truth or the actual classes. The columns indicate the predicted classes
- Thus, as it goes without saying, "A", "B" and "C" are the class labels
- This values inside the scores are the count of predictions
- For example, 1022 is the number of features/instances that are predicted as class "A" and actually also belongs to "A"
- The value 144 is the number of features/instances that are predicted as class "C", but they actually belong to class "A"
- Interpretation of this Confusion Matrix is very important to proceed further to defined evaluation metrics like Prediction, Recall and F-Measure



- This, from the confusion matrix, these True Positives, False Positives, False Negatives and True Negatives are formed
- From, these values, various performance Metrics can be evaluated. This is shown in the below figure.

For each class (or for two class problems):

Precision / PPV	tp / (tp + fp)	■ / (■ + ■)
Recall / Sensitivity	tp / (tp + fn)	■ / (■ + ■)
Specificity	tn / (tn + fp)	■ / (■ + ■)
Accuracy	(tp+tn) / (tp + fp + fn + tn)	(■ + ■)/(■ + ■ + ■ + ■)
F1-score	2*prec*sens / (prec+sens)	

(Source: https://www.slideshare.net/ThomasPloetz/bridging-the-gap-machine-learning-for-ubiquitous-computing-evaluation)

- With the above understanding in mind, let's expand this to Edge Detection to perform Evaluation Metrics on various Edge Detection Metrics.
- For this specific application, only two classes are there. It is a binary classification as either Edge or Not an Edge.
- Confusion Matrix will look like below:

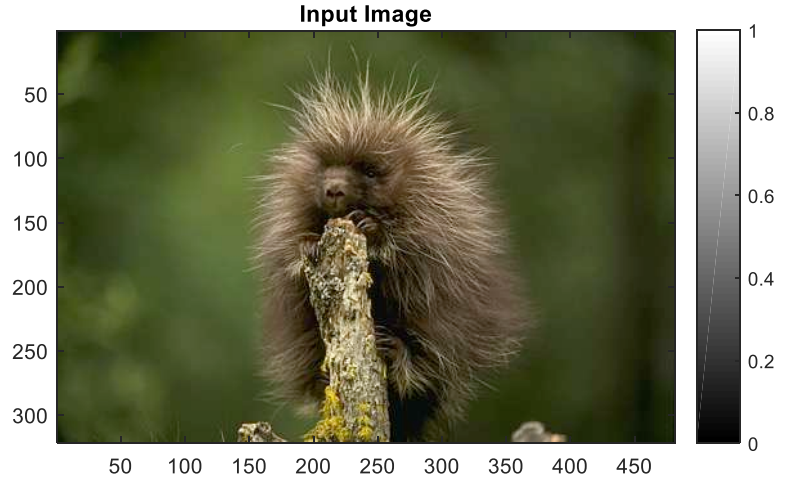
Confusion Matrix for Edge Detection	**Predictions**		
Ground Truth		**Edge**	**Not an Edge**
	Edge	True Positive	False Negative
	Not an Edge	False Positive	True Negative

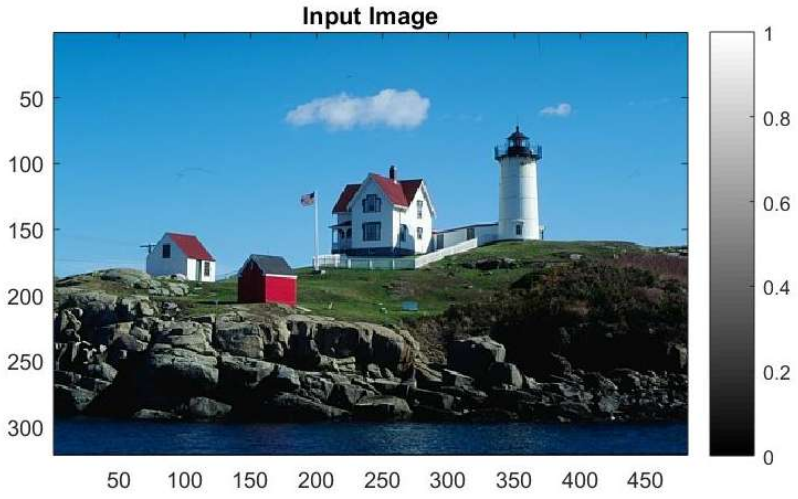
The explanation of the above table is given below: (Source: Question)

- True positive: Edge pixels in the edge map coincide with edge pixels in the ground truth. These are edge pixels the algorithm successfully identifies.
- True negative: Non-edge pixels in the edge map coincide with non-edge pixels in the ground truth. These are non-edge pixels the algorithm successfully identifies.
- False positive: Edge pixels in the edge map correspond to the non-edge pixels in the ground truth. These are fake edge pixels the algorithm wrongly identifies.
- False negative: Non-edge pixels in the edge map correspond to the true edge pixels in the ground truth. These are edge pixels the algorithm misses.

The experimental results are shown below and there is a tabular column for Animal and House images given to us. Performance evaluation Metrics were done for Structured Edge Detection, Sobel Edge Detection and Zero Crossing based LOG Edge detection

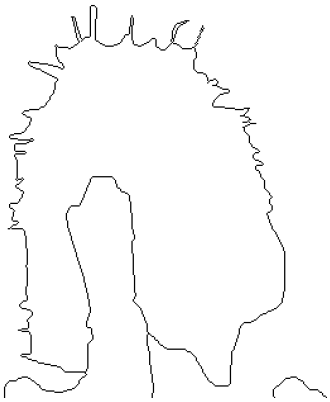
➢ EXPERIMENTAL RESULTS:



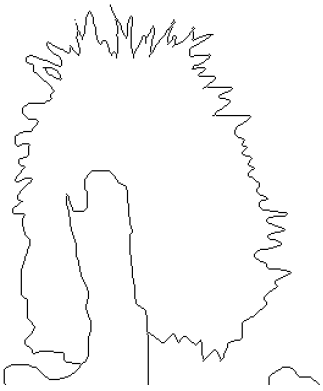


GIVEN GROUND TRUTH IMAGES OF ANIMAL

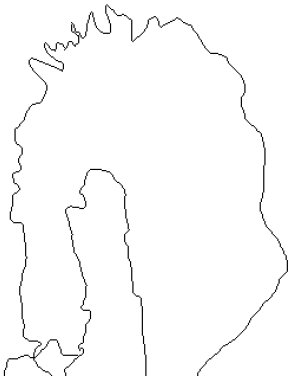
Animal - Ground Truth Image - 1



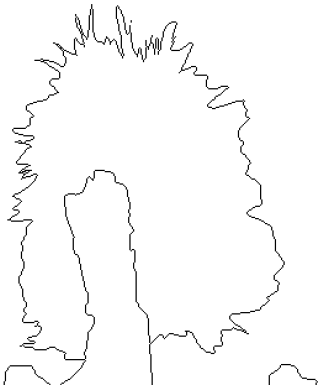
Animal - Ground Truth Image - 2



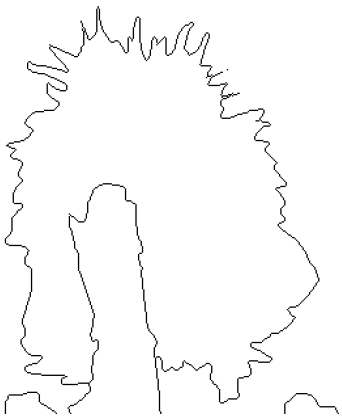
Animal - Ground Truth Image - 3



Animal - Ground Truth Image - 4

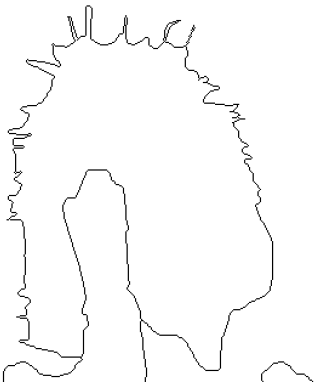


Animal - Ground Truth Image - 5

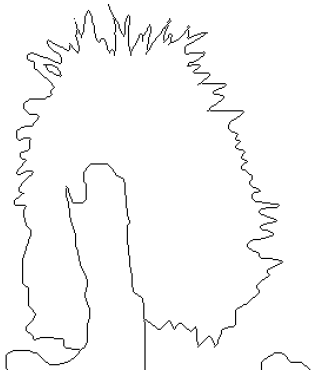


GIVEN GROUND TRUTH IMAGES OF HOUSE

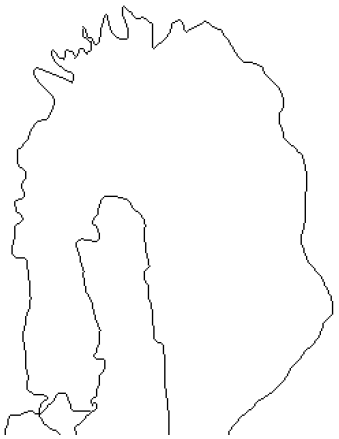
House - Ground Truth Image - 1



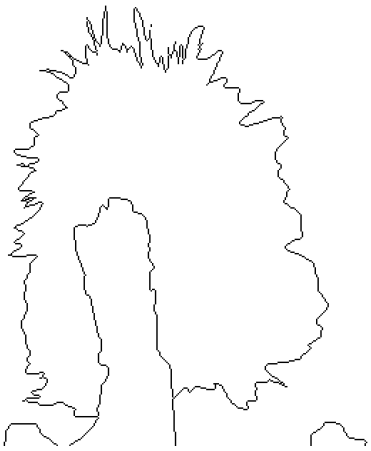
House - Ground Truth Image - 2



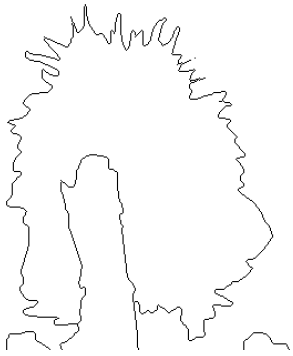
House - Ground Truth Image - 3



House - Ground Truth Image - 4



House - Ground Truth Image - 5



PERFORMANCE EVALUATION METRICS FOR GIVEN ANIMAL IMAGE

Structured Edge – Animal Image			
	Recall	**Precision**	**Max F Measure, Threshold**
Ground Truth 1	0.6276	0.6421	0.6348, 0.13
Ground Truth 2	0.6075	0.6337	0.6203, 0.12
Ground Truth 3	0.3971	0.5023	0.4436, 0.16
Ground Truth 4	0.6473	0.7395	0.6903, 0.13
Ground Truth 5	0.5551	0.6099	0.5812, 0.13
Threshold = 0.12 **Mean Recall = 0.6069** **Mean Precision = 0.7656** **F Measure = 0.6770**			

Sobel Edge Detection – Animal Image			
	Recall	**Precision**	**Max F Measure, Threshold**
Ground Truth 1	0.5449	0.1467	0.2312, 0.01
Ground Truth 2	0.5377	0.1690	0.2571, 0.01
Ground Truth 3	0.4094	0.09	0.1491, 0.01
Ground Truth 4	0.6287	0.1890	0.2907, 0.01
Ground Truth 5	0.5230	0.1512	0.2346, 0.01
Threshold = 0.01 **Mean Recall = 0.5352** **Mean Precision = 0.2428** **F Measure = 0.3340**			

Zero Crossing LOG Edge Detection – Animal Image			
	Recall	**Precision**	**Max F Measure, Threshold**
Ground Truth 1	0.9007	0.1614	0.2738, 0.01
Ground Truth 2	0.8532	0.1785	0.2952, 0.01
Ground Truth 3	0.6029	0.0920	0.1603, 0.01
Ground Truth 4	0.9542	0.1910	0.3182, 0.01
Ground Truth 5	0.8283	0.1594	0.2673, 0.01
Threshold = 0.54 **Mean Recall = 0.8418** **Mean Precision = 0.2674** **F Measure = 0.4059**			

PERFORMANCE EVALUATION METRICS FOR GIVEN HOUSE IMAGE

Structured Edge – House Image			
	Recall	**Precision**	**Max F Measure, Threshold**
Ground Truth 1	0.8757	0.7369	0.8003, 0.26
Ground Truth 2	0.6758	0.7779	0.7233, 0.23
Ground Truth 3	0.7101	0.8045	0.7544, 0.25
Ground Truth 4	0.7454	0.6989	0.7214, 0.22
Ground Truth 5	0.7534	0.5683	0.6479, 0.19
Threshold = 0.2 **Mean Recall = 0.7878** **Mean Precision = 0.8879** **F Measure = 0.8349**			

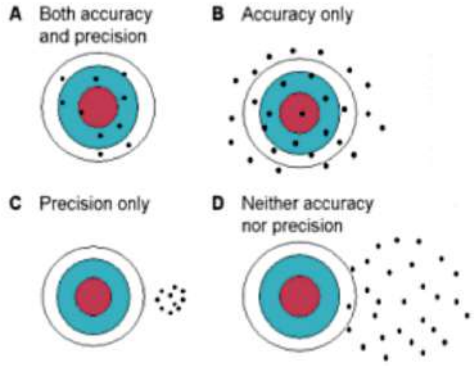
Sobel Edge Detection – House Image			
	Recall	**Precision**	**Max F Measure, Threshold**
Ground Truth 1	0.6153	0.1626	0.2572, 0.01
Ground Truth 2	0.5486	0.2356	0.3297, 0.01
Ground Truth 3	0.5425	0.2037	0.2962, 0.01
Ground Truth 4	0.6885	0.2528	0.3699, 0.02
Ground Truth 5	0.8616	0.3030	0.4484, 0.01
Threshold = 0.68 **Mean Recall = 0.6475** **Mean Precision = 0.8879** **F Measure = 0.4961**			

Zero Crossing LOG Edge Detection – House Image			
	Recall	**Precision**	**Max F Measure, Threshold**
Ground Truth 1	0.5892	0.1880	0.2851, 0.02
Ground Truth 2	0.4996	0.2592	0.3414, 0.01
Ground Truth 3	0.5372	0.2436	0.3352, 0.01
Ground Truth 4	0.6350	0.2817	0.3902, 0.01
Ground Truth 5	0.8045	0.3417	0.4797, 0.01
Threshold = 0.005 **Mean Recall = 0.6023** **Mean Precision = 0.4400** **F Measure = 0.5107**			

➢ DISCUSSION:

DISCUSSION 1:

- Thus, the final part of the question helped me understand various Performance Evaluation Metrics.
- In the approach section, I showed how a Confusion Matrix can be formed based on True Positive, False Positive, False Negative and True Negative
- I also showed Confusion Matrix can be better understood in terms of Precision, Recall and F-Measure Score
- Precision can be understood as the agreement among several dimensions. Higher the precision, lower is the difference between predicted and ground truth. An example difference between Precision and Accuracy is shown below



- Recall can be understood as the sensitivity. It can be viewed as the probability that the edge is detected by the algorithm correctly
- More than individual score, a combination of Recall and Precision can be used for better understanding. That is F-Measure Score.
- Higher the F-Measure Score, the better the model is. From the above results, F-Score is high for Animal and House images. This is expected, since in the previous part of the question, we saw the outputs directly. By visual inspection itself, the Structured Edge was performing better. Thus, my intuition was proved right from the Performance Evaluation Metrics.
- F-Measure for Sobel and Zero Crossing Edge detector is almost half of the F-Score of Structured Edge. This proves without any doubt that Structured Edge detection is supreme

DISCUSSION 2:

- F-Measure is highly Image dependent and can be proved from the above results. As I can see from the given images, Animal Image has very high frequency texture. While House Image has relatively less high frequency components. For even robust Structured Edge Detection:

  F_measure of House > F_measure of Animal (for Structured Edge)

- Thus F-Measure depends on the High frequency components present in the image and also based on method of Edge Detection
- Hence it is proved that, F measure of House is greater than Animal for the same model

DISCUSSION 3:

- To understand F-Measure, let's first understand that F-Measure is the harmonic mean of Precision and Recall. But, why Harmonic Mean?
    - Arithmetic Mean is best when it makes sense to sum up the values
    - Geometric Mean is best when it makes sense to multiply the values
    - Harmonic Mean is best when it comes to ratios and rates
- Since it is defined as the Harmonic Mean, if one value is high and the other value is low it badly affects the F-Measure Score Value. Since it is dealt with Ratios, both Precision and Recall values need to be the same to get a Maximum F-Measure
- This can be proved as follows:
    - Let's take P+R = C, a constant
    - P = R-C
    - F = 2 * P * R / (P+R)

To Maximize F:

- Differentiate of F, with respect to R, and equate to zero
- (2*2* (R-2) * C) / C = 0
- 4 * R = 2 * C
- R = C / 2,

For this to happen,

P = R

This Precision and Recall has to be same for F score to be maximum

PROBLEM 3
SALIENT POINT DESCRIPTORS AND IMAGE MATCHING

G) EXTRACTION AND DESCRIPTION OF SALIENT POINTS:

➢ ABSTRACT AND MOTIVATION:

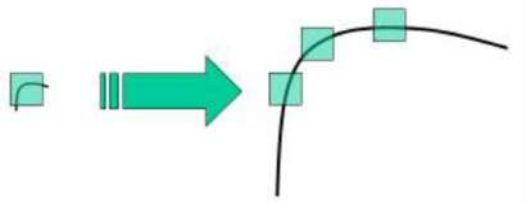
In the last couple of years, many corner detection techniques have been developed for many useful applications like Image Recognition, Image Matching etc. One famous such Corner Detection technique is Harris Corner Detection, which was a breakthrough in Corner Detection techniques which was rotation invariant. But still it wasn't Scale Invariant and it was still a research area. Then, D.Lowe from University of British Columbia proposed a Scale Invariant Feature Extraction algorithm in 2004. However, when people started using it, they found that SIFT was slow and a speeded-up approach is needed. In 2006, Bay H, Tuytelaars and Van Gool proposed a faster algorithm named Speeded Up Robust Features (SURF).

➢ APPROACH AND PROCEDURES:

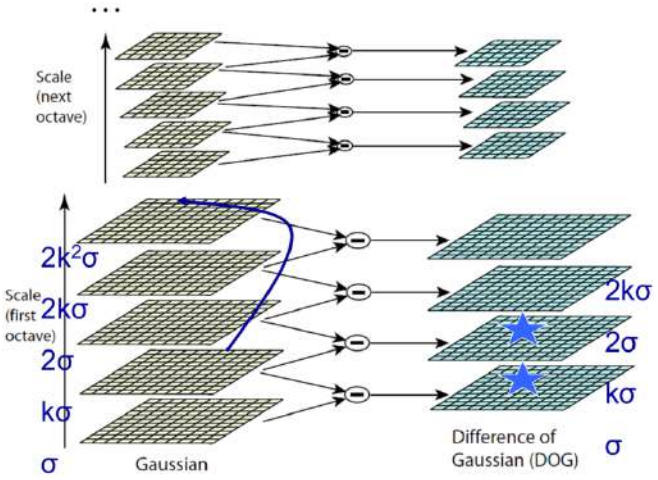
Scale-Invariant Feature Transform (SIFT):

SIFT Algorithm can be explained by four steps. They are given below:
(Source:http://opencv-python tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html)

- **Scale-space Extrema Detection:**
  The main problem with Harries Corner detection is when scaling is done on an image as shown below:

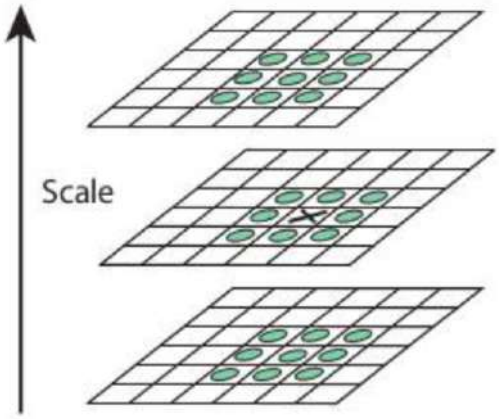


When the corner is scaled to a huge amount, the corners are confused and determining them is difficult. For this reason, SIFT uses different window sizes to detect key points with different scale.



Considering Laplacian of Gaussian (LoG) filter, it detects blobs of various sizes, with varying sigma σ. SIFT algorithm uses approximation of LOG, since the operation itself is little costly. Difference of Gaussians is obtained by subtracting Gaussians with two different sgma values like σ and $k\sigma$. After finding Laplacian of Gaussians, local extrema over scale and space is found.

From the example below, for the pixel marked as cross, it is compared with surrounding 8 neighbors and 9 pixels from top-next and previous-below scales. If it is a local extremum, it is considered as potential as key point.

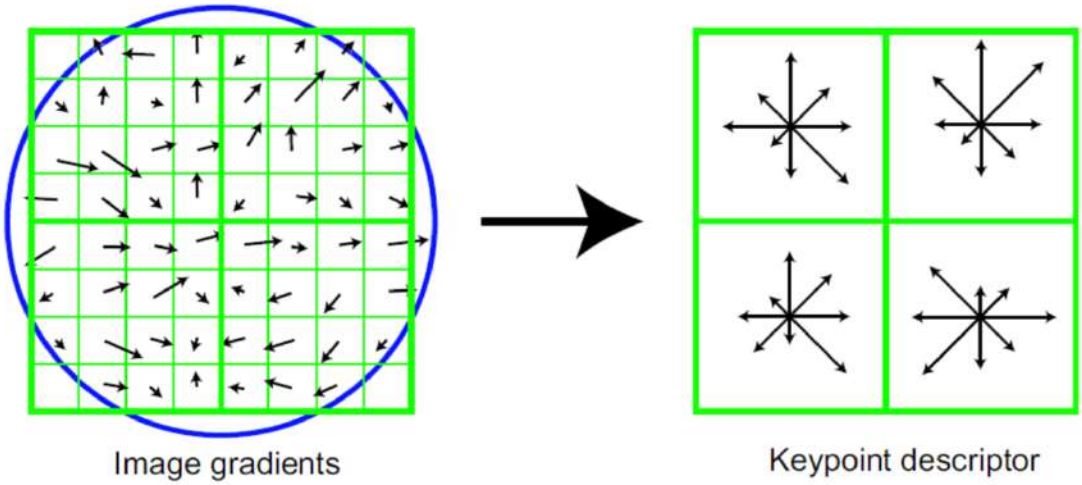


- **Keypoint Localization:**
  Once Keypoints are found using the above procedure, important keypoints need to be retained and others are to be discarded. Using Taylor Series expansion of scale space, a threshold called contrastThreshold is used. If the local extrema value is less than this threshold, the extrema is rejected.

  Similarly, LOG is very sensitive to edges and we need to filter out the edges detected to get corner points. So, a threshold called edgeThreshold is found using Hessian Matrix for this. Finally, low contrast key points and edge points are eliminated to have important key points.

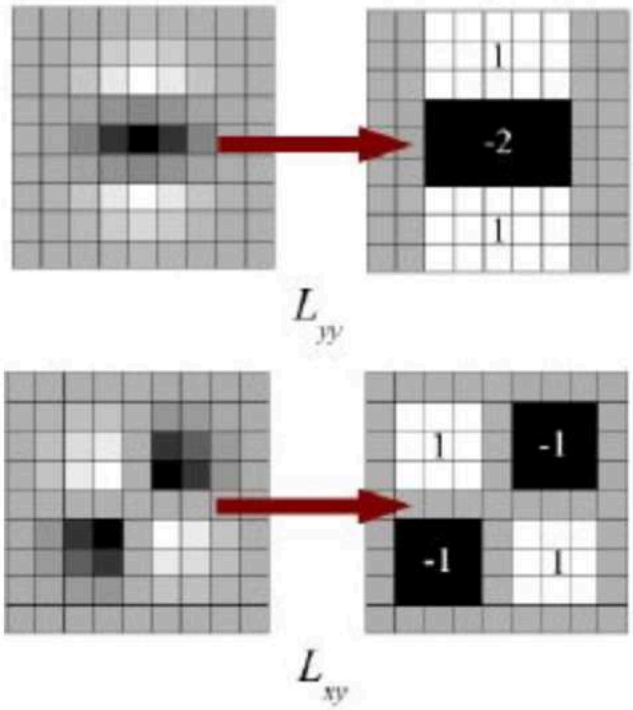
- **Orientation Assignment:**
  This part of the step is to assign orientation to each keypoint for rotation invariance. Based on a neighborhood taken around the keypoint location, depending on its scale, gradient magnitude and direction is calculated for that region. This helps to create keypoints of same scale and location but different directions.

- **Keypoint Descriptor:**
  This part of the process, a keypoint descriptor is created. For 16*16 neighbor of the keypoint, it is divided into 16 sub-blocks of size 4*4. For each sub block 8 bin orientation histogram is obtained. Totally 128 bin values are got. It is represented as a vector to form keypoint descriptor.

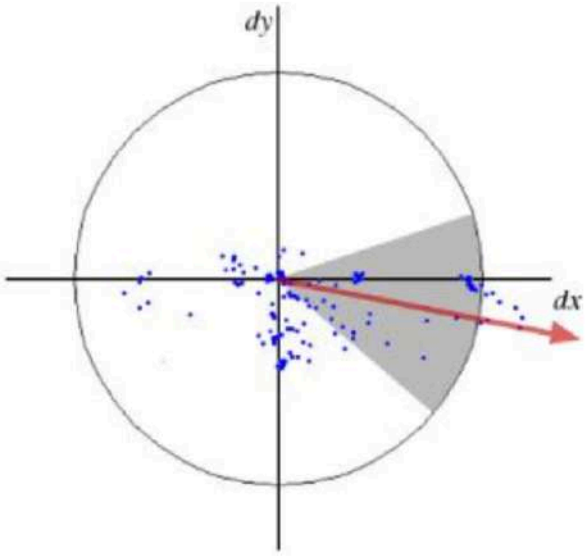


Speeded-Up Robust Features (SIFT):
(Source: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html)

It is an advanced and speeded up process to facilitate similar feature extraction like SIFT. SURF approximates LOG filter using a box filter. The advantage of using a box filter is that the convolution with box filter is easy in computation. The approximation can also be done for different scales. The determinant of Hessian matrix is used to find the scale and location in SURF. An illustration is shown below.



For orientation, wavelet responses are used in horizontal and vertical direction of 6*6 neighbors. They are plotted, and the orientation is calculated as shown below. A sliding window of angle 60 degrees is used to calculate the maximum orientation.



Again, for feature description. Wavelet responses are used in horizontal and vertical directions. 20*20 neighborhood is taken for a keypoint and it is divided into 4*4 sub regions. Horizontal and vertical wavelet responses are taken for each sub region and a vector is formed for a descriptor. On the whole, SURF adds a lot of features to improve the speed in every step.

Algorithm Implemented (C++) :

main() Function:

- Read given *bumblebee.jpg, optimus_prime.jpg, Ferrari_1.jpg, Ferrari_2.jpg* images using *imread()* function
- Define minHessian = 400
- Call *detectSIFTKeyPoints()* and *detectSURFKeyPoints()* function
- Write outputs using *imwrite()*

detectSIFTKeyPoints() Function:

- Declare a std vector for keypoints
- Define Matrix for output image
- Define detector using *SIFT::create()* function
- Detect keypoints using *detect()* function
- Draw key points on output image using *drawkeypoints()*

detectSURFKeyPoints() Function:

- Declare a std vector for keypoints
- Define Matrix for output image
- Define detector using *SURF::create()* function
- Detect keypoints using *detect()* function
- Draw key points on output image using *drawkeypoints()*

➢ EXPERIMENTAL RESULTS:

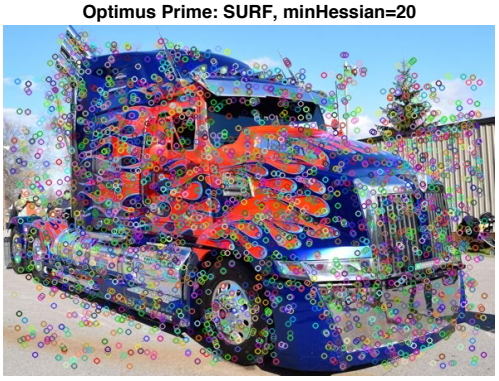
SIFT AND SURF FOR MINHESSIAN = 20 – ON COLOR IMAGES

Bumblebee: SIFT, minHessian=20

Bumblebee: SURF, minHessian=20

Optimus Prime: SIFT, minHessian=20

Optimus Prime: SURF, minHessian=20



SIFT AND SURF FOR MINHESSIAN = 400 – ON COLOR IMAGES

SIFT, minHessian=400

SURF, minHessian=400

SIFT, minHessian=400

SURF, minHessian=400

SIFT AND SURF FOR MINHESSIAN = 500 – ON COLOR IMAGES

SIFT, minHessian=500

SURF, minHessian=500

SIFT, minHessian=500

SURF, minHessian=500

SIFT AND SURF FOR MINHESSIAN = 600 – ON COLOR IMAGES

SIFT, minHessian=600

SURF, minHessian=600

SIFT, minHessian=600

SURF, minHessian=600

SIFT AND SURF FOR MINHESSIAN = 10000 – ON COLOR IMAGES

Bumblebee: SIFT, minHessian=10000

Bumblebee: SURF, minHessian=10000

Optimus Prime: SIFT, minHessian=10000

Optimus Prime: SURF, minHessian=10000

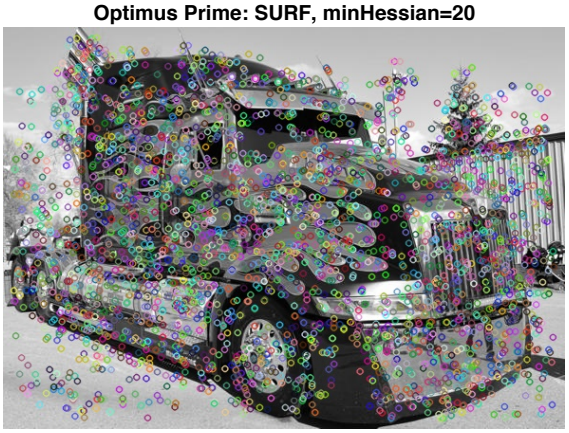
SIFT AND SURF FOR MINHESSIAN = 20 – ON GRAYSCALE IMAGES

Bumblebee: SIFT, minHessian=20

Bumblebee: SURF, minHessian=20

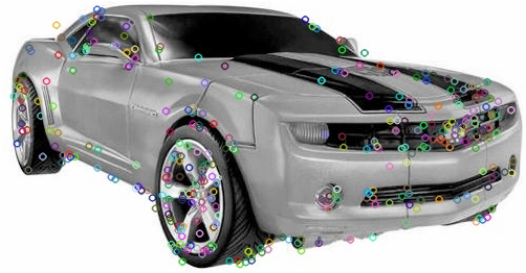
Optimus Prime: SIFT, minHessian=20

Optimus Prime: SURF, minHessian=20



SIFT AND SURF FOR MINHESSIAN = 400 – ON GRAYSCALE IMAGES

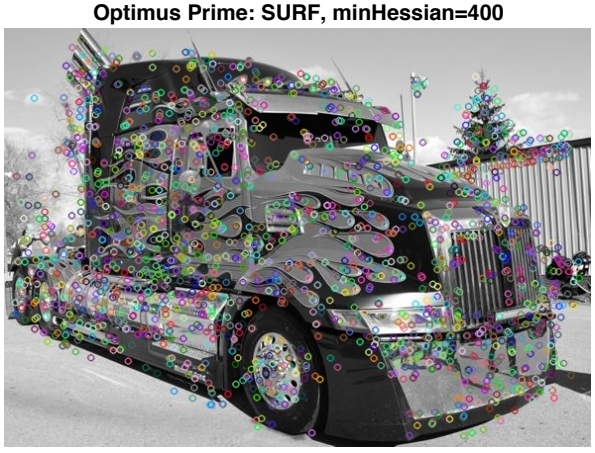
Bumblebee: SIFT, minHessian=400



Bumblebee: SURF, minHessian=400

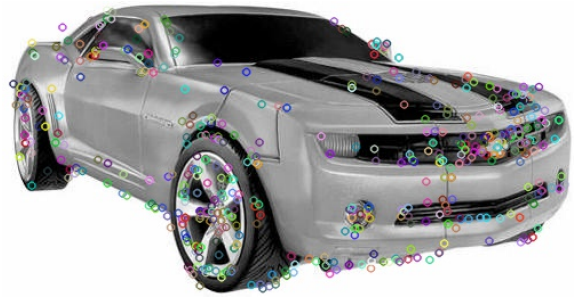
Optimus Prime: SIFT, minHessian=400

Optimus Prime: SURF, minHessian=400



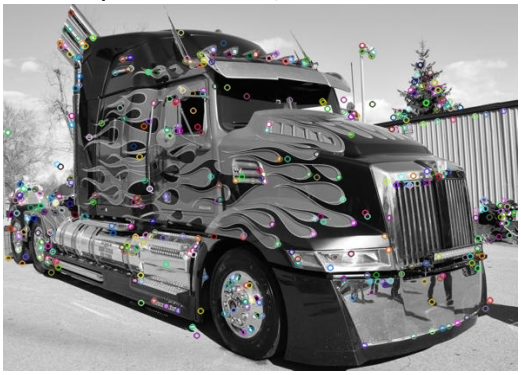
SIFT AND SURF FOR MINHESSIAN = 500 – ON GRAYSCALE IMAGES

Bumblebee: SIFT, minHessian=500

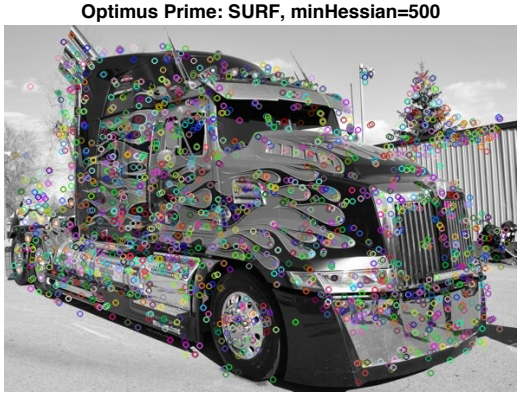


Bumblebee: SURF, minHessian=500

Optimus Prime: SIFT, minHessian=500

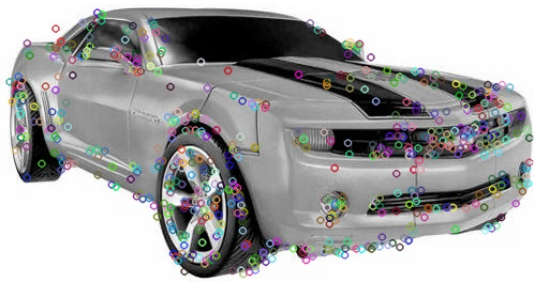


Optimus Prime: SURF, minHessian=500



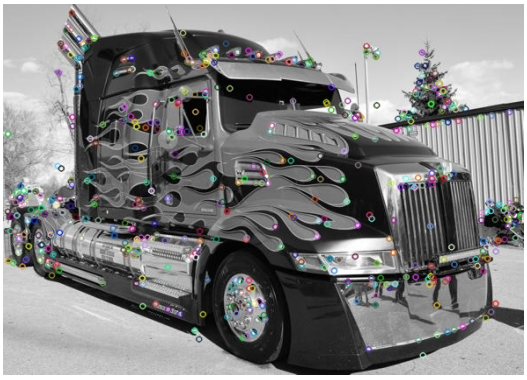
SIFT AND SURF FOR MINHESSIAN = 600 – ON GRAYSCALE IMAGES

Bumblebee: SIFT, minHessian=600



Bumblebee: SURF, minHessian=600

Optimus Prime: SIFT, minHessian=600



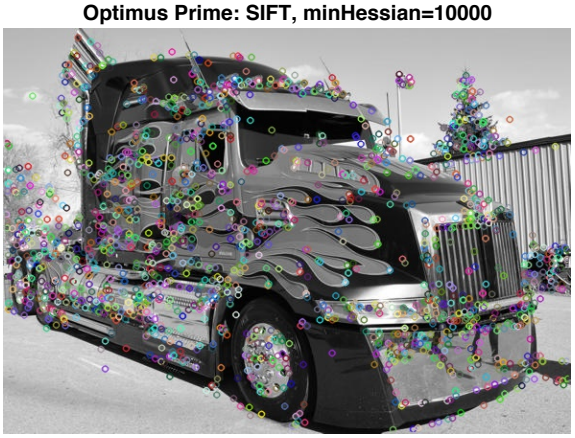
Optimus Prime: SURF, minHessian=600

SIFT AND SURF FOR MINHESSIAN = 10000 – ON GRAYSCALE IMAGES

Bumblebee: SIFT, minHessian=10000

Bumblebee: SURF, minHessian=10000

Optimus Prime: SIFT, minHessian=10000



Optimus Prime: SURF, minHessian=10000

➢ DISCUSSION:

The above SIFT and SURF Feature extraction from images were done using openCV library and XCode. openCV was installed on Mac OS with Contrib using the procedure mentioned in the link: Source: https://www.youtube.com/watch?v=hl-tS3TayyU

- Using openCV library with C++, SIFT and SURF salient points as features were extracted on given four images for different values of minHessian
- I tried for various minHessian values = 20, 400, 500, 600, 10000.
- For minHessian Value = 20, there are no SIFT features extracted. SURF features are many but they don't like important features
- For minHessian Value = 10000, there are no SURF features extracted. SIFT features are many but they don't like important features
- Below sources say that for normal images, typical minHessian value = 400 to 600 (Source: https://stackoverflow.com/questions/17613723/whats-the-meaning-of-minhessian-surffeaturedetector)
- It is proved that SURF is 3 times faster than SIFT and is better in all categories
- Also, SURF gives more number of keypoints than SIFT with such less computation time. The concept of wavelet responses and box filters are making the computation faster when compared to SIFT.
- It is also proved that, SURF is always good in computing keypoints even when images are blurred and rotated
- On comparison in terms of viewpoint change and illumination change, it is proved that SIFT performs better than SURF
- Thus, SIFT and SURF has its own pros and cons. One can choose which method they would want to use, depending upon the application considering parameters like speed, blurring, rotation, viewpoint change and illumination change.
- A comparison table is given below:

Parameter Considered	SIFT	SURF
Computation Speed	-	Best
Real Time Applications	-	Best
Blurred images	-	Best
Rotation changes	-	Best
Viewpoint Changes	Best	-
Illumination Changes	Best	-

- Some applications of SIFT and SURF feature extraction are given below:
  - Image Matching problems
  - Image Recognition problems
  - Get Structure from Motion
  - Image Stitching
  - Bag of Visual words for object recognition

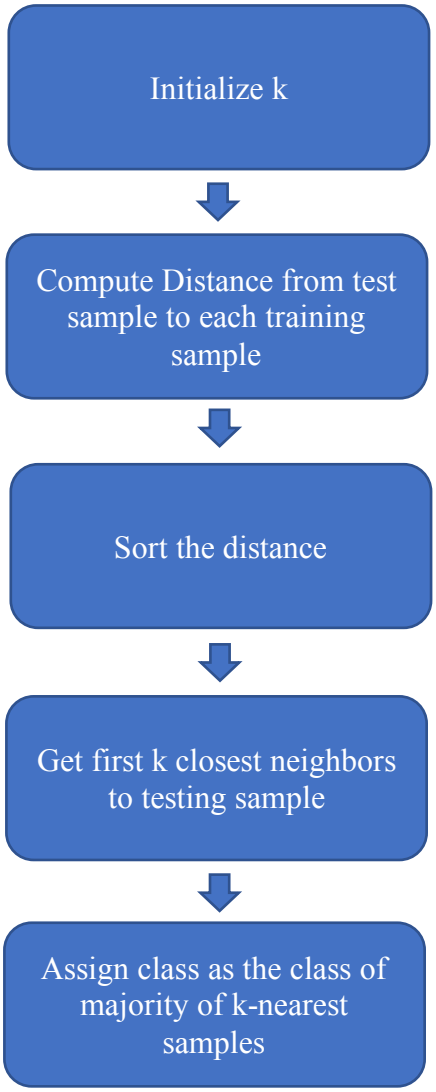
H) IMAGE MATCHING:

➢ ABSTRACT AND MOTIVATION:

From the above part of question, for given four images, I extracted SIFT and SURF features using openCV in C++. Also, in the discussion part, I listed out various applications of such salient points as feature extraction. This part of the question is the application of SIFT and SURF feature extraction. I have performed image matching on the given input images and shown below.

➢ APPROACH AND PROCEDURES:

Using SIFT and SURF algorithm detailed in previous part of the question, I have implemented Image Matching. For Image Matching Flann based Matcher is used. FLANN is the abbreviation for Fast Library for Approximate Nearest Neighbor. It is the optimized algorithm for Nearest Neighbor algorithm. It is proved fast even in very High Dimensional Feature space.



K-Nearest Neighbor Algorithm (KNN)

- Thus, the above shown flow chart is for K-Nearest Neighbor classification
- The descriptors are obtained from SIFT and SURF detectors
- Based on these descriptors, best matching points for both the images to be matched are found
- To find the matching points, above mentioned FLANN based matcher is used
- Once FLANN based matching points are found, good matching points need to be filtered and preserved
- For that minimum and maximum distance between the points are observed and based on a threshold 0.2, certain points are preserved, and certain points are rejected
- Finally based on good matching points, the final image is drawn
- The above process is carried repeatedly over various minHessian values = 400, 500 and 600
- The procedure is also repeated based for color and grayscale images to compare the differences between them

Algorithm Implemented (C++) :

main() Function:

- Read given *bumblebee.jpg, optimus_prime.jpg, Ferrari_1.jpg, Ferrari_2.jpg* images using *imread()* function
- Define minHessian = 400
- Call *imageMatchingUsingSIFT()* and *imageMatchingUsingSURF()* function for all inputs
- Write outputs using *imwrite()*

detectSIFTKeyPoints() Function:

- Declare a std vector for keypoints for both the images
- Define Matrix for output image
- Define detector using *SIFT::create()* function for both the images
- Detect descriptor using *detectAndCompute()* function for both the images
- Initialize *FlanBasedMatcher* and match both the descriptors
- Find minDist and maxDist of the
- Find best match using a threshold of 0.2
- Produce matching using *drawMatches()* using goodMatches points

detectSURFKeyPoints() Function:

- Declare a std vector for keypoints for both the images
- Define Matrix for output image
- Define detector using *SURF::create()* function for both the images
- Detect descriptor using *detectAndCompute()* function for both the images
- Initialize *FlanBasedMatcher* and match both the descriptors
- Find minDist and maxDist of the
- Find best match using a threshold of 0.2
- Produce matching using *drawMatches()* using goodMatches points

➢ EXPERIMENTAL RESULTS:

SIFT AND SURF FOR MINHESSIAN = 20 – ON COLOR IMAGES

Ferrari 1 - SIFT, minHessian=20

Ferrari 1 - SURF, minHessian=20

Ferrari 2 - SIFT, minHessian=20

Ferrari 2 - SURF, minHessian=20

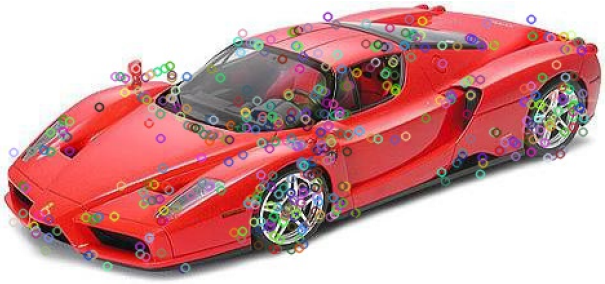
SIFT AND SURF FOR MINHESSIAN = 400 – ON COLOR IMAGES

Ferrari 1 - SIFT, minHessian=400

Ferrari 1 - SURF, minHessian=400

Ferrari 2 - SIFT, minHessian=400

Ferrari 2 - SURF, minHessian=400



SIFT AND SURF FOR MINHESSIAN = 500 – ON COLOR IMAGES

Ferrari 1 - SIFT, minHessian=500

Ferrari 1 - SURF, minHessian=500

Ferrari 2 - SIFT, minHessian=500

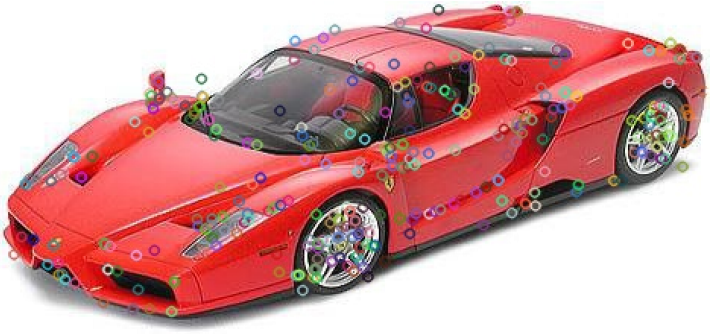
Ferrari 2 - SURF, minHessian=500

SIFT AND SURF FOR MINHESSIAN = 600 – ON COLOR IMAGES

Ferrari 1 - SIFT, minHessian=600

Ferrari 1 - SURF, minHessian=600

Ferrari 2 - SIFT, minHessian=600



Ferrari 2 - SURF, minHessian=600

SIFT AND SURF FOR MINHESSIAN = 10000 – ON COLOR IMAGES

Ferrari 1 - SIFT, minHessian=10000

Ferrari 1 - SURF, minHessian=10000

Ferrari 2 - SIFT, minHessian=10000

Ferrari 2 - SURF, minHessian=10000

SIFT AND SURF FOR MINHESSIAN = 20 – ON GRAYSCALE IMAGES

Ferrari 1 - SIFT, minHessian=20

Ferrari 1 - SURF, minHessian=20

Ferrari 2 - SIFT, minHessian=20

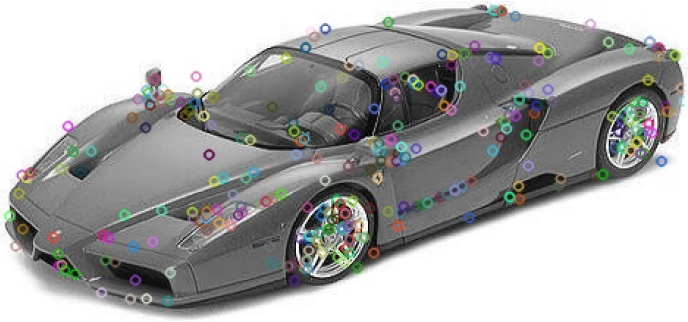
Ferrari 2 - SURF, minHessian=20

SIFT AND SURF FOR MINHESSIAN = 400 – ON GRAYSCALE IMAGES

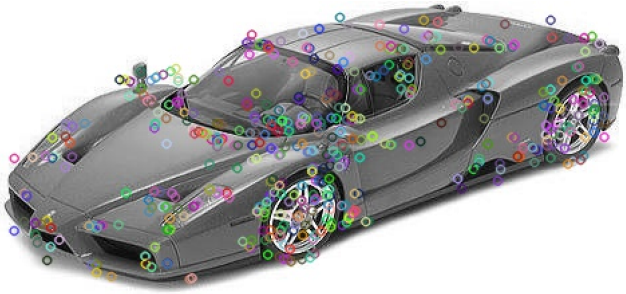
Ferrari 1 - SIFT, minHessian=400

Ferrari 1 - SURF, minHessian=400

Ferrari 2 - SIFT, minHessian=400



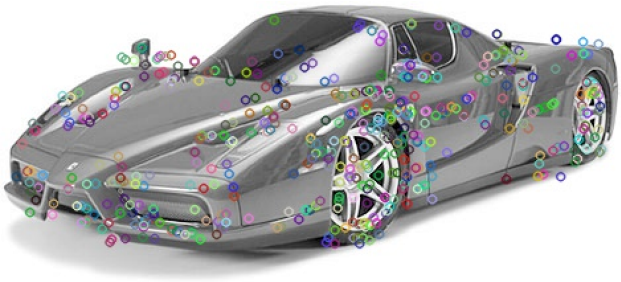
Ferrari 2 - SURF, minHessian=400



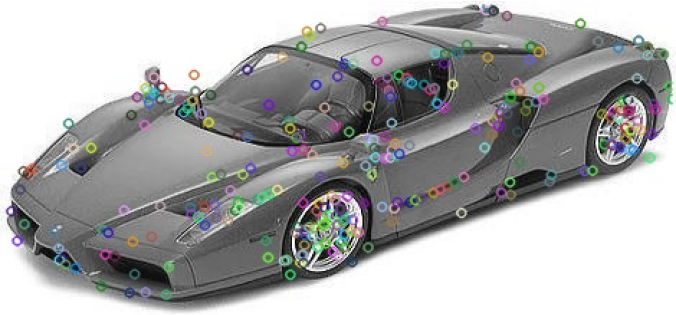
SIFT AND SURF FOR MINHESSIAN = 500 – ON GRAYSCALE IMAGES

Ferrari 1 - SIFT, minHessian=500

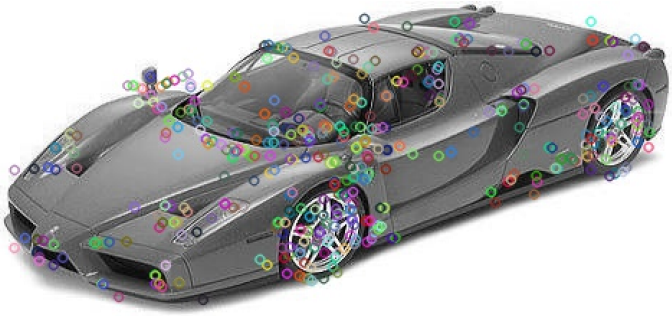
Ferrari 1 - SURF, minHessian=500



Ferrari 2 - SIFT, minHessian=500

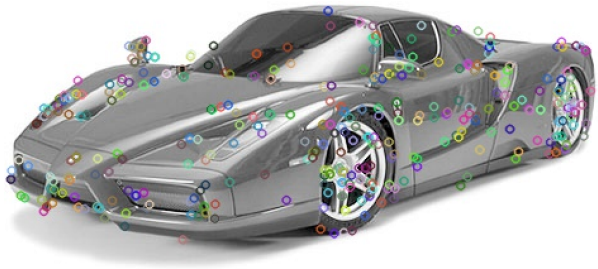


Ferrari 2 - SURF, minHessian=500

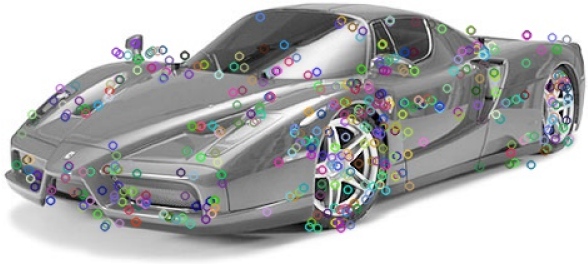


SIFT AND SURF FOR MINHESSIAN = 600 – ON GRAYSCALE IMAGES

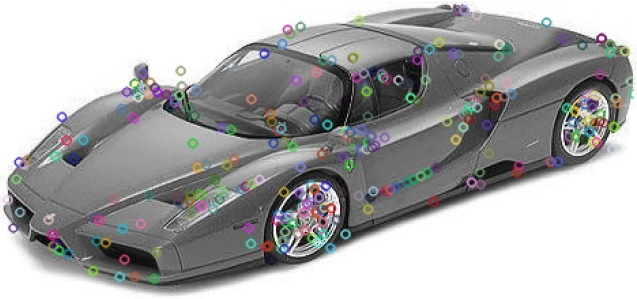
Ferrari 1 - SIFT, minHessian=600



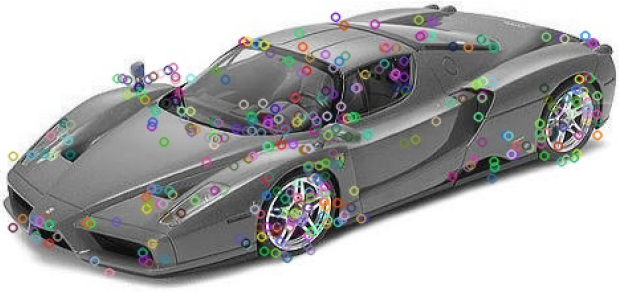
Ferrari 1 - SURF, minHessian=600



Ferrari 2 - SIFT, minHessian=600



Ferrari 2 - SURF, minHessian=600



SIFT AND SURF FOR MINHESSIAN = 10000 – ON GRAYSCALE IMAGES

Ferrari 1 - SIFT, minHessian=10000

Ferrari 1 - SURF, minHessian=10000

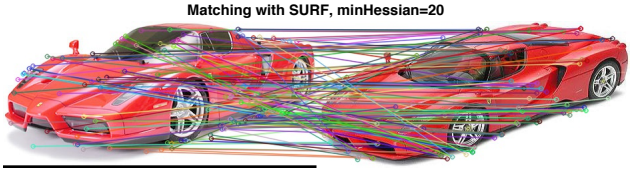
Ferrari 2 - SIFT, minHessian=10000

Ferrari 2 - SURF, minHessian=10000

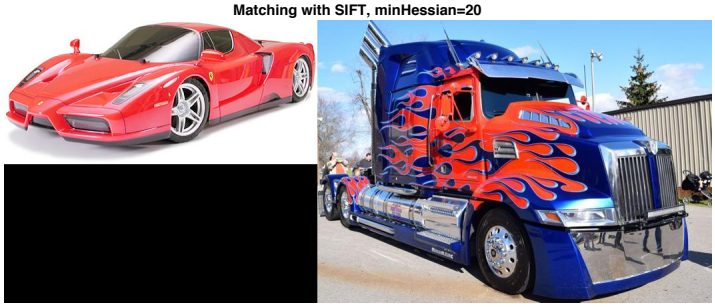
IMAGE MATCHING ON COLOR IMAGES USING MINHESSIAN=20

Matching with SIFT, minHessian=20

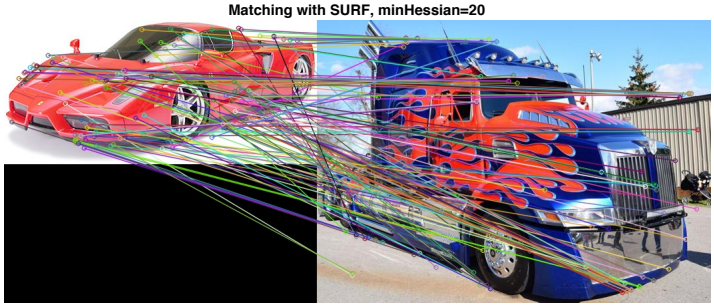
Matching with SURF, minHessian=20



Matching with SIFT, minHessian=20



Matching with SURF, minHessian=20



Matching with SIFT, minHessian=20

Matching with SURF, minHessian=20

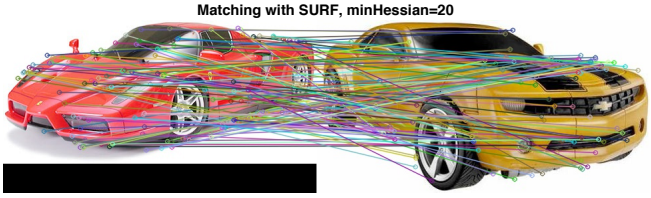
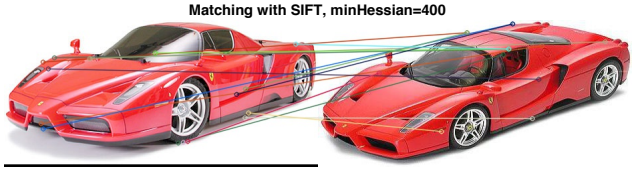
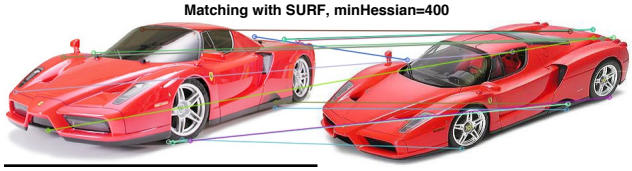


IMAGE MATCHING ON COLOR IMAGES USING MINHESSIAN=400

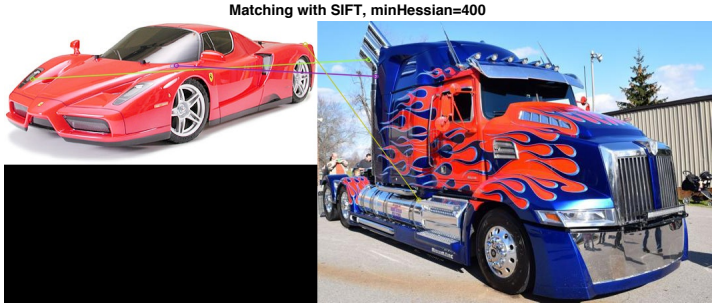
Matching with SIFT, minHessian=400



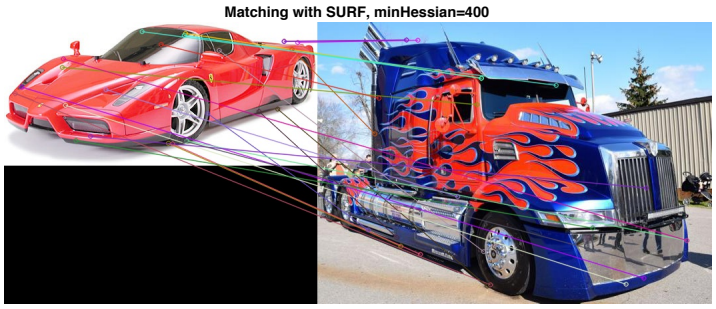
Matching with SURF, minHessian=400



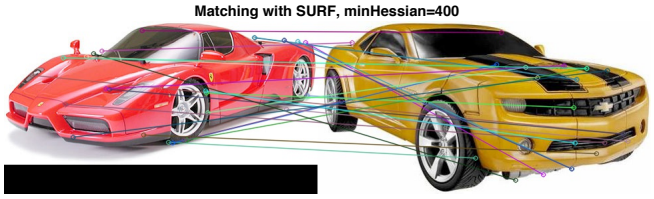
Matching with SIFT, minHessian=400



Matching with SURF, minHessian=400



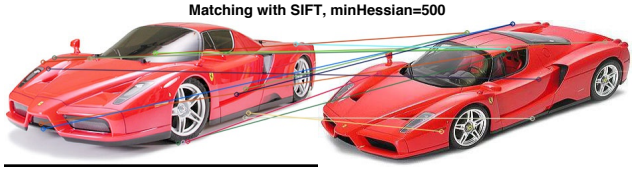
Matching with SURF, minHessian=400



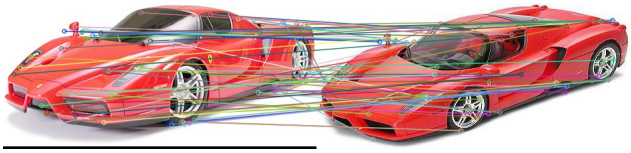
Matching with SIFT, minHessian=400

IMAGE MATCHING ON COLOR IMAGES USING MINHESSIAN=500

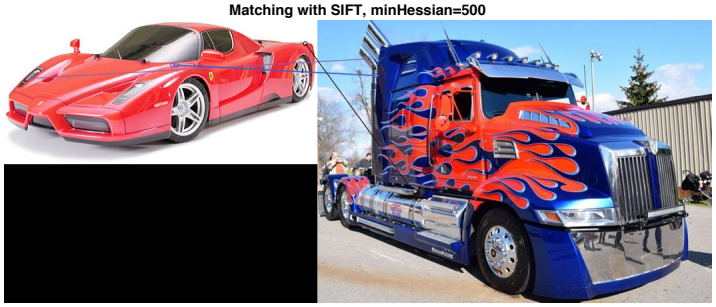
Matching with SIFT, minHessian=500



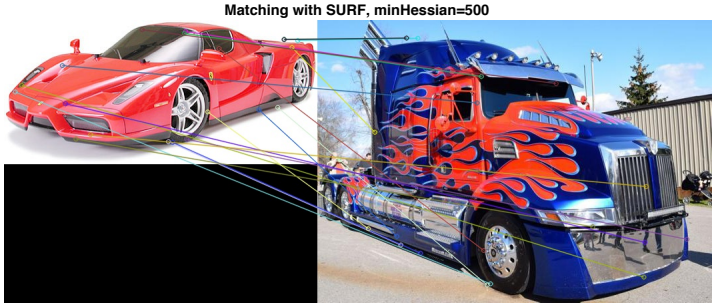
Matching with SURF, minHessian=500



Matching with SIFT, minHessian=500



Matching with SURF, minHessian=500



Matching with SIFT, minHessian=500

Matching with SURF, minHessian=500

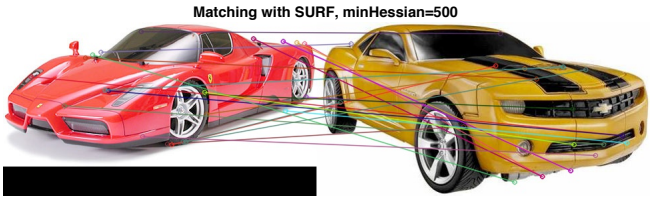
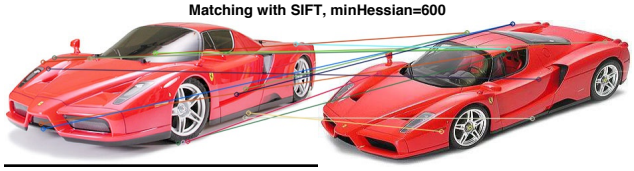
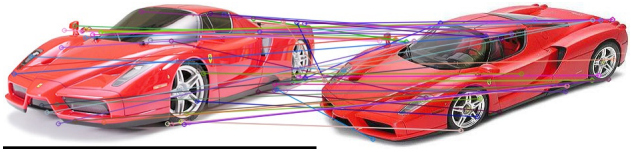


IMAGE MATCHING ON COLOR IMAGES USING MINHESSIAN=600

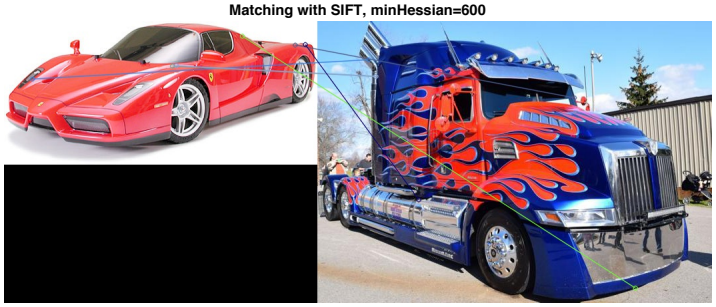
Matching with SIFT, minHessian=600



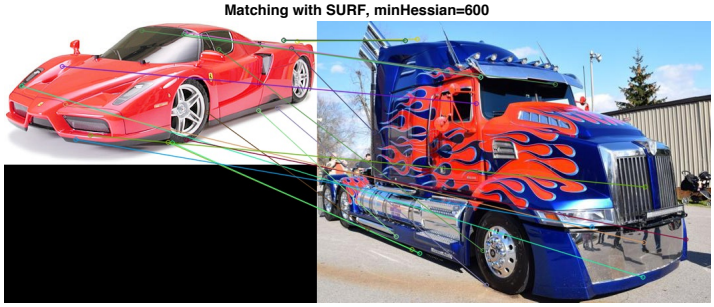
Matching with SURF, minHessian=600



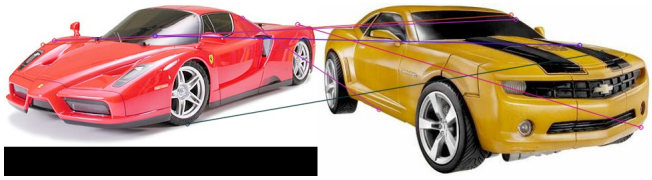
Matching with SIFT, minHessian=600



Matching with SURF, minHessian=600



Matching with SIFT, minHessian=600



Matching with SURF, minHessian=600

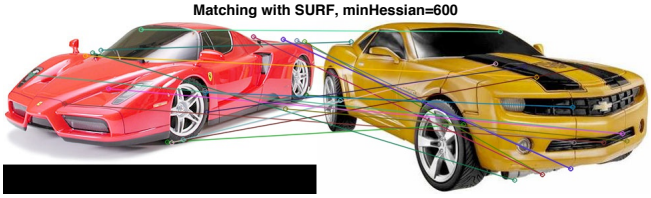
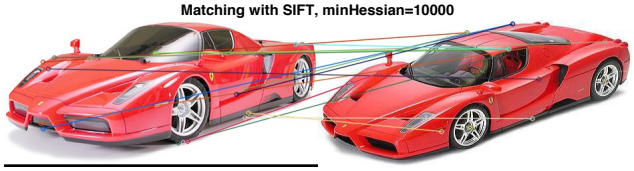
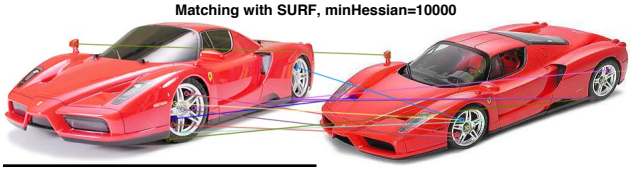


IMAGE MATCHING ON COLOR IMAGES USING MINHESSIAN=10000

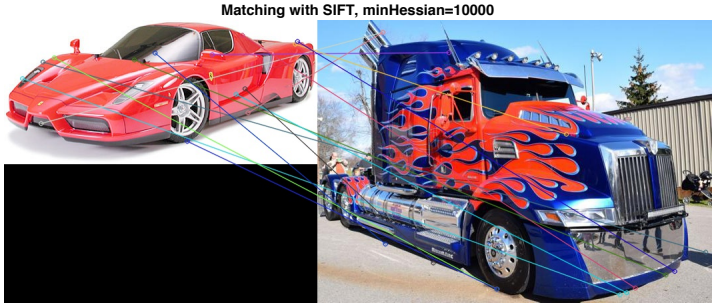
Matching with SIFT, minHessian=10000



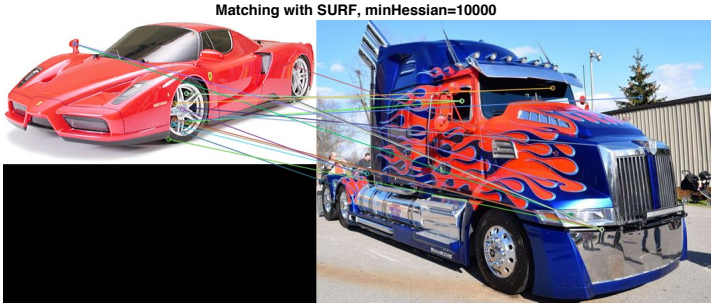
Matching with SURF, minHessian=10000



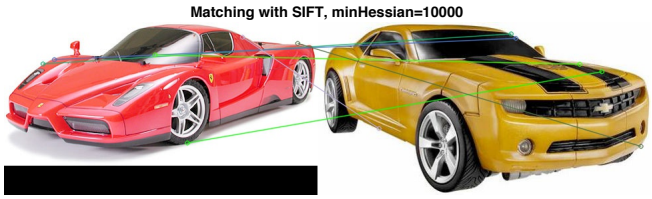
Matching with SIFT, minHessian=10000



Matching with SURF, minHessian=10000



Matching with SIFT, minHessian=10000



Matching with SURF, minHessian=10000

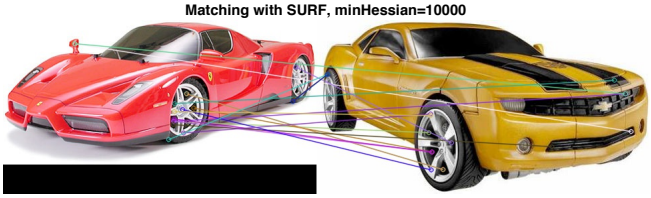
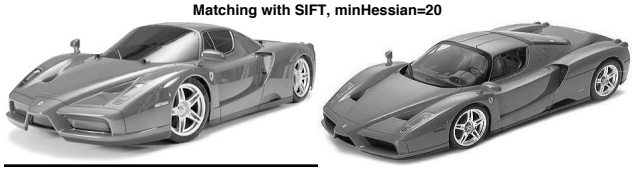
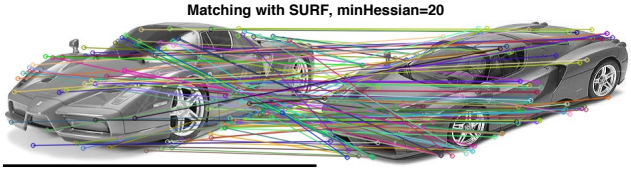


IMAGE MATCHING ON GRAYSCALE IMAGES USING MINHESSIAN=20

Matching with SIFT, minHessian=20

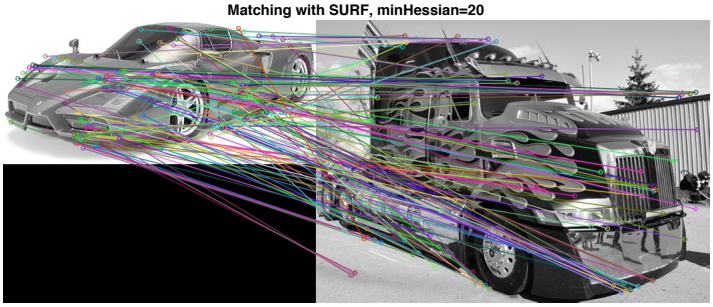


Matching with SURF, minHessian=20

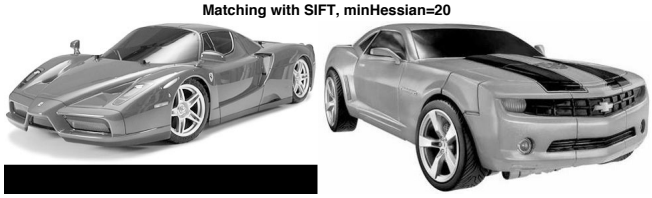


Matching with SIFT, minHessian=20

Matching with SURF, minHessian=20



Matching with SIFT, minHessian=20



Matching with SURF, minHessian=20

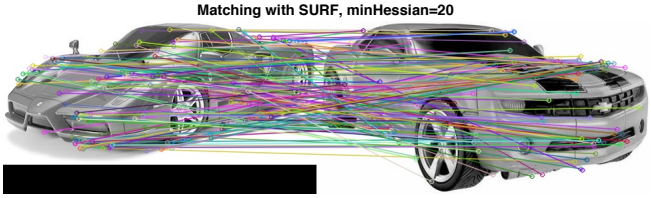
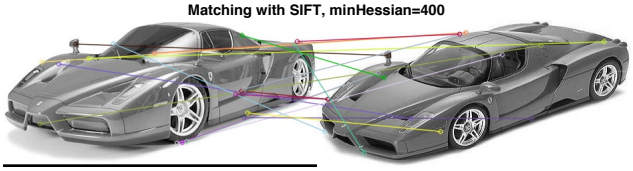
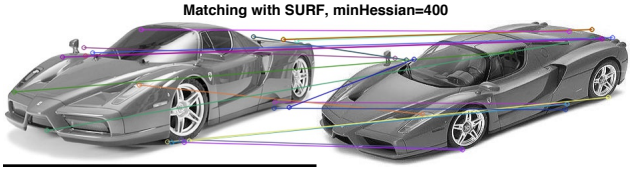


IMAGE MATCHING ON GRAYSCALE IMAGES USING MINHESSIAN=400

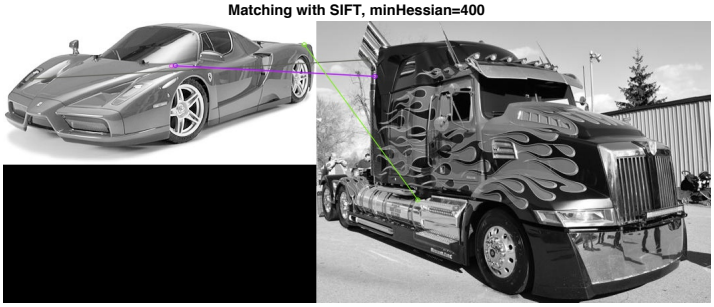
Matching with SIFT, minHessian=400



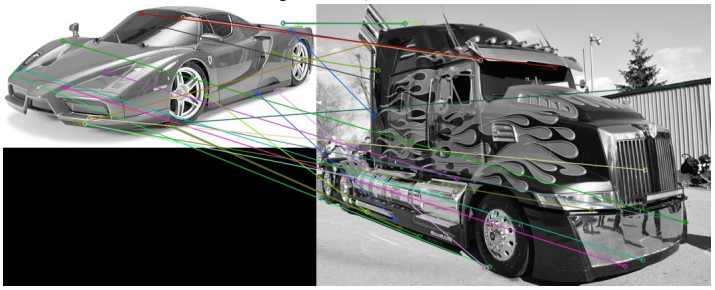
Matching with SURF, minHessian=400



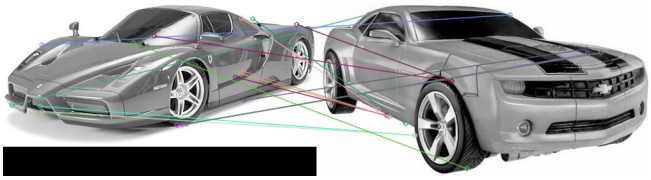
Matching with SIFT, minHessian=400



Matching with SURF, minHessian=400



Matching with SIFT, minHessian=400



Matching with SURF, minHessian=400

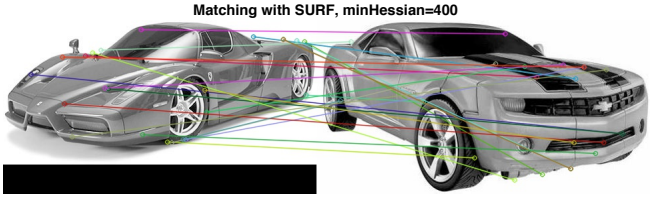
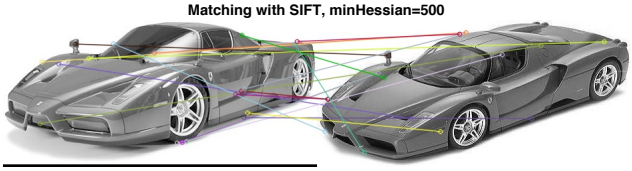
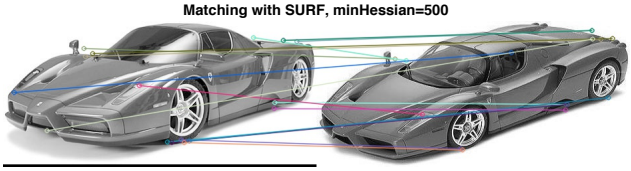


IMAGE MATCHING ON GRAYSCALE IMAGES USING MINHESSIAN=500

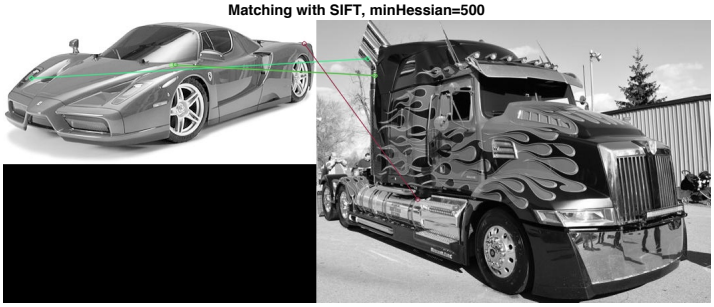
Matching with SIFT, minHessian=500



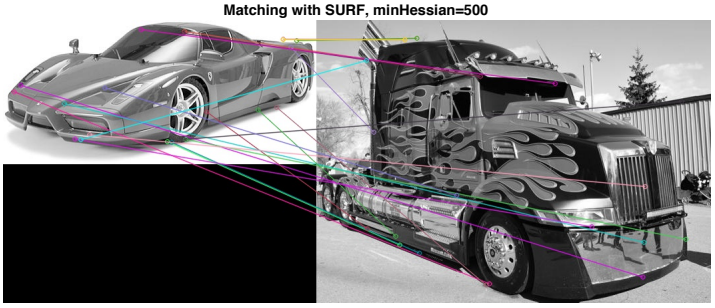
Matching with SURF, minHessian=500



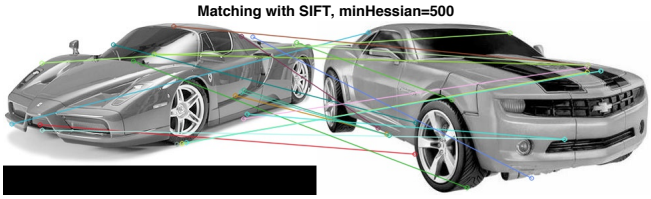
Matching with SIFT, minHessian=500



Matching with SURF, minHessian=500



Matching with SIFT, minHessian=500



Matching with SURF, minHessian=500

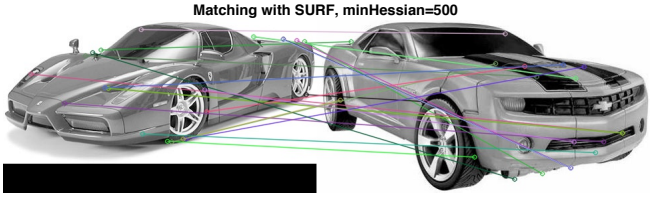
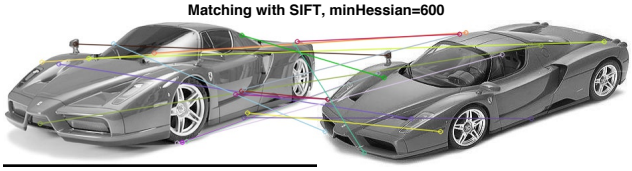
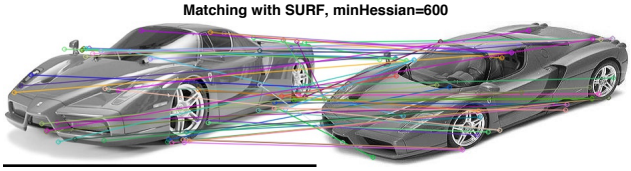


IMAGE MATCHING ON GRAYSCALE IMAGES USING MINHESSIAN=600

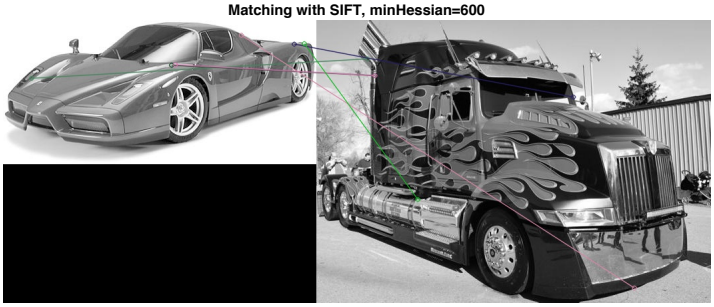
Matching with SIFT, minHessian=600



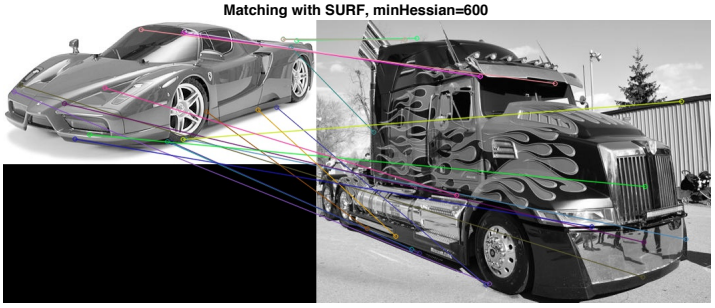
Matching with SURF, minHessian=600



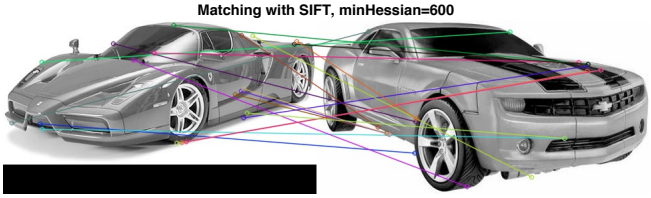
Matching with SIFT, minHessian=600



Matching with SURF, minHessian=600



Matching with SIFT, minHessian=600



Matching with SURF, minHessian=600

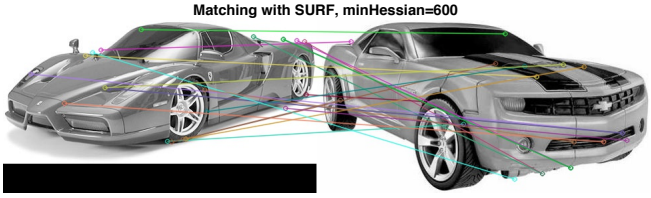
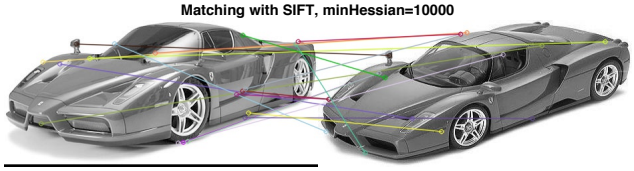
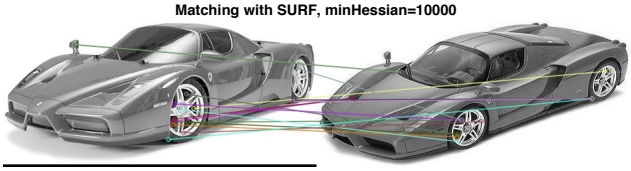


IMAGE MATCHING ON GRAYSCALE IMAGES USING MINHESSIAN=10000

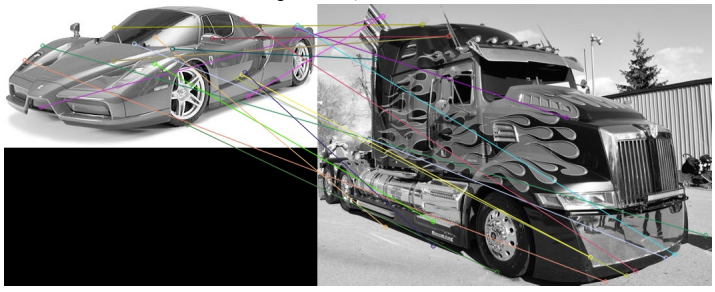
Matching with SIFT, minHessian=10000



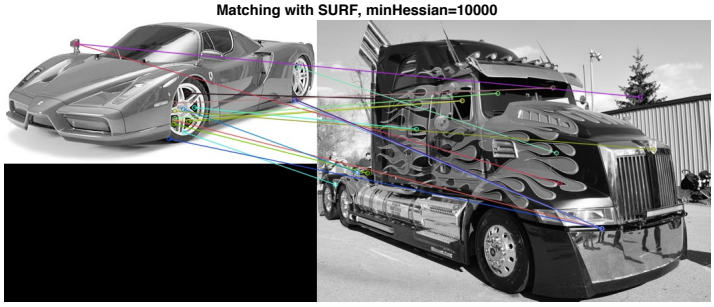
Matching with SURF, minHessian=10000



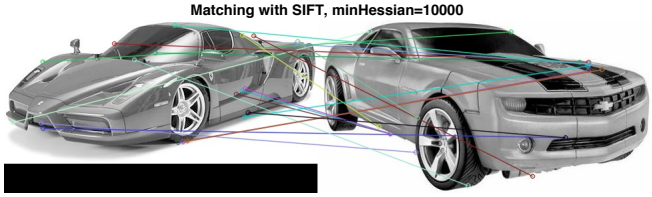
Matching with SIFT, minHessian=10000



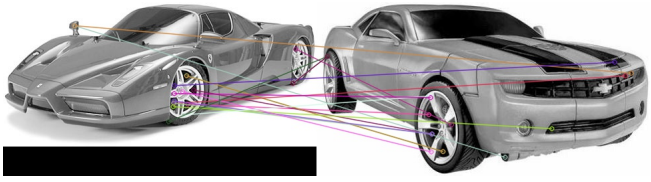
Matching with SURF, minHessian=10000



Matching with SIFT, minHessian=10000



Matching with SURF, minHessian=10000



➢ DISCUSSION:

- In this part of the question I explained how SIFT/SURF feature extraction can be combined with FLANN based Image Matching Techniques to find matching points between different input images
- The above experimental results are shown first for SIFT and SURF features extraction for Ferrari 1 and 2 images. They are shown for both color and grayscale images and for different minHessian values = 20, 400, 500, 600, 10000
- Next, I show image matching keypoints with lines between the images showing which keypoints from image 1 are matched to which keypoints from iamge 2
- Following observations can be inferred from the above experimental results:
  - SURF as expectedly extracts more keypoint descriptors from Ferrari 1 and 2 than SIFT exttaction
  - SIFT/SURF extraction almost remains the same for both color and grayscale images. I found online that converting the color to grayscale images will given better feature extraction. But I found, from the given images that it is not always true. My argument is also supported by the link given below: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0029740
  - SIFT/SURF extraction is very bad for min Hessian values = 20 and 10000. Hence matching lines are also bad based on FLANN matcher
  - The number of good matching points between Ferrari 1 and Ferrari 2 is greater for SURF based feature extraction method for any minHessian value, color images and grayscale images
  - This shows that Ferrari 1 and Ferrari 2 is best matched images
  - The number of good image matching points between Ferrari 1 and Optimus Prime, Ferrari 1 and Bumblebee are significantly less than Ferrari 1 and Ferrari 2 images.
  - One might think since second and third case are completely different images and hence there should be not matching points at all. But since thery are vehicles and similar features are expected being vehicles, like glass in the front, doors, wheels, there are some matching points.
- One can also do Brute Force Matcher with ORB descriptors or SIFT descriptors. But FLANN based Image descriptors are better in terms of computational speed and results of matching
  - http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html
- Also, FREAK based image descriptors are proved to have competetive performances when compared with SIFT or SURF based descriptors in various computer vision problems. The paper describing it is given below:
  - https://www.researchgate.net/publication/269291815_SIFT_vs_FREAK_Assessing_the_usefulness_of_two_keypoint_descriptors_for_3D_face_verification

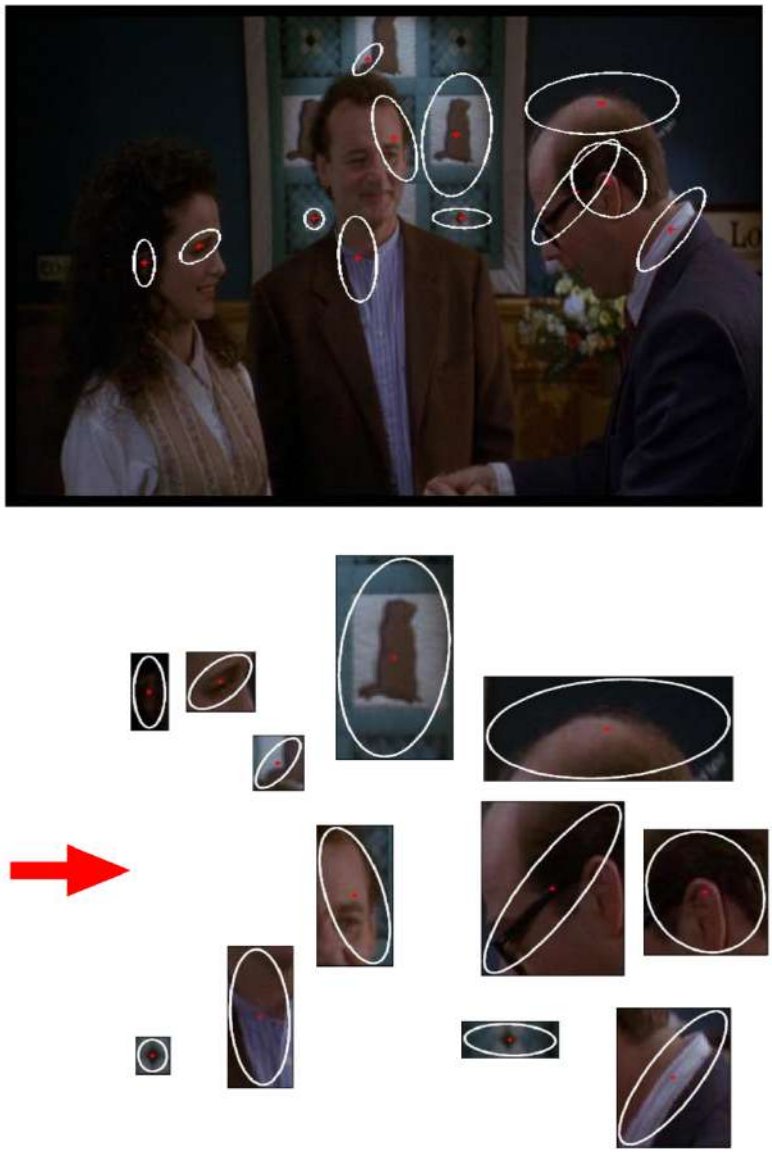
I) BAG OF WORDS:

➢ ABSTRACT AND MOTIVATION:

The previous parts of question, I performed SIFT and SURF on given four images. I also matched images using FLANN based image matcher. In this part of the question, I have built a more robust image matching algorithm that is commonly used in the Computer Vision world. This Image Matching algorithm is built based on Bag of Words concept and Histogram Matching. Going forward I have explained the algorithm and shown results for Matching using Bag of Words.

➢ APPROACH AND PROCEDURES:

Bag of Words is commonly used in Computer Vision World outside for various Object Recognition techniques. The words formed in Codebook are from descriptors acquired by SIFT method. For this part of the question I have trained Code Book using Ferrari 1, Optimus Prime and Bumblebee images. Better explanation of Codebooks is given below:



From the above images, image 1 shows feature point descriptors extracted using SIFT technique. The second image shows how code books are formed. If you look carefully, the feature points are centers and an elliptical curve is shown around. Any point within that curve belongs to that cluster. And codebooks are formed in this way such that centroids are decided for every word/image and other descriptors are clustered to the closest centroid by K-Means algorithm. (Source: http://www.robots.ox.ac.uk/~az/icvss08_az_bow.pdf)

Once all feature points are beings clustered, a histogram of count of descriptors belonging to the cluster/code word is formed for every image. Once the histograms are found for every image, histogram matching is done to match images. The test image is Ferrari 2. Histograms are found based on Code Words, for all 4 images including the test image. Then Ferrari 2's histogram is matched with the other 3 histograms (training image histograms)

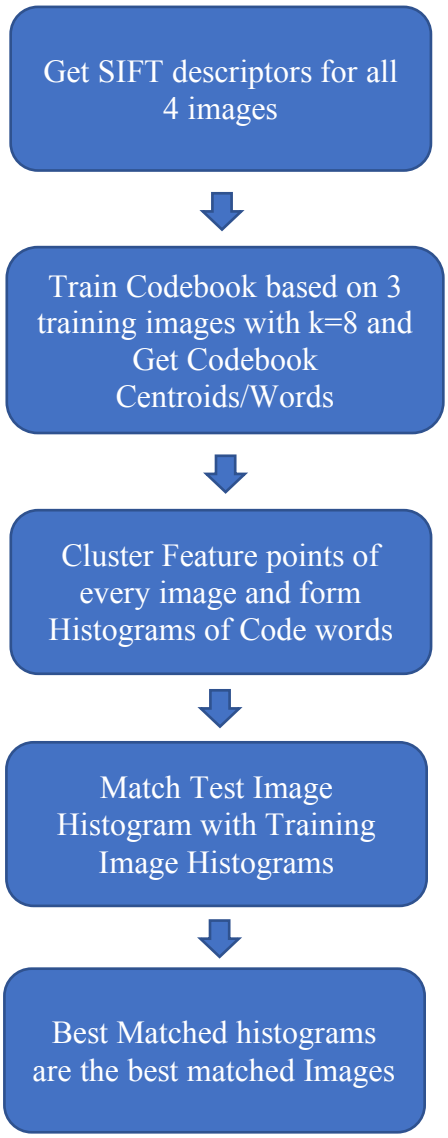
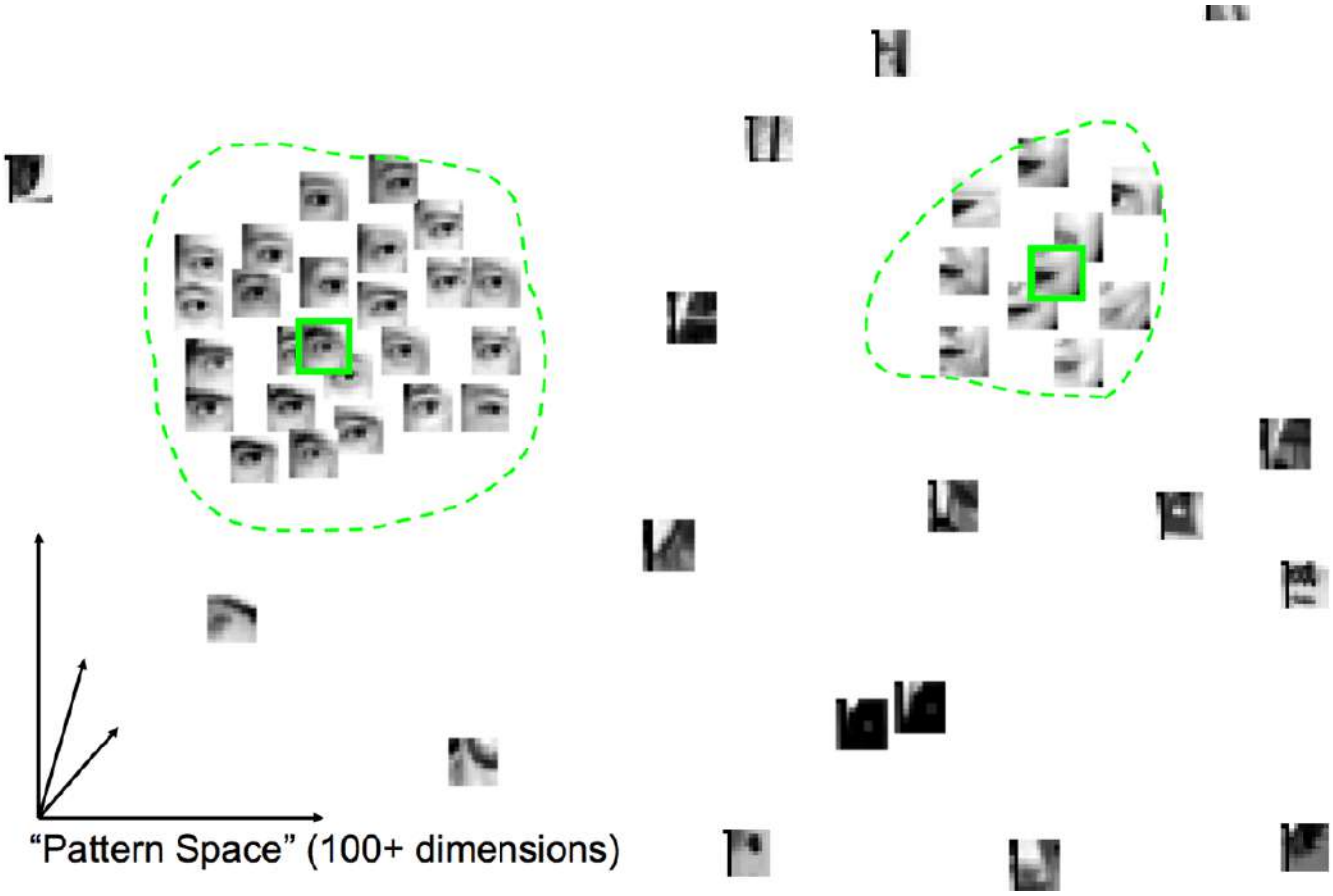


Image Matching using Bag of Words and Histogram Matching



Example Clustering of Bag of Words (Square box indicated Words)
(Source: http://www.robots.ox.ac.uk/~az/icvss08_az_bow.pdf)

HISTOGRAM MATCHING:

Histogram matching is done based on absolute error difference in the values of histogram. The best matched histogram is the one which has lowest error. The formula is given below.

$$\text{Hist Error} = \sum_{k=1}^{n} \text{abs}(\, H_{\text{test}}(k) - H_{\text{train}}(k))$$

Where, $H_{\text{test}}(k)$ = Frequency/Histogram Value for k^{th} word in test image,
$H_{\text{train}}(k)$ = Frequency/Histogram Value for k^{th} word in train image,
k = code word/ descriptor centroid,
n = number of clusters

- This Histogram Error is calculated for test image with every train image
- Best Matched Histogram is found

Algorithm Implemented (C++):

main() Function:

- Read given *bumblebee.jpg, optimus_prime.jpg, Ferrari_1.jpg, Ferrari_2.jpg* images using *imread()* function
- Define minHessian = 400
- Create an object for *bagOfWords* Class
- Call *detectAllImageSIFTDescriptors()* method
- Call *trainBagOfWords()* method
- Call *findAllImagesVocabHistogram()* method
- Call *matchTestImageHistogram()* method

bagOfWords Class:

- Define Constructor and allocate memories for required data structures
- Define Destructor and free memories for required data structures

detectSIFTDescriptors () Function:

- Declare a std vector for keypoints for both the images
- Define Matrix for descirptor
- Define detector using *SIFT::create()* function for both the images
- Detect descriptor using *detectAndCompute()* function for both the images

detectAllImageSIFTDescriptors() Function:

- Call *detectSIFTDescriptors()* for given 4 images

trainBagOfWords() Function:

- Define Bag or words object for *BOWKMeansTrainer* Class
- Add train image descriptors to the object using *add()* function
- Cluster them using *cluster()* function

findVocabHistogram() function:

- Loop through every descriptor
- Find the L2 distance between each descriptor and cluster centroids
- Find the closest cluster
- Increment the count for histogram array for that cluster

findVocabHistogram() function:

- Find histogram of all the images using *findVocabHistogram()* function

findVocabHistogram() function:

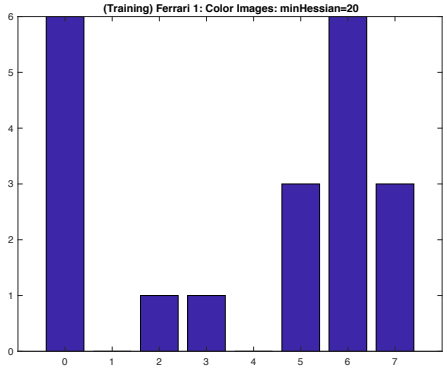
- Find histogram of all the images using *findVocabHistogram()* function

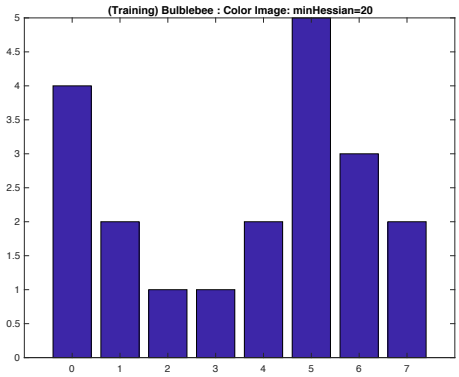
matchTestImageHistogram() function:

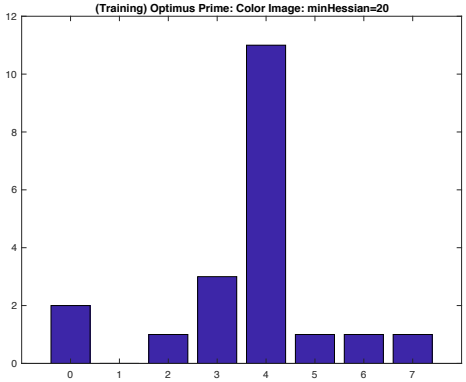
- Find error for all images using *findVocabHistogram*() function
- Match test image with training images which has minimum error

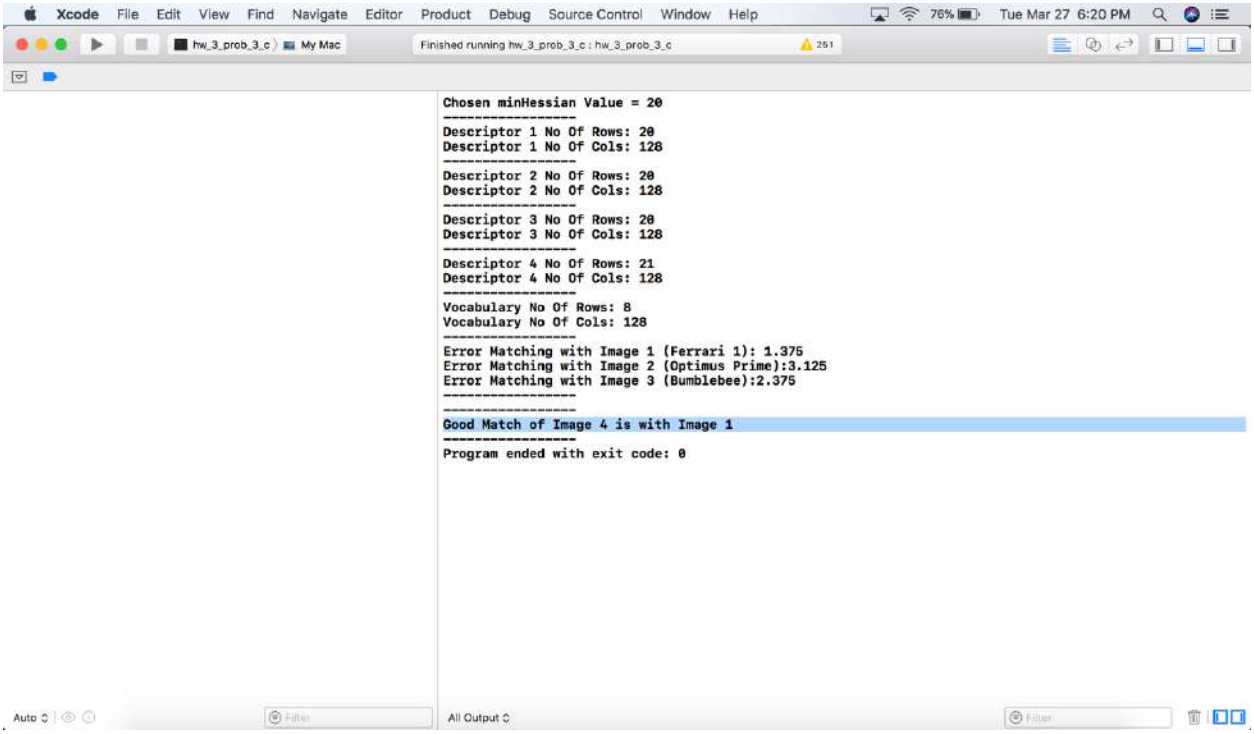
➢ EXPERIMENTAL RESULTS:

HISTOGRAM MATCHING ON COLOR IMAGES WITH MINHESSIAN=20

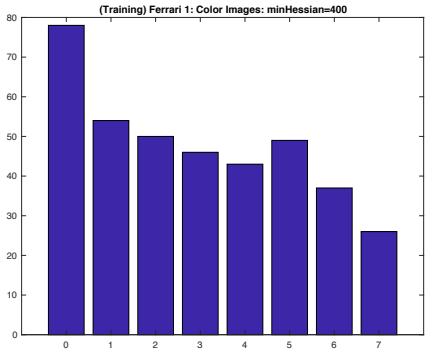


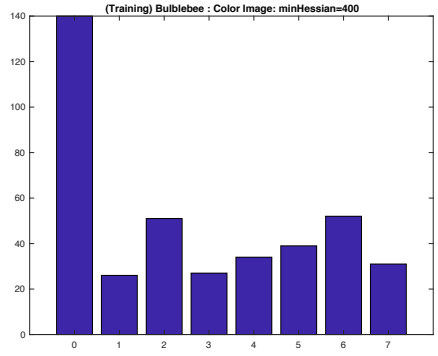


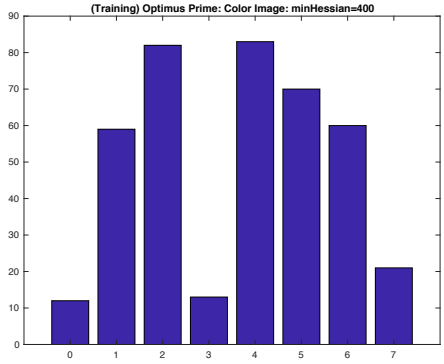


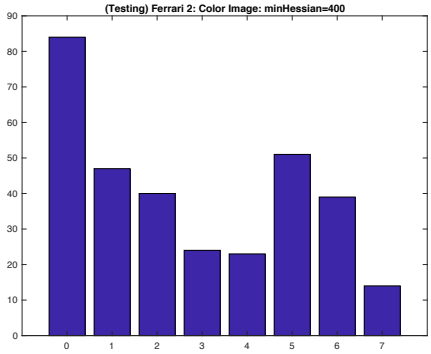


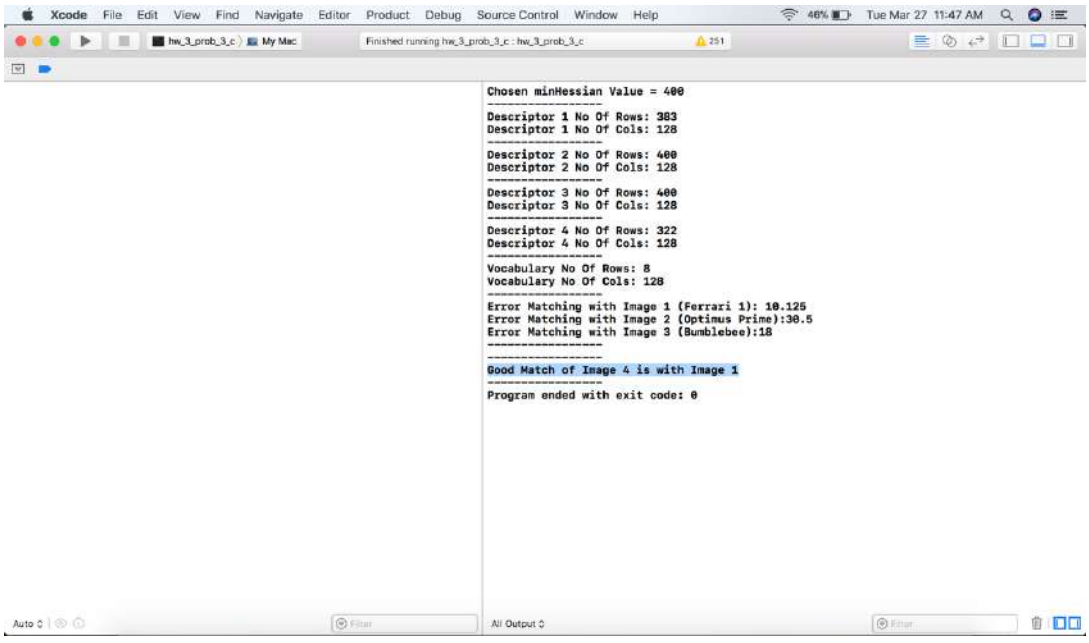
HISTOGRAM MATCHING ON COLOR IMAGES WITH MINHESSIAN=400



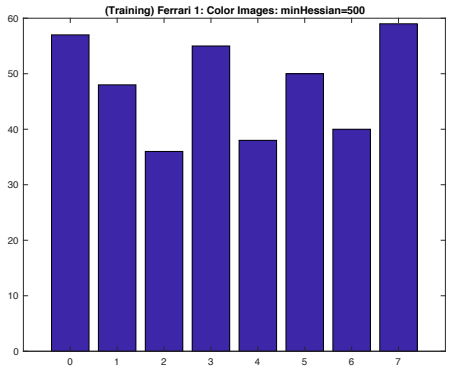


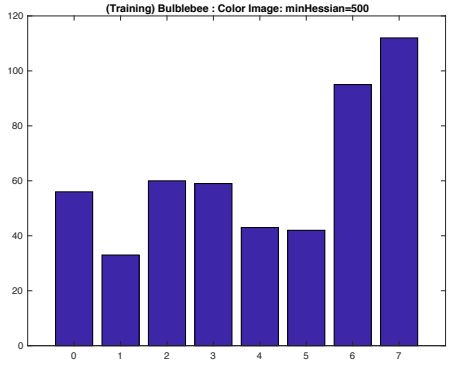


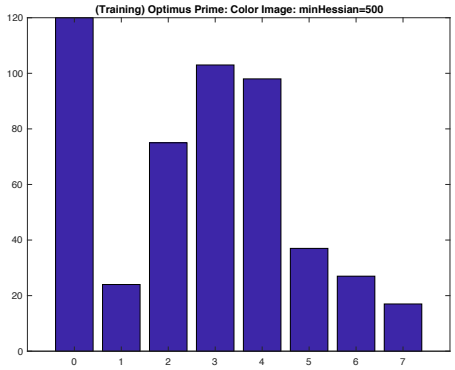


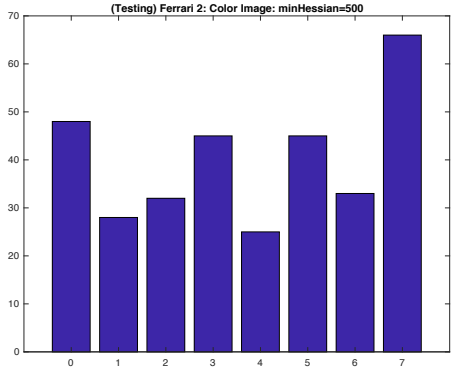


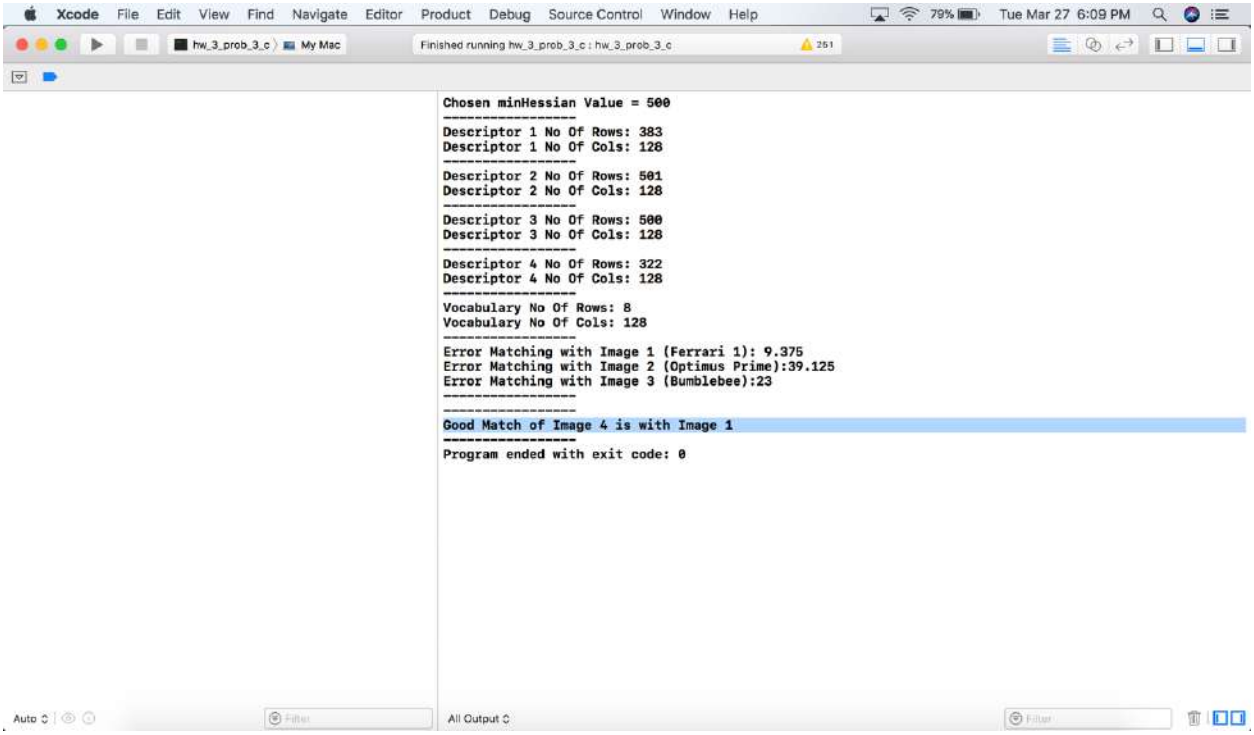
HISTOGRAM MATCHING ON COLOR IMAGES WITH MINHESSIAN=500



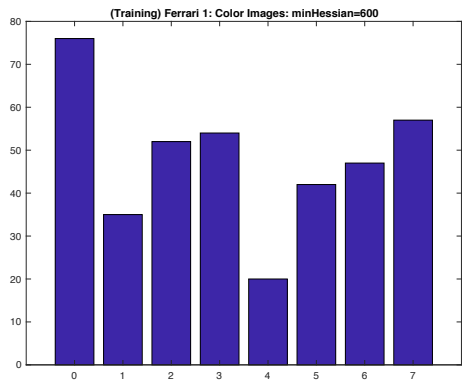


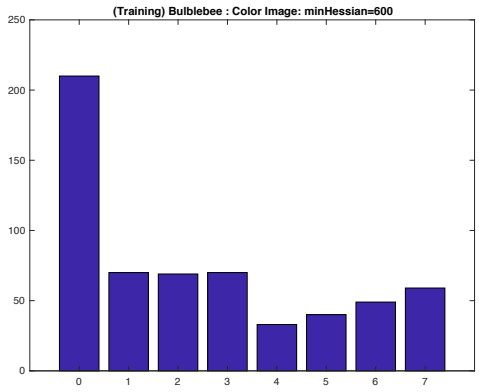


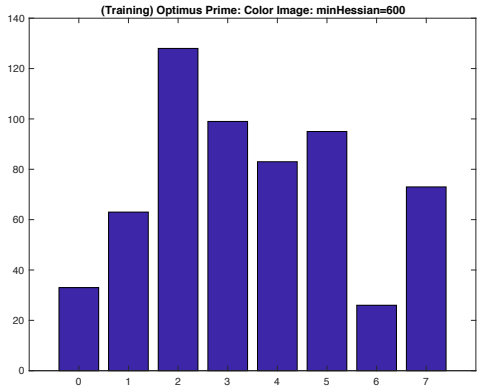


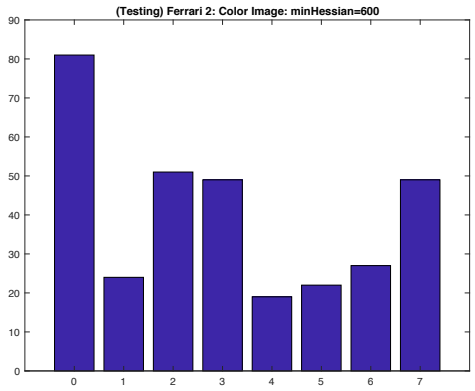


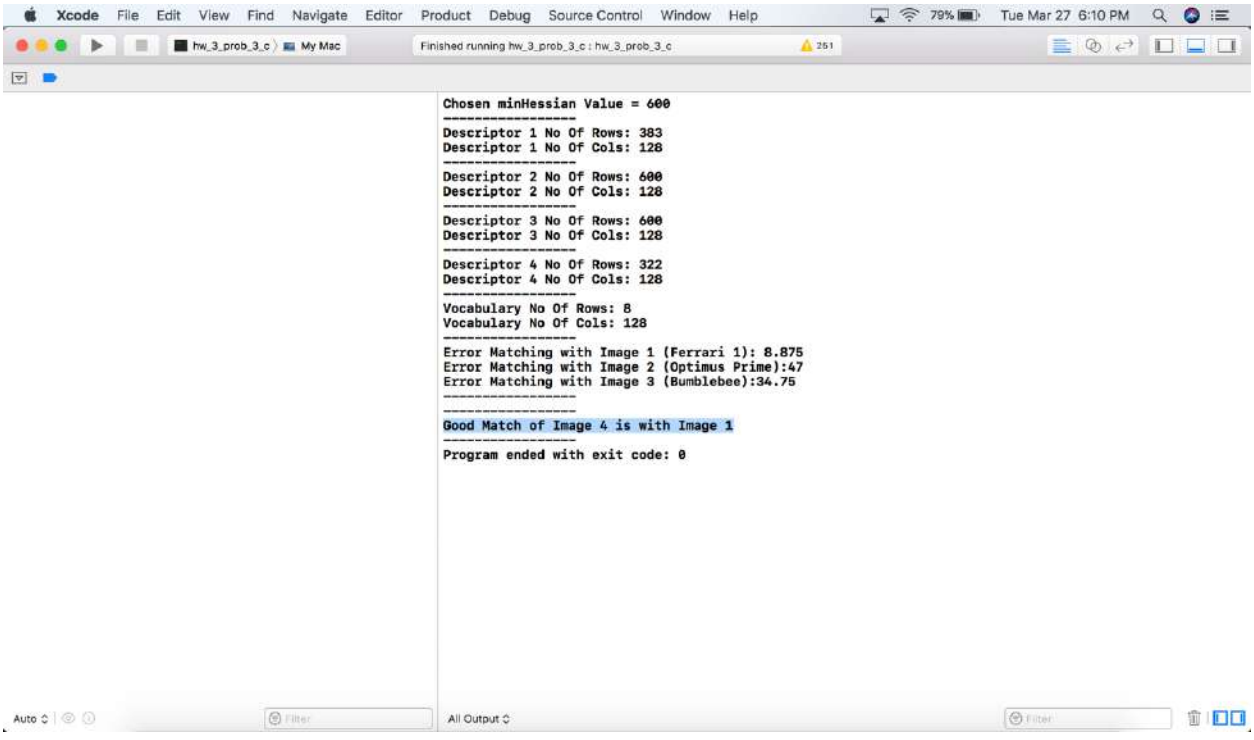
HISTOGRAM MATCHING ON COLOR IMAGES WITH MINHESSIAN=600



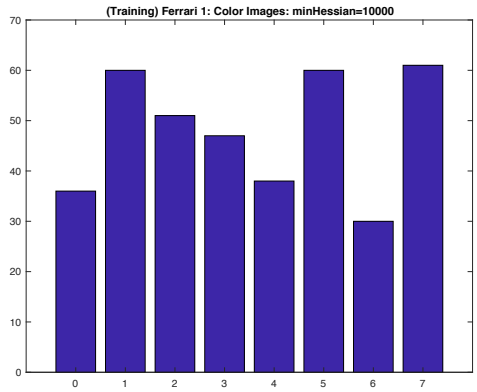


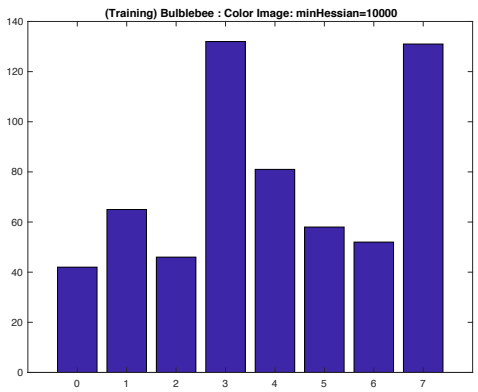


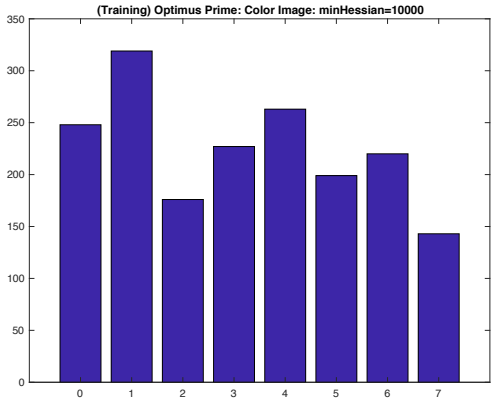


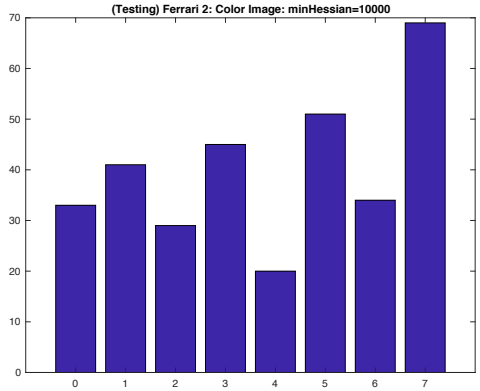


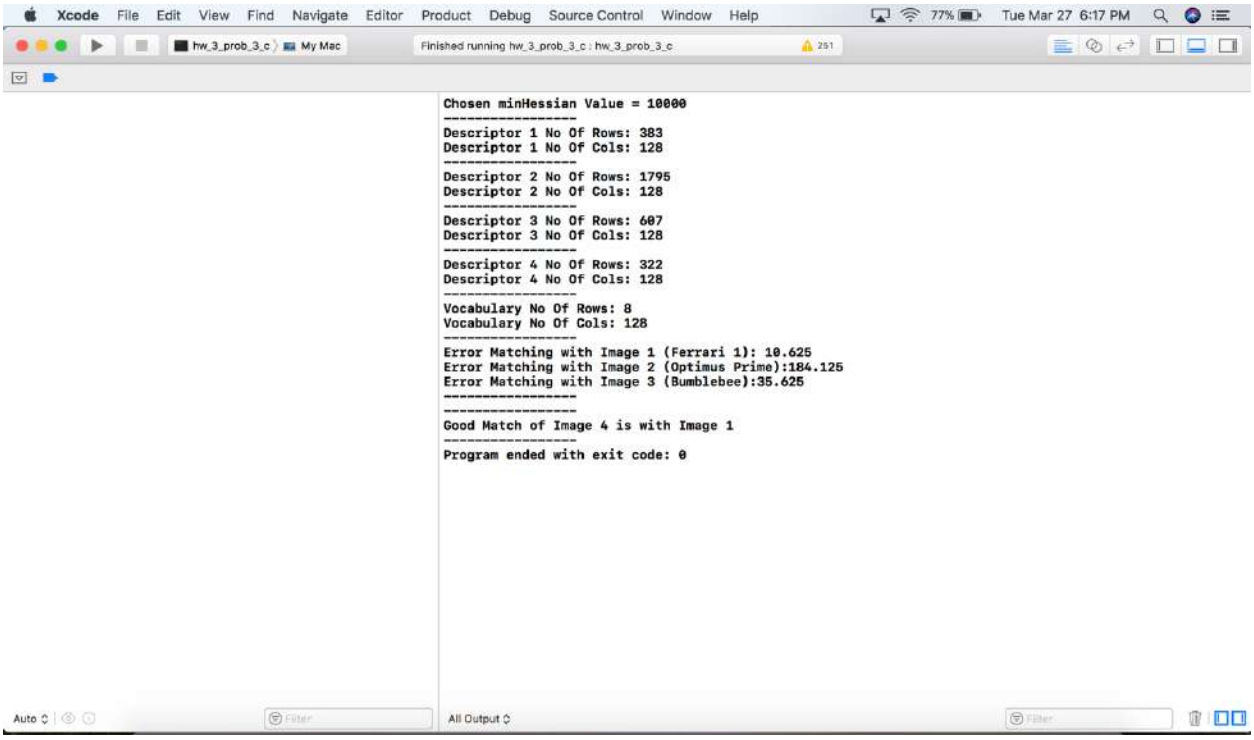
HISTOGRAM MATCHING ON COLOR IMAGES WITH MINHESSIAN=10000



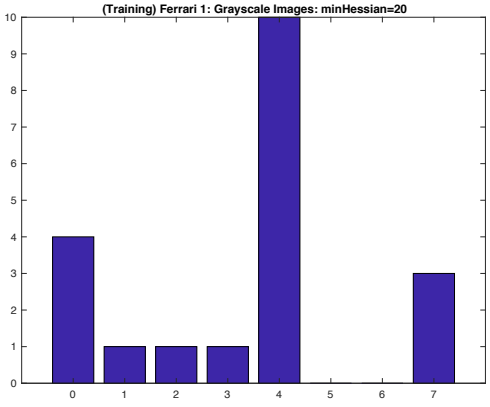


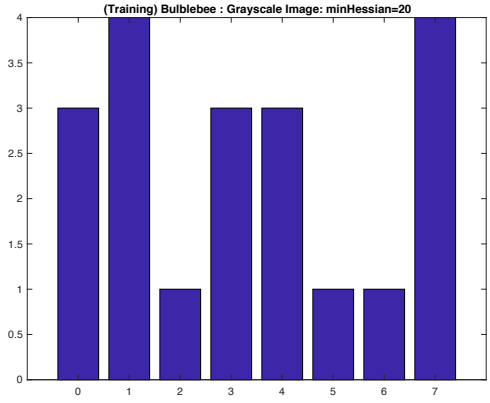


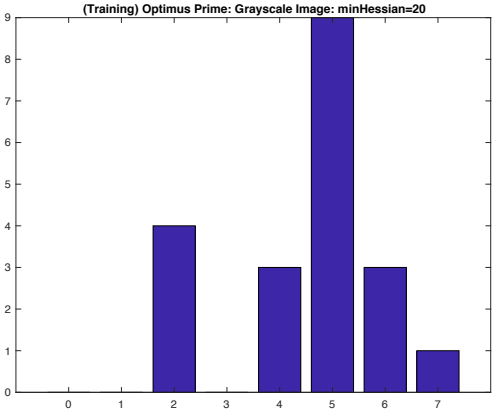


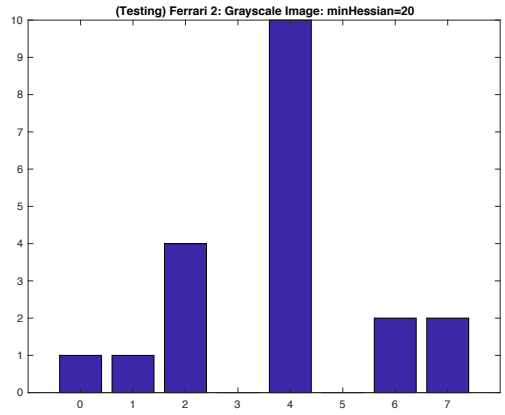


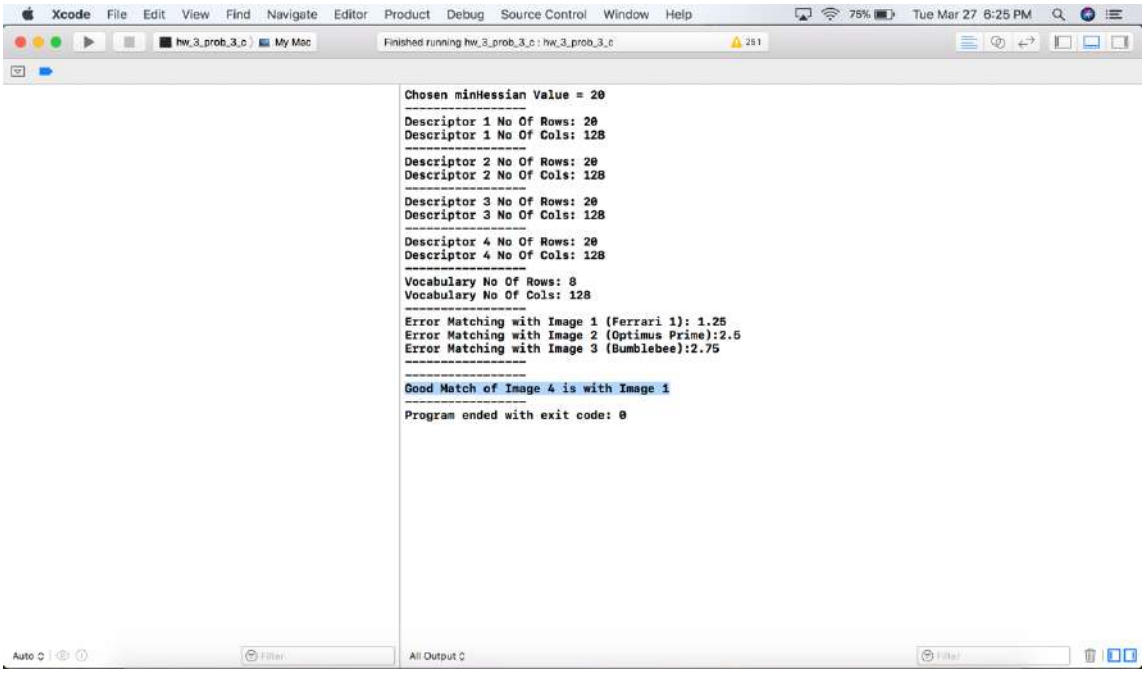
HISTOGRAM MATCHING ON GRAYSCALE IMAGES WITH MINHESSIAN=20



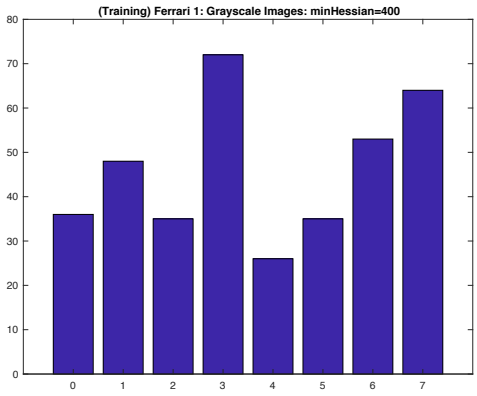


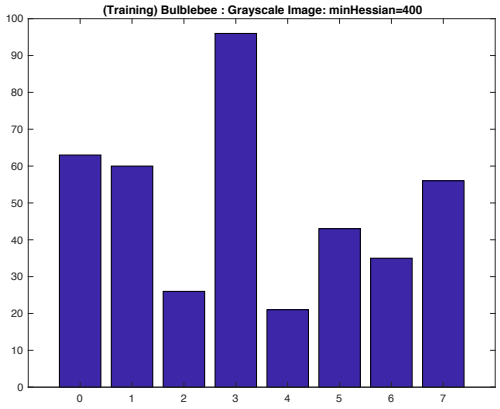


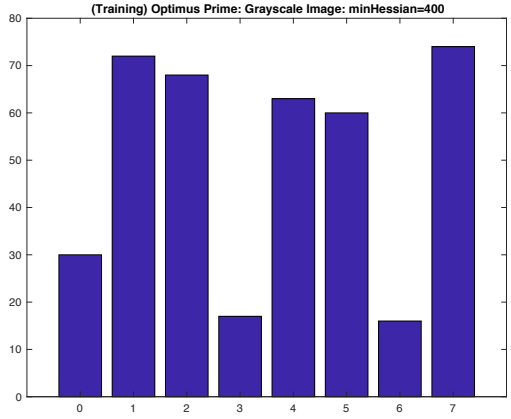


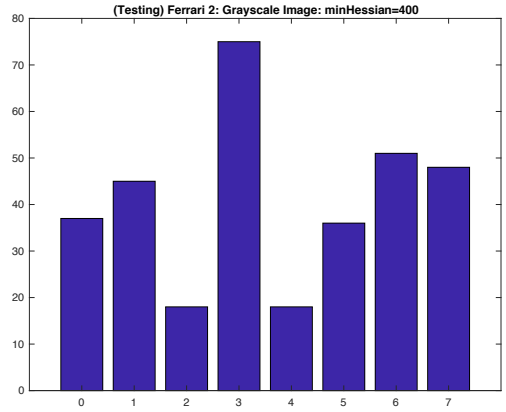


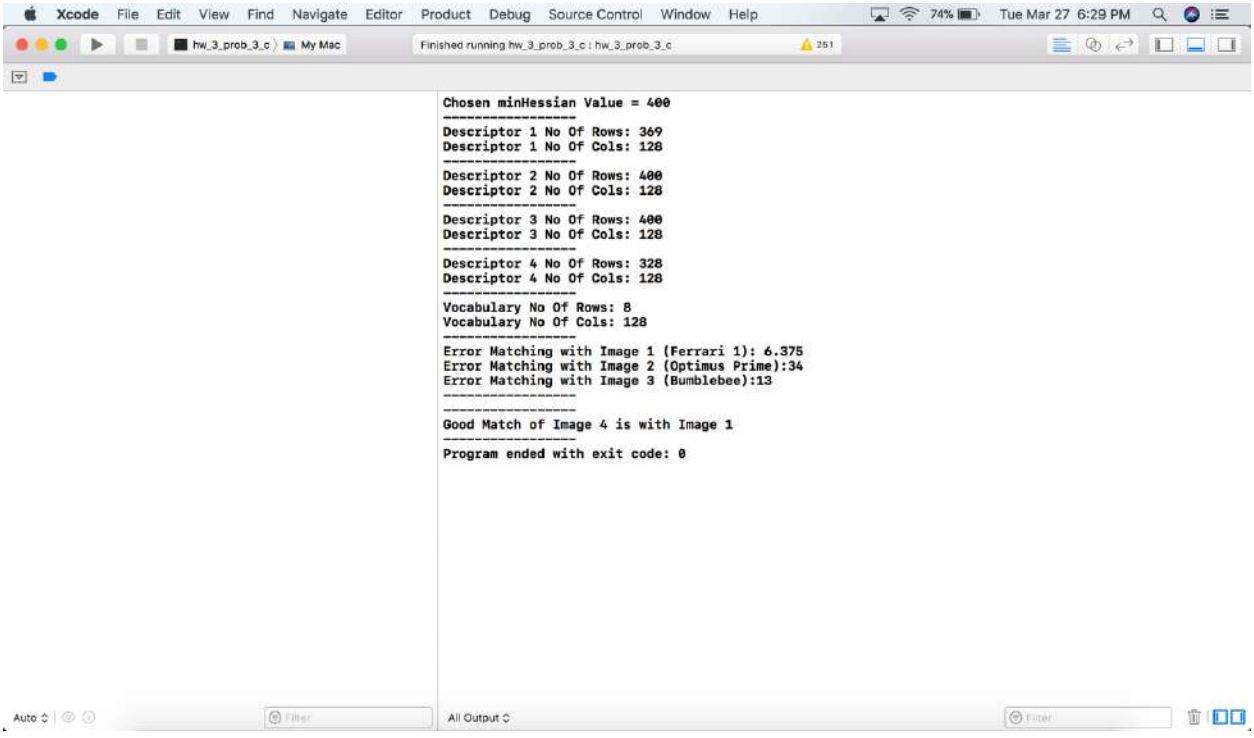
HISTOGRAM MATCHING ON GRAYSCALE IMAGES WITH MINHESSIAN=400



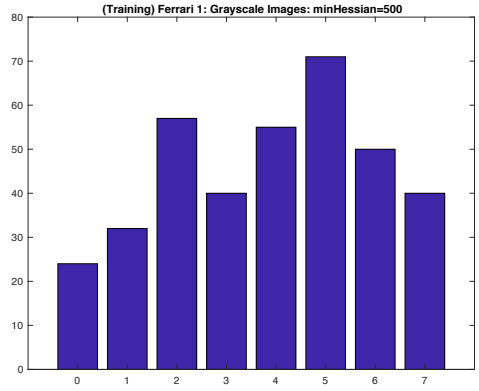


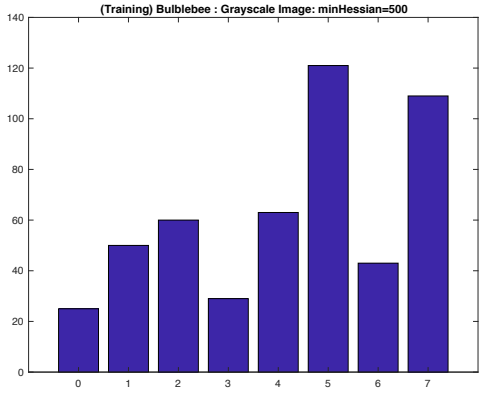


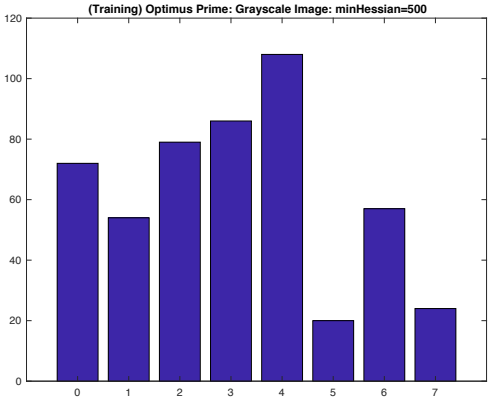


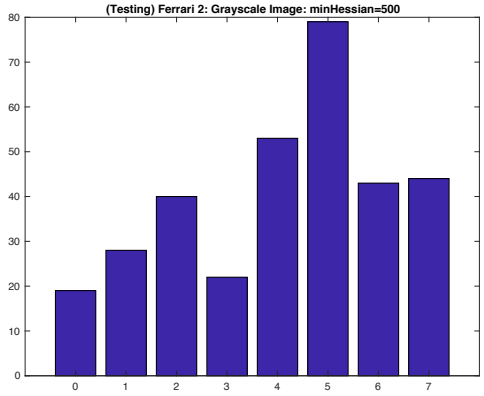


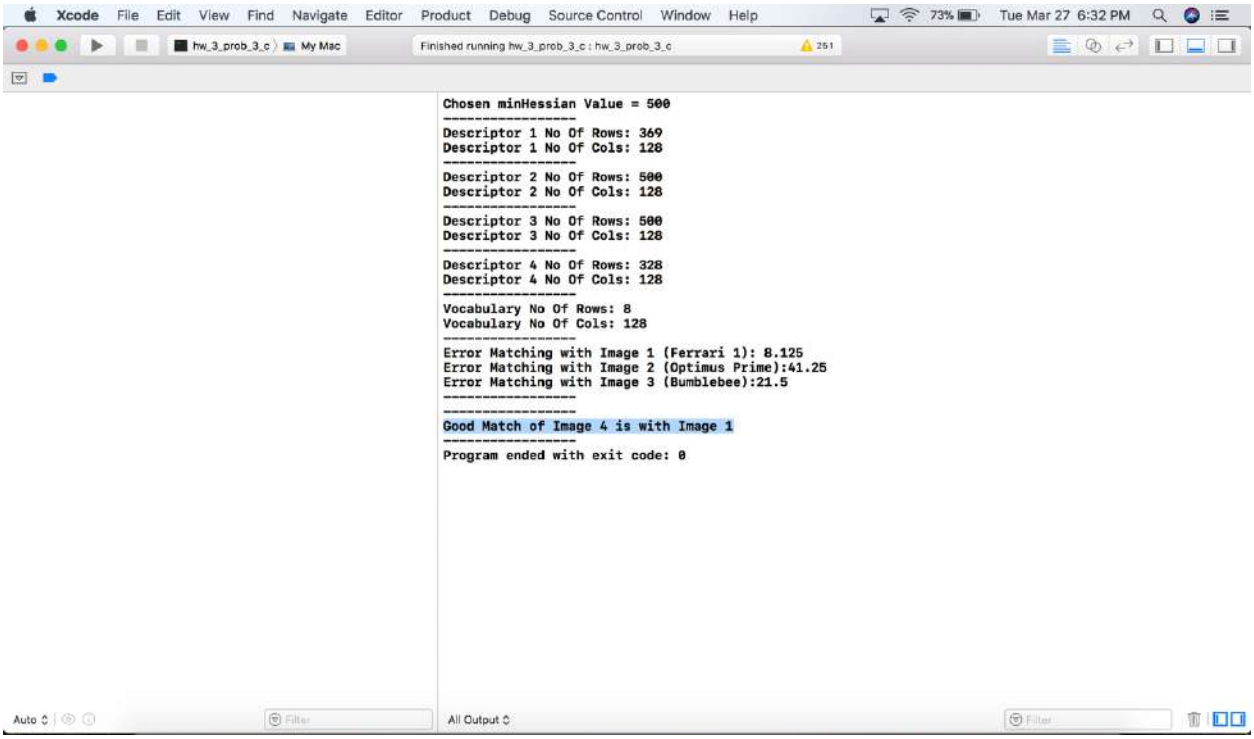
HISTOGRAM MATCHING ON GRAYSCALE IMAGES WITH MINHESSIAN=500



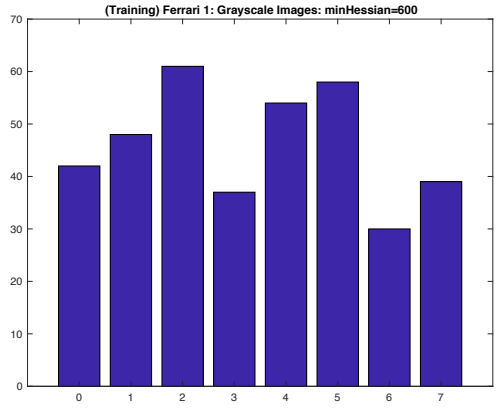


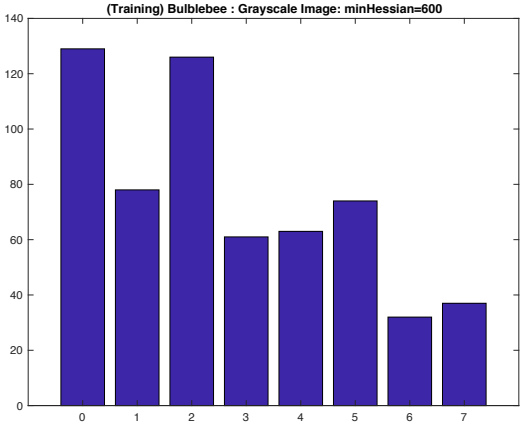


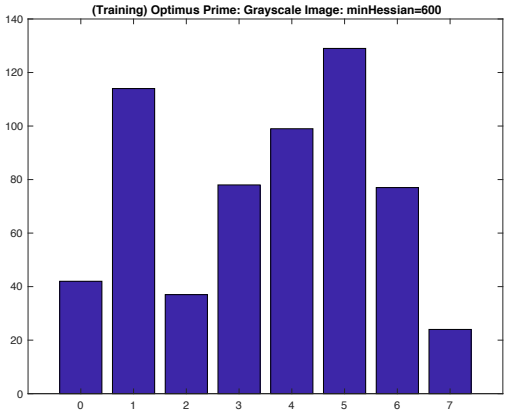


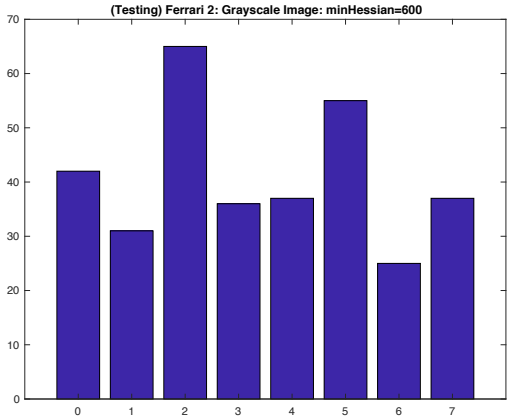


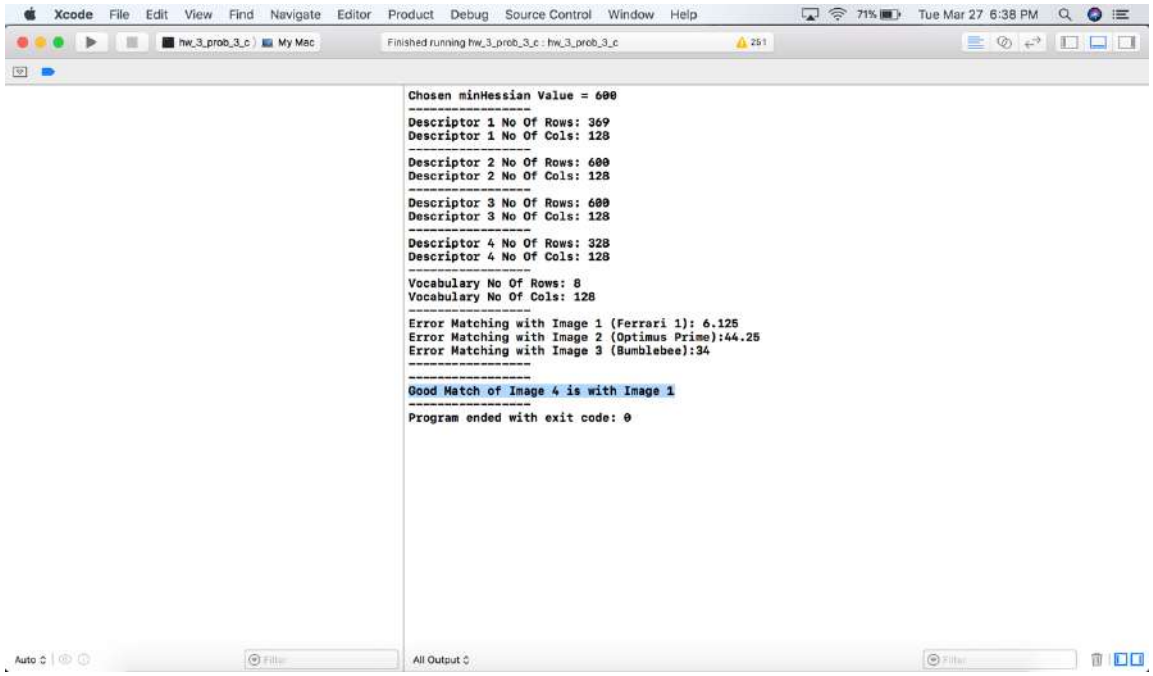
HISTOGRAM MATCHING ON GRAYSCALE IMAGES WITH MINHESSIAN=600



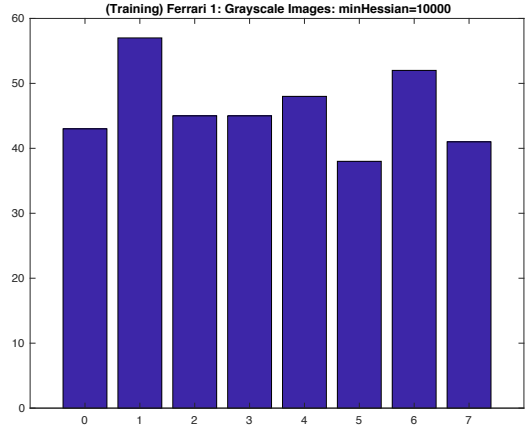


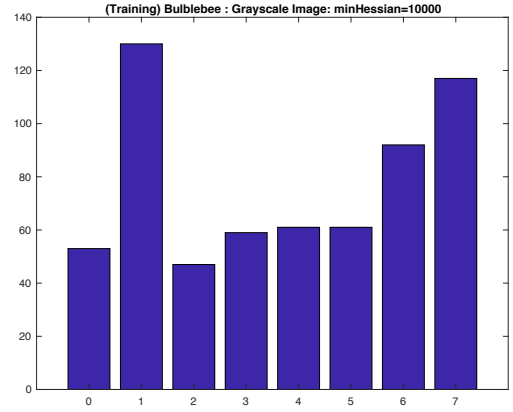


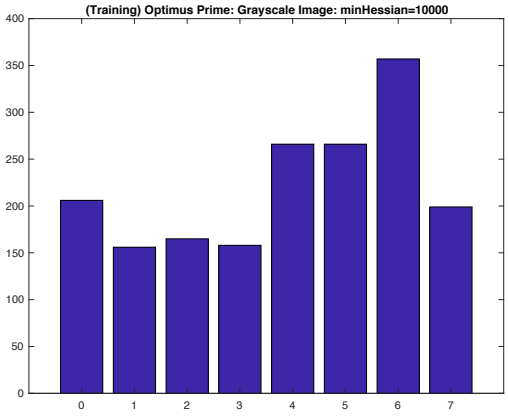


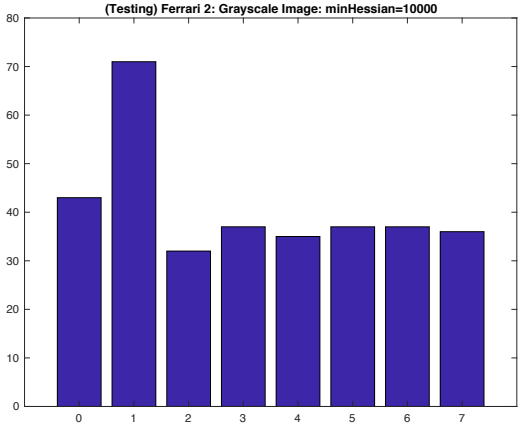


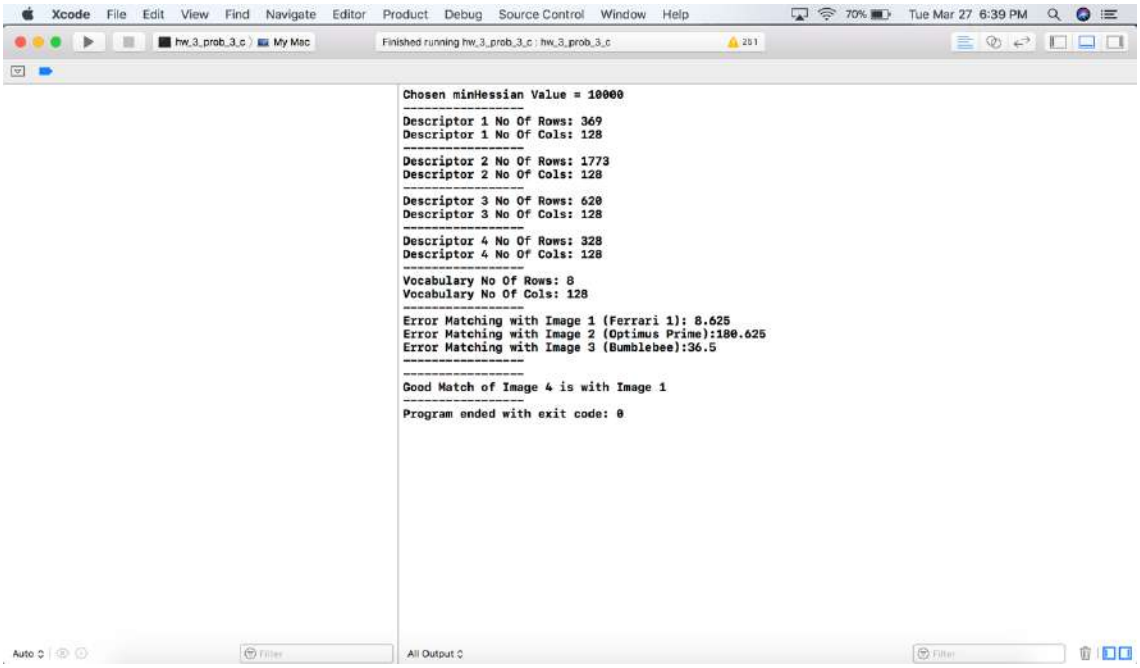
HISTOGRAM MATCHING ON GRAYSCALE IMAGES WITH MINHESSIAN=10000











➢ DISCUSSION:

- The problem was given as an open-ended question. The aim was to use Bag of Words concept to match images.
- There are various suggestions provided in the below link to match Images
  - Comparing Histograms
  - Template Matching
  - Feature Matching

  (Source: https://stackoverflow.com/questions/11541154/checking-images-for-similarity-with-opencv)
- I used Bag of Words concept and Comparing Histograms of code words to match Images
- In this process, I used Ferrari 1, Bumblebee and Optimus Prime as training images and Ferrari 2 as test image. I matched test image Ferrari 2 with the other three images.
- All of the above results show that Image 4 (Ferrari 2) matches with Image 1 (Ferrari 1)
- I performed the above experiments in various ways
  - Color Images / Grayscale Images
    - Min Hessian value = 20, 400, 500, 600, 10000
- All the above experiments show perfectly that Ferrari 1 and Ferrari 2 images match perfectly
- On visual comparison itself, I could see the histogram of code words of Ferrari 1 and Ferrari 2 matched well
- And the histogram matching technique also worked as expected and gave Ferrari 1 and Ferrari 2 as best matched outputs
- Though SIFT feature extraction method is very bad for min Hessian values = 20 and 10000, the image matching technique works the same way as other min Hessian values.
- This proves that irrespective of color/grayscale images and irrespective of min Hessian values, Image Matching technique that is proposed is robust and able to match perfectly
- By close observation Ferrari 2 image is same as Ferrari 1 image except the viewing angle difference. Hence the matching method based on Bag of Words proposed works well irrespective of different views of angle.

- **Thus, the entire problem 3 gave us a good idea of two different feature extraction methods: SIFT and SURF.**
- **For each part of the question, various trials were made on both color and grayscale images, and it was proved that they don't make any difference.**
- **They were also tried with different min Hessian values = 20, 400, 500, 600, 10000. Also, I realized that 400 to 600 are the ideal values.**
- **Image Matching techniques using FLANN based Matcher and Bag of Words using Histogram Matching were done and proved that Ferrari 1 and Ferrari 2 are best matched images.**

APPENDIX

(DESCRIPTION OF FUNCTIONS USED FROM DIP_MyHeaderFile.h)

fileRead() function:

- Reads the given filename to a 1D array

allocMemory2D() function:

- Allocate memory for a 2D array with the given row and column size and initialize to zero using 2 nested for loops
- Returns image2D

allocMemory3D() function:

- Allocate memory for a 3D array with the given row and column size and initialize to zero using 3 nested for loops
- Returns image3D

Image1Dto2D() function:

- Converts given 1D image to the output 2D image (grayscale – single channel)
- Returns image2D

Image1Dto3D() function:

- Converts given 1D image to the output 3D image (reads RGBRGBRGBRGB..)
- Returns image3D

Image2Dto1D() function:

- Converts given 2D image to the output 1D array

Image3Dto1D() function:

- Converts given 3D image to the output 1D array

seperateChannels() function:

- Separates the given 3D image to 2D array based on the index 0 – Red, 1 – Green , 2 – Blue
- Return the 2D image

combineChannels() function:

- Combines given three 2D image arrays to 3D array
- Returns image3D

fileWrite() function:

- Write the given 1D array of unsigned char to the given filename

fileWriteHist() function

- Write the given 1D array of unsigned integers to the given filename

freeMemory2D () function

- Free memory of every allocated array in the 2D array using delete []

freeMemory3D () function:

- Free memory of every allocated array in the 3D array using delete []

Histogram2DImage () function:

- Traverse through the given 2D array using 2 nested for loops
    - Get the pixel intensity value at the location
    - Increment the histogram array based on the pixel obtained