EE 569: Digital Image Processing
Homework #2

By
Brinal Chimanlal Bheda
USC ID - 8470655102
bbheda@usc.edu
Spring 2019

Issue date: 01/20/2019
Submission Date: 02/12/2019

Problem 1: Edge Detection

(a) Sobel Edge Detector

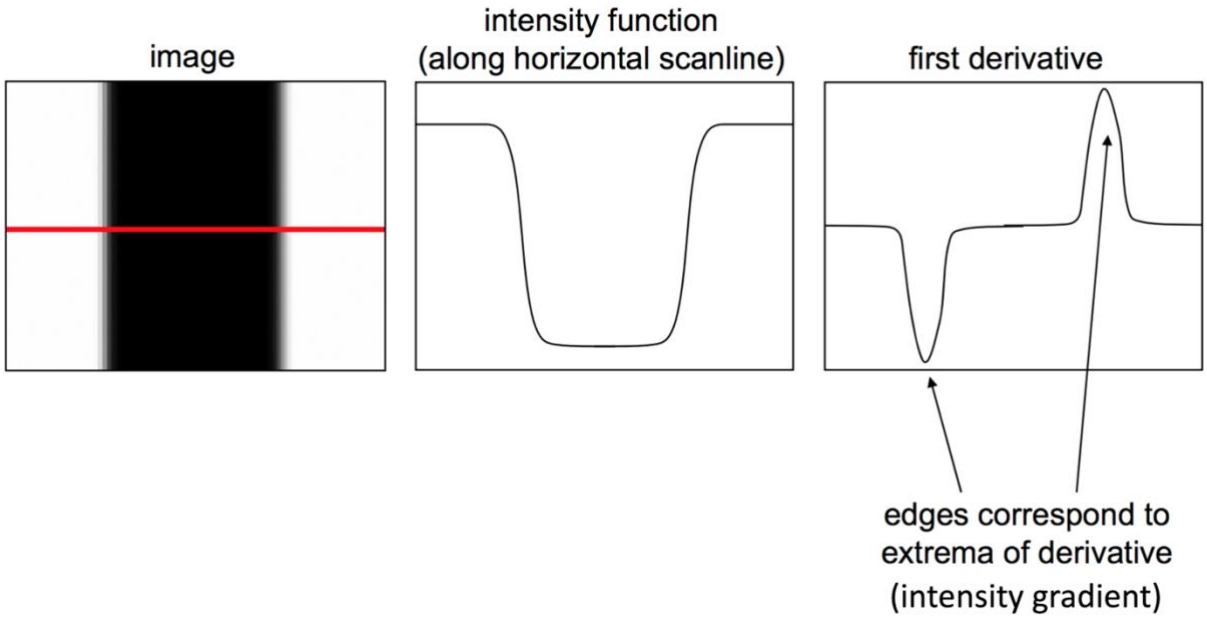
I. Abstract and Motivation

Edge Detection is widely used in Image Processing and Computer Vision algorithms. Edge detection is determining the boundaries of objects in a given image. They determine the structure of the image such that when a human looks at it, it recognizes the objects or features of the image at the first instance. Although, image processing algorithms have developed a lot over the years, edge detection is still a vital research area. Edge detection using Convolutional Neural Network is being used in a lot of applications. Edges in the images are the areas where there is a sudden change of intensities and hence it is said to be high frequency.

Edge detection is used to reduce unnecessary information and preserve the structure of the picture. It is mainly used in detecting the changes in the intensity or brightness in an image, which can be determined by using the gradient. It is mainly used to extract important features like corners, lines and curves. That's why it is widely used in the object recognition, object tracking, segmentation, image retrieval and scene parsing. I have used 3 edge detectors in my homework – Sobel, Canny, Structured Edge. I have compared the edge detection algorithm results from the 3 methods and evaluated the performance for each.

(Source: http://www.cs.utoronto.ca/~fidler/slides/CSC420/lecture5.pdf
https://www.slideshare.net/ishraqabd/edge-detection
http://www.cs.unc.edu/~lazebnik/spring09/lec06_edge.pdf)

Sobel edge detector assumes that the intensity of the image jumps a lot at the edge. Example is shown below,



The second image represents the intensity function. The third image corresponds to the first derivation also called as intensity gradient. We can see in the picture above that at the extrema of the derivative, we can consider it as the edge in the image.

The definition of gradient, it's magnitude and orientation are given below,

Find edge by checking gradient

Gradient

$$\nabla f = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

Magnitude

$$|\nabla f| = \sqrt{g_y^2 + g_x^2}$$

Orientation

$$\theta = \tan^{-1}\left[\frac{g_y}{g_x}\right]$$

We need to use discrete method while doing computations on computer.

Approach

And so we use the 3x3 Sobel mask. There are 2 types of mask – first one is used to calculate the x component of gradient and the second one is used to calculate the y gradient. So when we do the convolution by such masks, use the element in the mask to multiply with the pixel one by one and then add all the products together. And that is the value for the center pixel. Compute the finite difference at the center pixel by convolution.

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

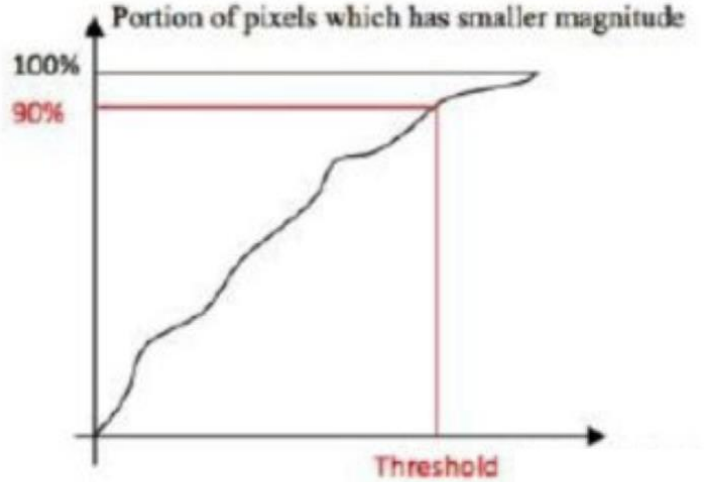
Gx

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

Gy

Now, we normalize the results for the output because we do not have control on the gradient components as they may fall out of range 0 to 255. We obtain 2 gray images – x component and y component image for the original image. And we observe that the x component image is able to capture all the vertical lines distinctly. And the y component image captures the horizontal features. The final image shows the magnitude of all the gradients.

After we get the magnitude map, we have to set the threshold such that below that percentage the gradient response is too weak but above that it is strong enough to help find the edge pixels. We need to find the cumulative distribution of gradient magnitude vs portion of pixels which has smaller magnitude. For example, if we set the threshold to be 85%, it may show extra dots, such that there is noise. But if we increase the threshold, say 90%, the image appears clear.



Thus, we can say that the Sobel detector is very sensitive to noise. This is a drawback.

II. Approach and Procedure

Convert RGB images to grey-level image first. This is done by using the luminosity method. The formula to convert a colored image to gray scale image is given below,

Gray scale image = 0.21*(Red channel) + 0.72*(G channel) + 0.07*(B channel)

Find max and min value obtained for magnitude of gradient. Normalize the x-gradient and the y-gradient values to 0-255. Find CDF of magnitude to find threshold with desired probability and using this threshold, binarize the output edge image. Tune the thresholds (in terms of percentage) to obtain your best edge map. An edge map is a binary image whose pixel values are either 0 (edge) or 255 (background). Sobel edge detector is more centered on regions with high spatial frequency that corresponds to the edges.

Differentiate the pixel values along X and Y direction using 3x3 Sobel Edge Masks. There are two filters used to calculate the gradient of the image in the horizontal and vertical direction respectively. From X and Y gradient values, the magnitude and direction of the pixel gradient is found. Gy is the rotation of Gx by 90 degree.

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

Gx

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

Gy

Find edge by checking gradient

Gradient

$$\nabla f = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

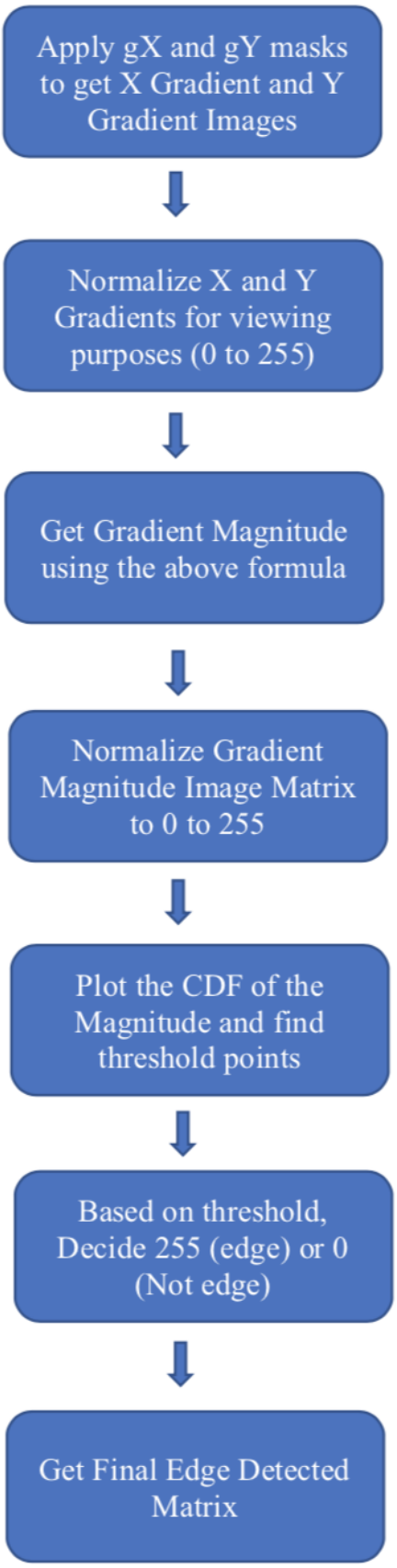
Magnitude

$$|\nabla f| = \sqrt{g_y^2 + g_x^2}$$

Orientation

$$\theta = \tan^{-1}\left[\frac{g_y}{g_x}\right]$$

After finding the magnitude of gradient every pixel, we do thresholding on scaled values of Magnitude between 0 to 255 to find edges. The thresholds are chosen by trial and error basis for different percentages depending on the Cumulative Histogram plot. Mostly, the threshold is chosen in such a way that 90% pixels of the cumulative probability graph are considered as edges.


Apply gX and gY masks to get X Gradient and Y Gradient Images
Normalize X and Y Gradients for viewing purposes (0 to 255)
Get Gradient Magnitude using the above formula
Normalize Gradient Magnitude Image Matrix to 0 to 255
Plot the CDF of the Magnitude and find threshold points
Based on threshold, Decide 255 (edge) or 0 (Not edge)
Get Final Edge Detected Matrix

Sobel Edge Detector Flow Chart

Algorithm implemented in C++ :

- Read the input image “tiger.raw” and “pig.raw” whose dimensions are height_size, width_size, BytesPerPixel
- Convert the RGB color image into a gray scale image by using the luminosity formula mentioned above
- Apply the Gx gradient and Gy gradient masks on each pixel value and find the magnitude of the image by using the magnitude formula shown above using Gx and Gy
- Since the values obtained for the magnitude, Gx gradient and Gy gradient are very high, we normalize the value in the range of 0 to 255. Thus, the minimum and maximum values for the gradient are calculated
- Normalization is done using the formula:

  Normalized image = (input image – minimum value) / (maximum value – minimum value)
- Find the Cumulative Distribution function (cdf) of the magnitude with increasing order of intensities
- Take the threshold as 90% of the edge value and 10% of the pixels in the cdf plot.
- The intensity value corresponding to the pixel threshold is calculated:

  Output image = 0 if input pixel value > threshold
  = 255 if input pixel value < threshold
- Combine the thresholded output image
- Write the computed image data array on output.raw file using the fwrite() function

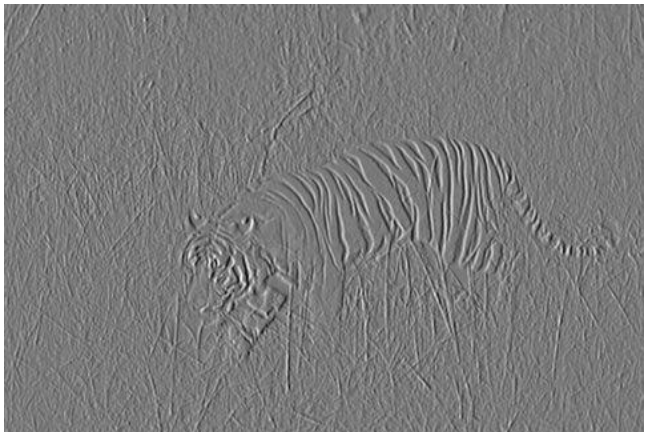
III. Experimental Results

Input image tiger.raw

Input image pig.raw

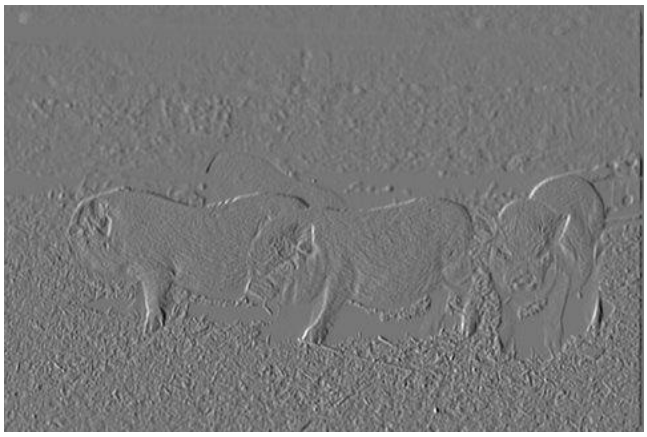
tiger.raw gray scale image

pig.raw gray scale image

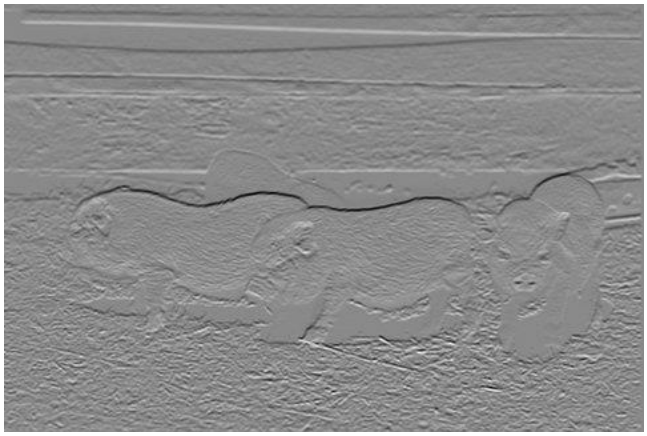


Normalized Gx gradient image showing vertical edges

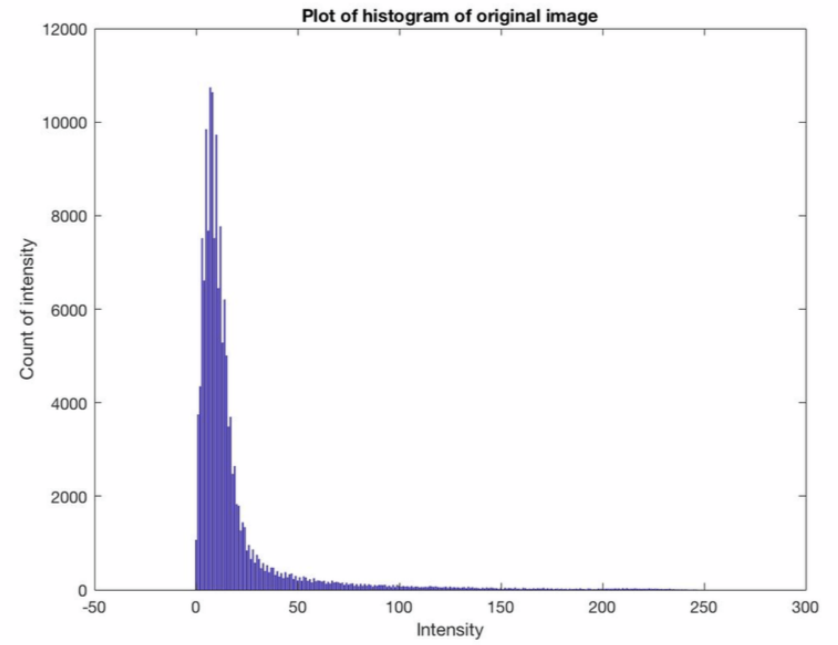
Normalized Gy gradient image showing horizontal edges



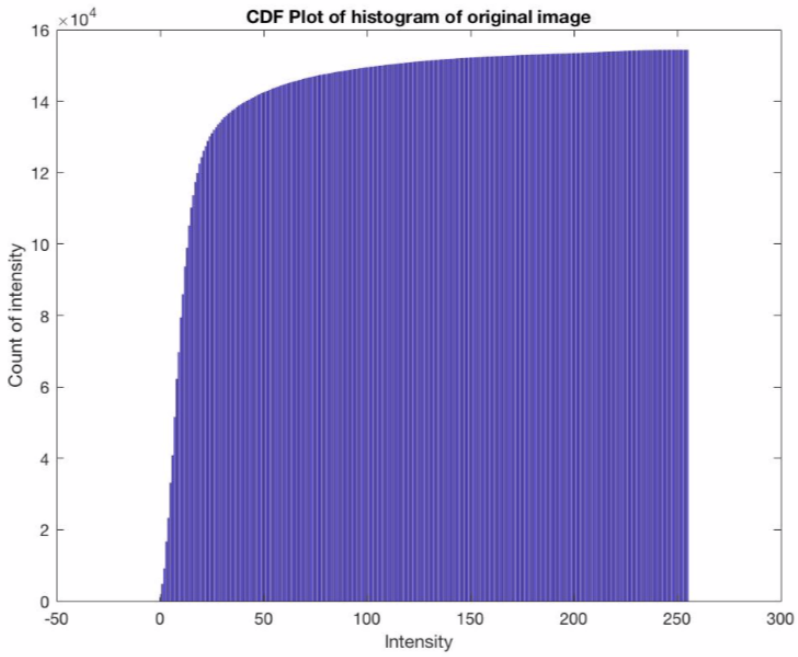
Normalized Gx gradient image showing vertical edges



Normalized Gy gradient image showing horizontal edges

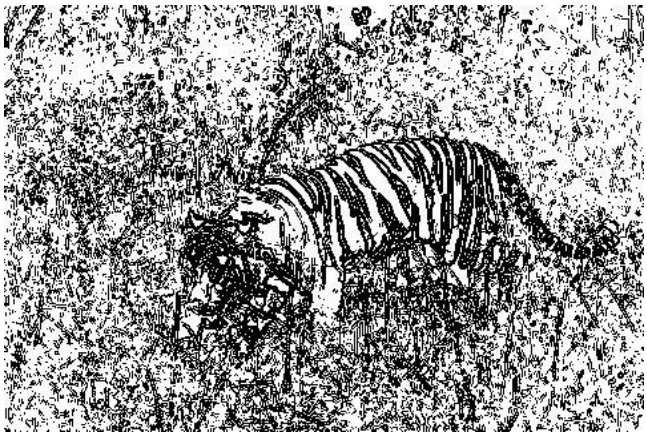


Histogram of intensity values for Sobel Detector

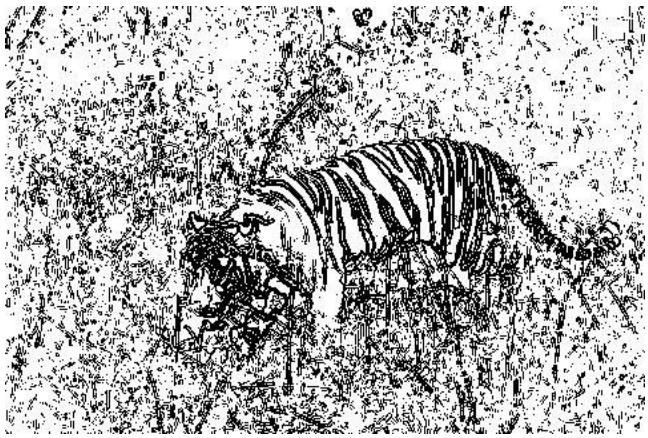


Cumulative plot of intensity values for Sobel Detector

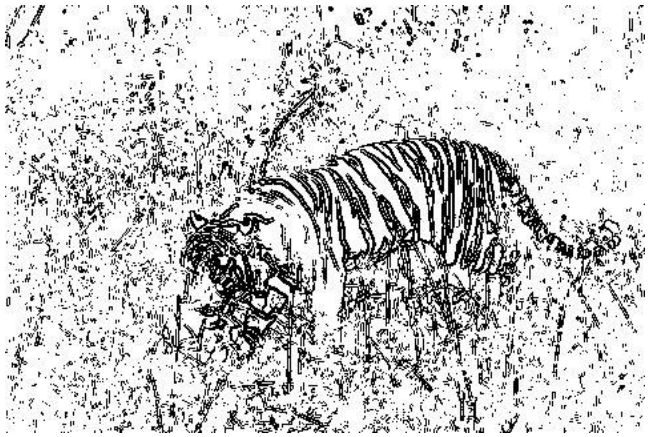
Applying Sobel Edge Detector using different thresholds:



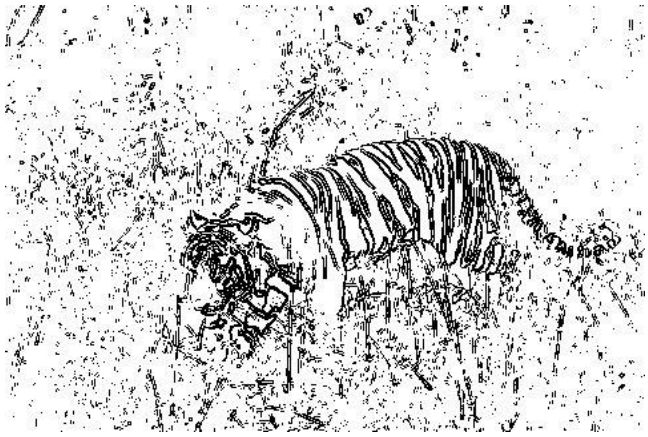
Edge detection using 60%



Edge detection using 70%

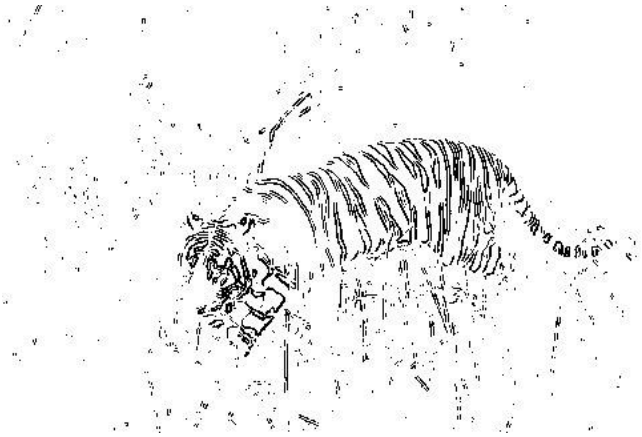


Edge detection using 80%

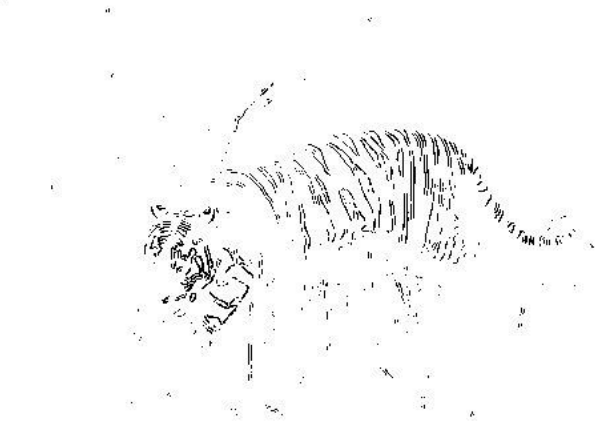


Edge detection using 85%

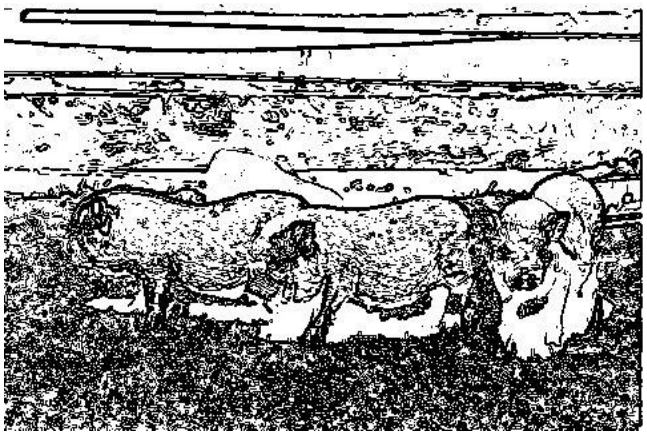
Edge detection using 90%



Edge detection using 95%

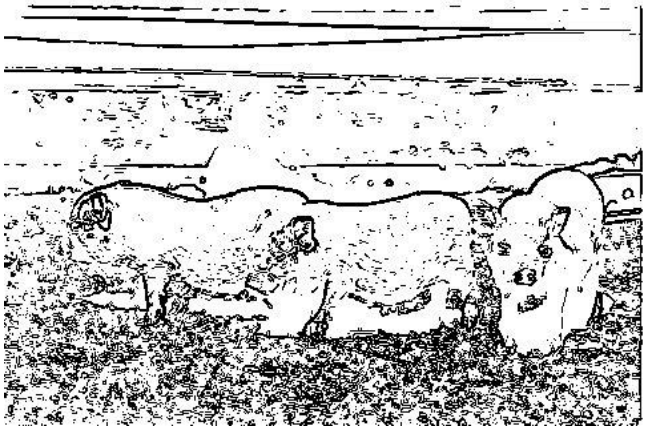


Edge detection using 98%

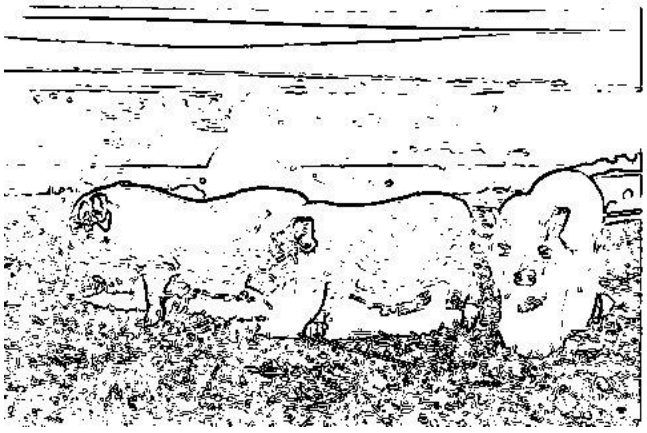


Edge detection using 60%

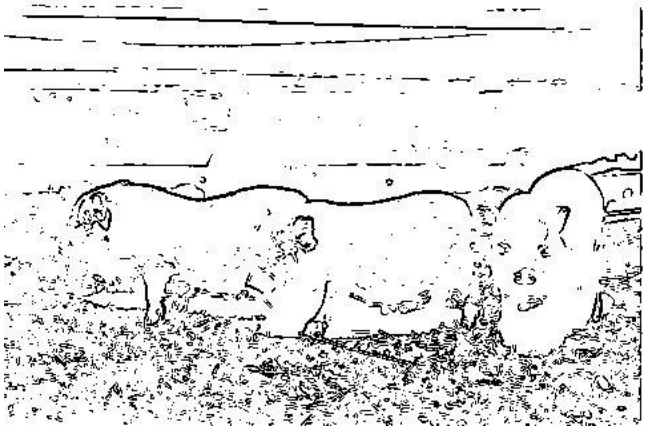
Edge detection using 70%



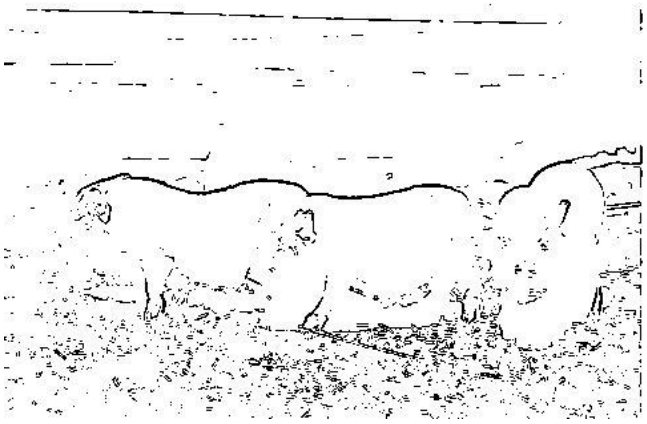
Edge detection using 80%



Edge detection using 85%



Edge detection using 90%



Edge detection using 95%

Edge detection using 98%

IV. Discussion

The Sobel filter is used to calculate the change of gradient in x and y directions. The absolute magnitude was obtained similarly. The cumulative plot helps in deciding the threshold intensity value used to make the edge map. the Sobel detector has one knee point that is why one threshold is required to prepare the edge map.

Sobel detector is very sensitive to noise and hence we need noise filtering before applying the Sobel masks. Also, we use the Non-maximum suppression (NMS) after using the Sobel edge detector so as to detect the proper location of edges. This helps improving the overall result.

Sobel detects edge after single derivative and decides edges based on threshold. We tune the threshold to increase or decrease the accuracy of edge detection. Sobel is very sensitive to the threshold we select. Sobel Edge detection was done using different thresholds at 60%, 70%, 80%, 85%, 90%, 95% and 98%. On performing edge detection on both noise free and noisy images, 90% threshold seems to give the best output for both tiger and pig images. 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.

The edge maps for different thresholds for the tiger and pig images are shown above. In the tiger image, we observe that the stripes are shown accurately which help us to recognize the image of the tiger, however, the face does not capture all the edges. In the pig image, we can see that the pig face isn't coming out clear. Also, the two pigs which are in front and back of each other, are very difficult to distinguish in the edge map.

(b) Canny Edge Detector

I. Abstract and Motivation

While Sobel edge detector is quite straight forward, Canny edge detector uses some post processing procedures to help us improve the edge map. It is the most common edge detection algorithm used. We have various edge detection algorithms because till now we have not been able to come up with a single edge detection algorithm for every image.

Canny Edge detection is a multi-stage algorithm developed by John Canny. His main aim was to solve two main criteria in the edge detection field. First, detection of all edges in image with very low error rate. Second, localize edge points such that distance between edges and center of the true edge is minimum. He wanted to ensure that the detector has only single response to an edge. Canny detection achieves more reliability and involves simple computation.

(Source: https://en.wikipedia.org/wiki/Canny_edge_detector)

Canny edge detection is a technique to extract useful structural information from different vision objects and dramatically reduce the amount of data to be processed. It has been widely applied in various computer vision systems. Canny has found that the requirements for the application of edge detection on diverse vision systems are relatively similar. Thus, an edge detection solution to address these requirements can be implemented in a wide range of situations. The general criteria for edge detection include:

1. Detection of edge with low error rate, which means that the detection should accurately catch as many edges shown in the image as possible
2. The edge point detected from the operator should accurately localize on the center of the edge.
3. A given edge in the image should only be marked once, and where possible, image noise should not create false edges.

To satisfy these requirements Canny used the calculus of variations – a technique which finds the function which optimizes a given functional. The optimal function in Canny's detector is described by the sum of four exponential terms, but it can be approximated by the first derivative of a Gaussian.

Among the edge detection methods developed so far, Canny edge detection algorithm is one of the most strictly defined methods that provides good and reliable detection. Owing to its optimality to meet with the three criteria for edge detection and the simplicity of process for implementation, it became one of the most popular algorithms for edge detection.

The procedure is as follows:

- Filter image with derivative of Gaussian
- Find magnitude and orientation of gradient
- Non-maximum Suppression (NMS)
- Double Thresholding (Hysteresis Thresholding)

(Source: https://www.mathworks.com/help/images/ref/edge.html)

(Source: https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html)

Non-maximum suppression is an edge thinning technique.
Non-maximum suppression is applied to find "the largest" edge. After applying gradient calculation, the edge extracted from the gradient value is still quite blurred. With respect to criterion 3, there should only be one accurate response to the edge. Thus, non-maximum suppression can help to suppress all the gradient values (by setting them to 0) except the local maxima, which indicate locations with the sharpest change of intensity value. The algorithm for each pixel in the gradient image is:

1. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.
2. If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (i.e., the pixel that is pointing in the y-direction, it will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.

Double Threshold:
After application of non-maximum suppression, remaining edge pixels provide a more accurate representation of real edges in an image. However, some edge pixels remain that are caused by noise and color variation. In order to account for these spurious responses, it is essential to filter out edge pixels with a weak gradient value

and preserve edge pixels with a high gradient value. This is accomplished by selecting high and low threshold values. If an edge pixel's gradient value is higher than the high threshold value, it is marked as a strong edge pixel. If an edge pixel's gradient value is smaller than the high threshold value and larger than the low threshold value, it is marked as a weak edge pixel. If an edge pixel's value is smaller than the low threshold value, it will be suppressed. The two threshold values are empirically determined, and their definition will depend on the content of a given input image.

Edge Tracking by Hysteresis:
So far, the strong edge pixels should certainly be involved in the final edge image, as they are extracted from the true edges in the image. However, there will be some debate on the weak edge pixels, as these pixels can either be extracted from the true edge, or the noise/color variations. To achieve an accurate result, the weak edges caused by the latter reasons should be removed. Usually a weak edge pixel caused from true edges will be connected to a strong edge pixel while noise responses are unconnected. To track the edge connection, blob analysis is applied by looking at a weak edge pixel and its 8-connected neighborhood pixels. As long as there is one strong edge pixel that is involved in the blob, that weak edge point can be identified as one that should be preserved.

The Canny edge detector is an edge detection technique utilizing image's intensity gradients and non- maximum suppression with double thresholding. In this part, we applied the Canny edge detector to both *Tiger* and *Pig* images. Generated edge maps by trying different low and high thresholds. The threshold we use below is similar to the high threshold.
Used an online source code such as the Canny edge detector in the MATLAB image processing toolbox using the edge function and the OpenCV (i.e. Open Source Computer Vision Library) using the canny function in C++.

II. Approach and Procedure

Canny Edge detector helps in finding different range of edges in an image and it is a multiple-stage algorithm. In this first we apply Gaussian filter to remove the noise and then we perform Sobel edge detection followed by Non- maximum suppression. We have two thresholds in Canny edge detection – high and low thresholds. By choosing 2 thresholds, we have the better option of choosing the edge points. We do it as follows. If the point is above the maximum threshold, we classify it as edge point. If it is less than the minimum threshold, we classify it as non-edge point. If it

is between the two and if it is connected to edge point, then we classify it as edge point otherwise it is classified as non- edge point.

The Process of Canny edge detection algorithm can be broken down to 5 different steps:

1. Apply Gaussian filter to smooth the image in order to remove the noise
2. Find the intensity gradients of the image
3. Apply non-maximum suppression to get rid of spurious response to edge detection
4. Apply double threshold to determine potential edges
5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges.

Algorithm implemented in Matlab:

- Open the file where the input image is stored by providing the path name, and read the input .jpg image by using imread()
- Convert the rgb image into the gray scale image
- Apply the canny edge detector using the edge() function
- Edge() function inputs the parameters gray scale input image, name of the edge detector being used i.e. Canny, threshold applied to the image to display the edge map i.e. 0.05, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.70, 0.80
- Display the output of the image showing the edge map using imshow()
- Write the output in the Canny output image and store it in the same folder naming output_canny for different thresholds using imwrite()

III. Experimental Results

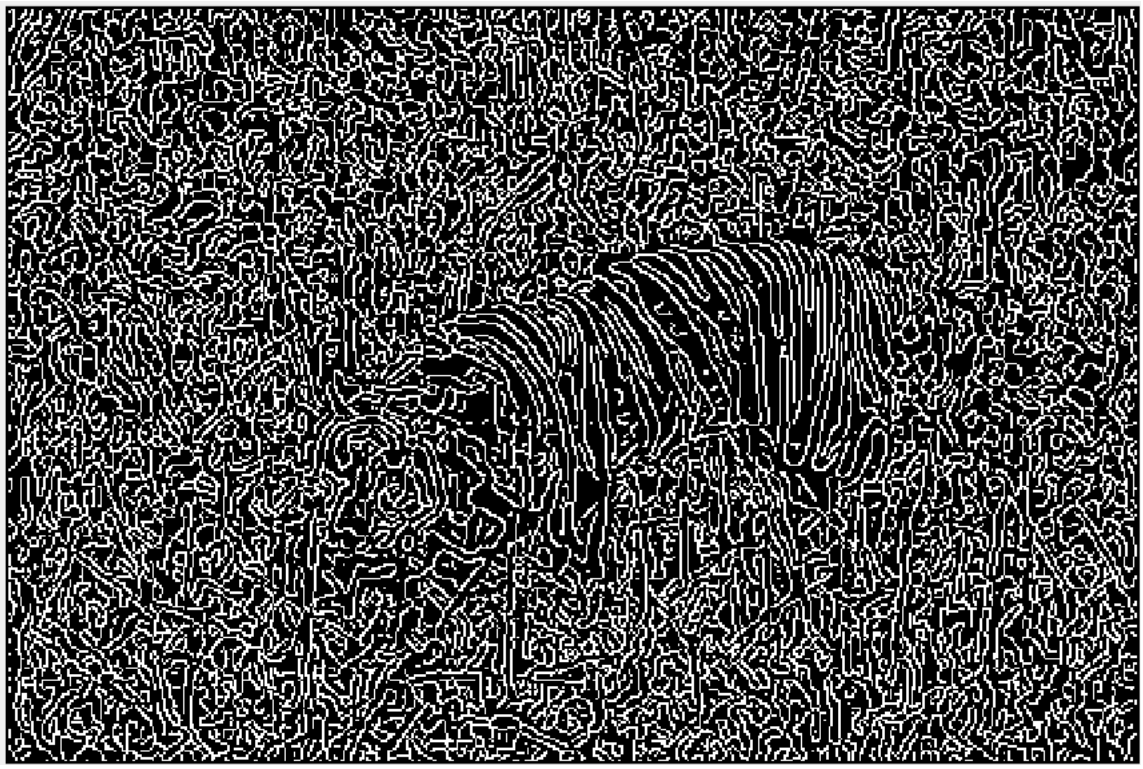
Input image tiger.raw

Input image pig.raw

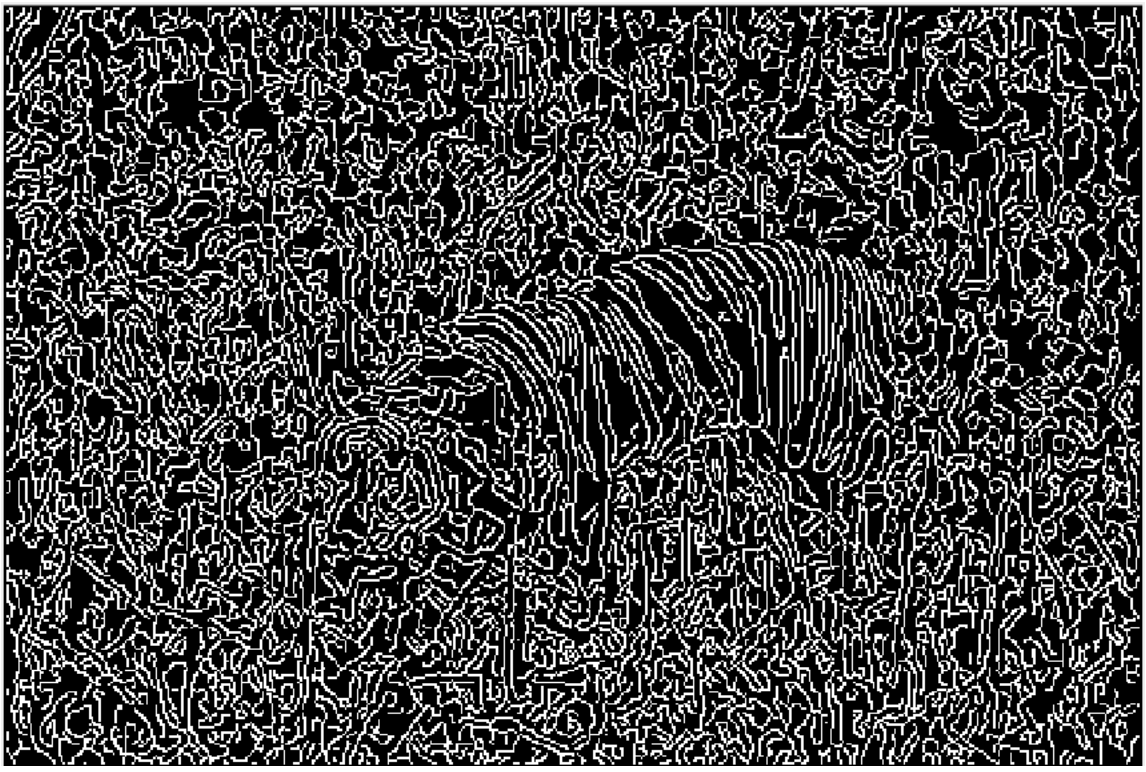
tiger.raw gray scale image

pig.raw gray scale image

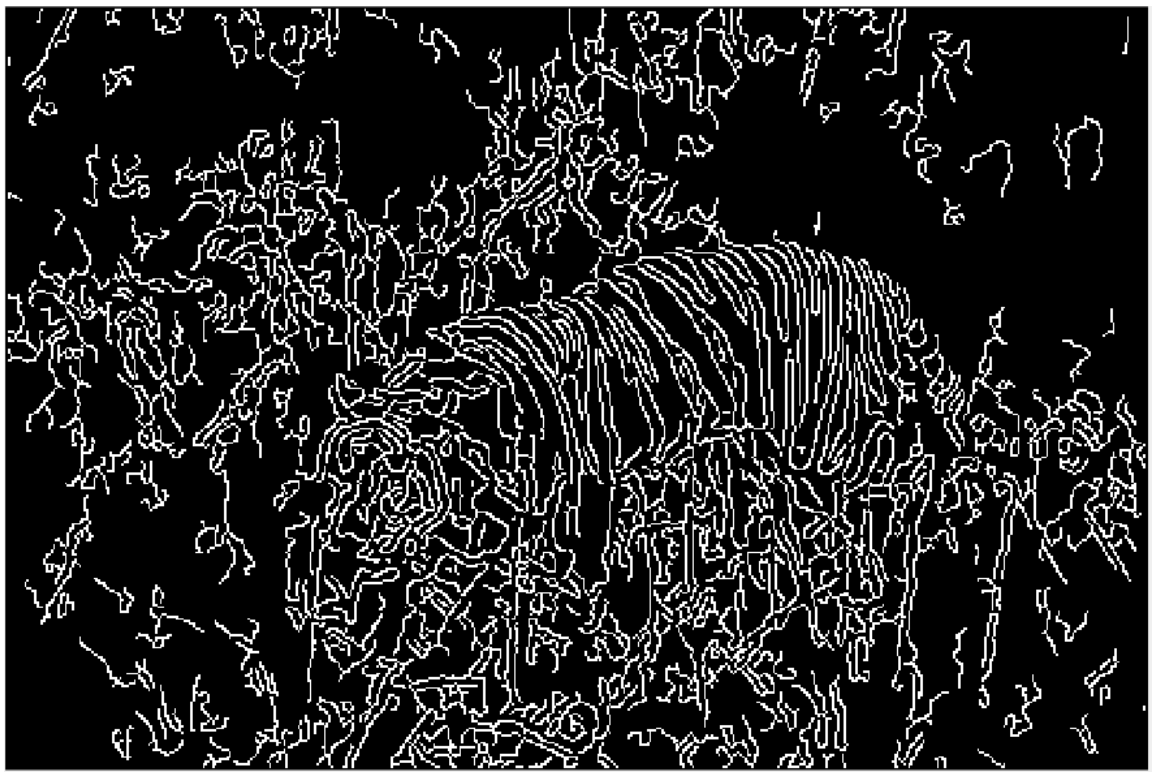
Applying Canny Edge Detector using different thresholds:



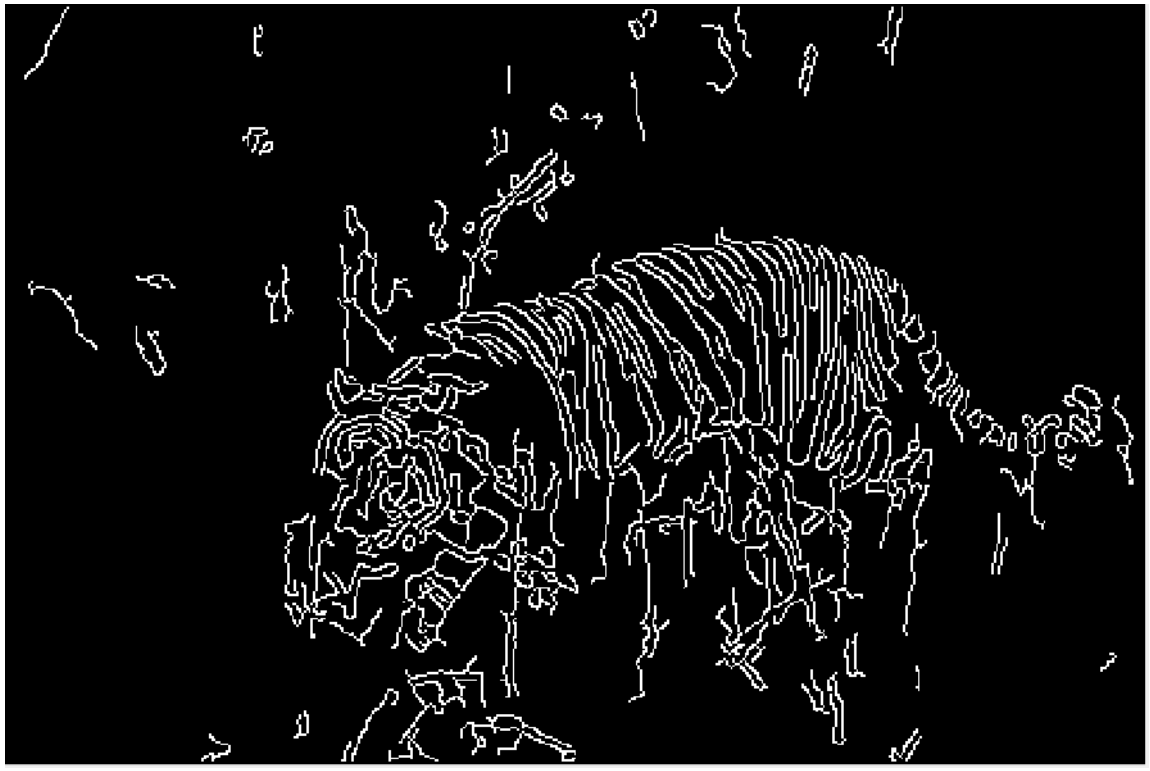
Edge detection using threshold as 0.05



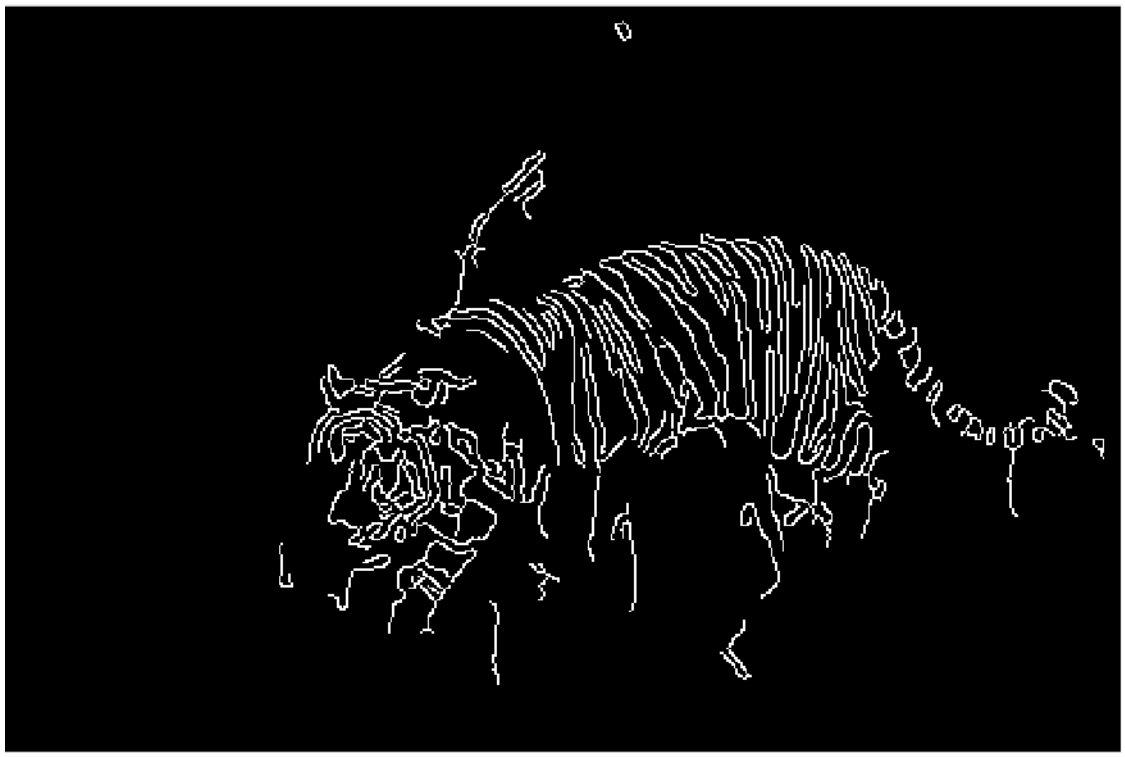
Edge detection using threshold as 0.10



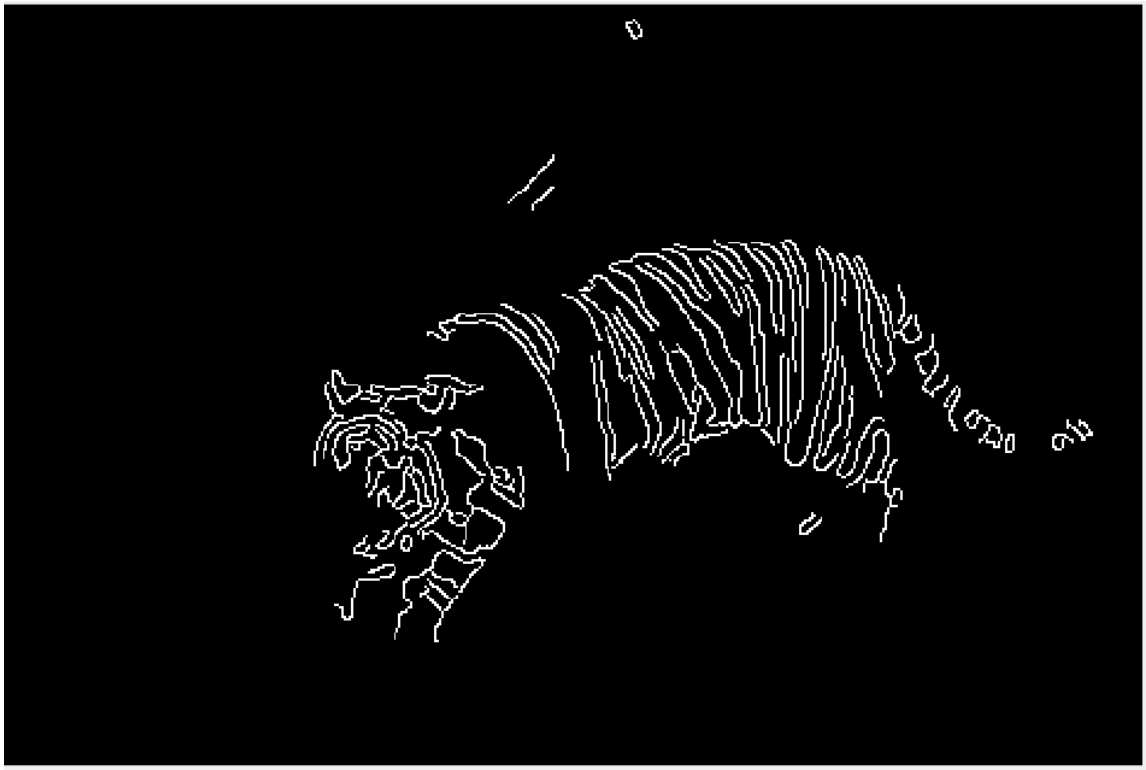
Edge detection using threshold as 0.20



Edge detection using threshold as 0.30



Edge detection using threshold as 0.40



Edge detection using threshold as 0.50

Edge detection using threshold as 0.60

Edge detection using threshold as 0.70

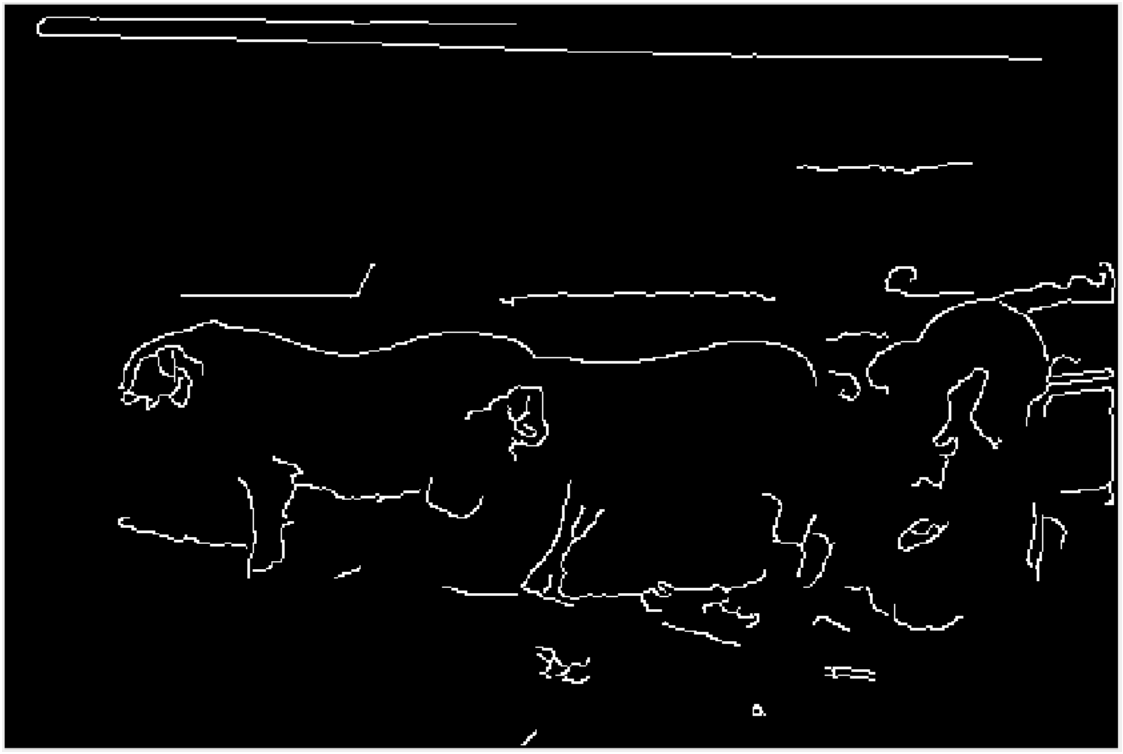
Edge detection using threshold as 0.80

Edge detection using threshold as 0.05

Edge detection using threshold as 0.10

Edge detection using threshold as 0.20

Edge detection using threshold as 0.30

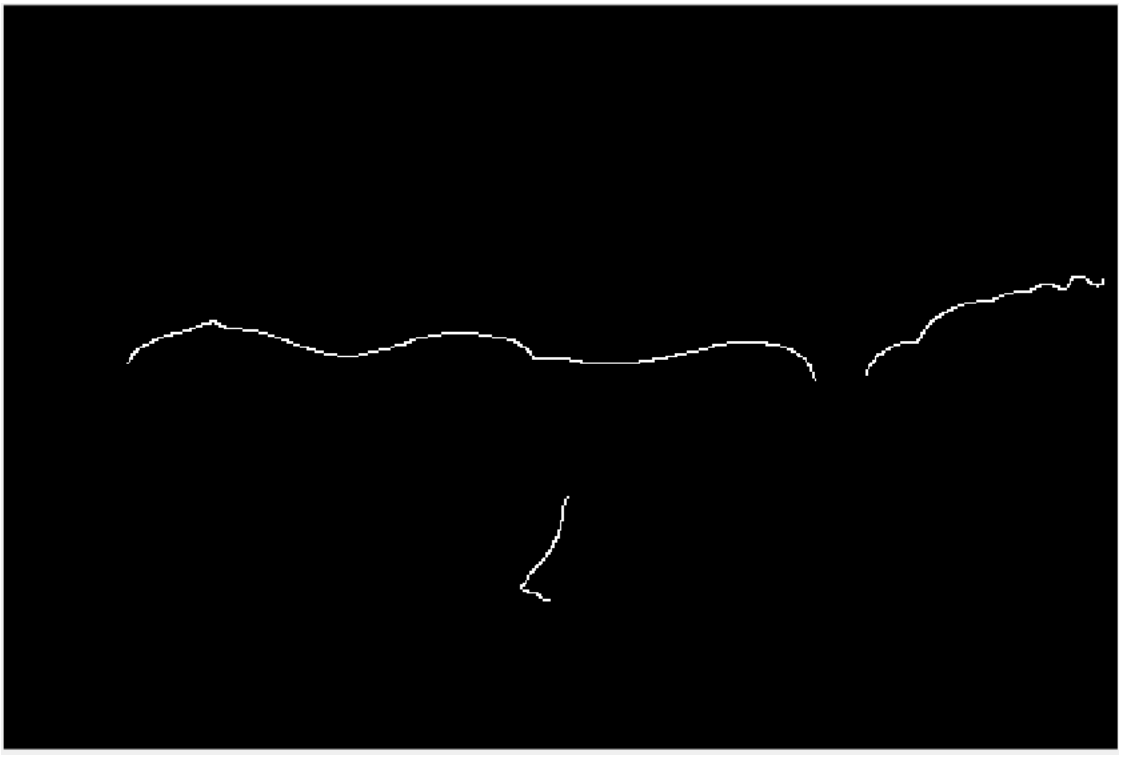


Edge detection using threshold as 0.40

Edge detection using threshold as 0.50

Edge detection using threshold as 0.60

Edge detection using threshold as 0.70



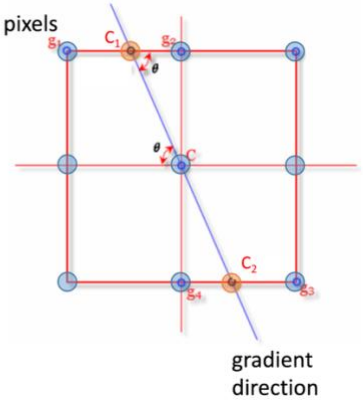
Edge detection using threshold as 0.80

IV. Discussion

Edge detection can be improved by using canny edge detection. It models edges as sharp changes in brightness between pixels. This method is based on point pixel property and expansion of Sobel detector with an additional layer of intelligence. This algorithm finds all the edges at minimal error rate. The Canny algorithm is adaptable to various environments. Its parameters allow it to be tailored to recognition of edges of differing characteristics depending on the particular requirements of a given implementation. However, the regular recursive implementation of the Canny operator does not give a good approximation of rotational symmetry and therefore gives a bias towards horizontal and vertical edges. Canny edge detector cannot differentiate if the edge is from the object or texture of an object. Changing the parameters of the Canny detector, one cannot avoid the edge detected inside the object. Canny edge detector doesn't possess this layer of intelligence that prevents it from detecting uninformative edges inside the object.

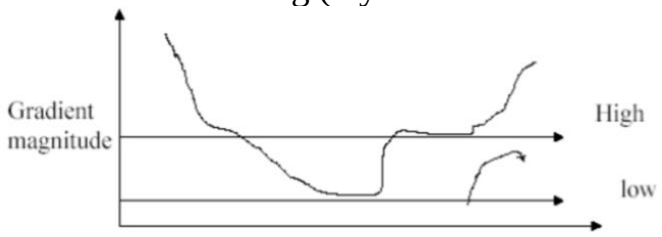
Non-maximum suppression (NMS)

The original output of the image edge map is quite rough. It's not good enough. We need a thin line to represent the edges. So this step will help us to remove the pixels which are not considered to be part of the edge. So basically, it is used to thin the edge. Thus, only thin lines (candidate edges) remain. So the approach is to suppress all the gradient values to 0 except the local maxima. For example below,



All the blue dots are pixels and the point of interest is the center one C. We find the gradient orientation of that pixel by using the formula above and along that gradient direction we find two pixels or points C1 and C2. They can be calculated by interpolation by the other two horizontal pair of points. And we do the compression of the gradient among these 3 points found C, C1, C2. If the gradient at C is larger than the gradients of C1 and C2 then this is the edge point we want, elsewise we ignore that point. Preserve gradient magnitude at pixel C, if it's value is larger than that of C1 and C2 (by interpolation), otherwise suppress its value to be 0.

Double Thresholding (Hysteresis Thresholding)



Canny recommended a *upper*:*lower* ratio between 2:1 and 3:1.

Canny introduced two thresholds instead of one, the lower one and the higher one. If the gradient is larger than the higher one, then it's strong enough, its very good. If it is lower than the lower one, it is too weak, and we ignore it. If it is between the two, then we have to double check whether it is connected to a point which we already decided as a pixel already. If yes, it is connected, then we keep that pixel otherwise we just ignore it.

- If a pixel gradient is higher than the *upper* threshold, the pixel is accepted as an edge
- If a pixel gradient value is below the *lower* threshold, then it is rejected.
- If the pixel gradient is between the two thresholds, then it will be accepted only if it is connected to a pixel that is above the *upper* threshold.

The Canny algorithm contains a number of adjustable parameters, which can affect the computation time and effectiveness of the algorithm.

- The size of the Gaussian filter: The smoothing filter used in the first stage directly affects the results of the Canny algorithm. Smaller filters cause less blurring, and allow detection of small, sharp lines. A larger filter causes more blurring, smearing out the value of a given pixel over a larger area of the image. Larger blurring radii are more useful for detecting larger, smoother edges – for instance, the edge of a rainbow.
- Thresholds: The use of two thresholds with hysteresis allows more flexibility than in a single-threshold approach, but general problems of thresholding approach still apply. A threshold set too high can miss important information. On the other hand, a threshold set too low will falsely identify irrelevant information (such as noise) as important. It is difficult to give a generic threshold that works well on all images. No tried and tested approach to this problem yet exists. It's a trial and error method, wherein we try different thresholds on the image and find the best match for us. It varies with the type of image used.

The edge maps for different thresholds for the tiger and pig images are shown above. In the tiger image, we observe that the stripes are shown accurately which help us to

recognize the image of the tiger, however, the face does not capture all the edges. In the pig image, we can see that the pig face isn't coming out clear. Also, the two pigs which are in front and back of each other, are very difficult to distinguish in the edge map. But Canny edge detector has done quite a fair job. And we can see that there are 3 pigs in the image when the threshold is little low. However, as we compare the results for the two images, we see that the tiger edge map is better than the pig edge map. But both the edge maps are quite accurate.
Threshold set at 0.3-0.5, mainly around 0.4 helps to get the better edge maps, which is simple and provides a good edge understanding. The edges are thin, sharp and clear. As I kept on increasing the threshold, edges were disappearing. As I reduced the threshold, the image was becoming noisy.

(c) Structured Edge

I. Abstract and Motivation

The Structured Edge detector is a new, improvised and advanced technique and is a data-driven edge detection approach. Data driven means there are two parts in this algorithm – one is training and other is testing. So the model itself needs to learn the feature or the structure of the image in the training process and then after it has learned something, it applies the learning to the testing part to tell whether the test image is what we want or not. So it learns everything from training and it applies what it learns to testing. In the theory part of this question, it explains the algorithm of Structured Edge (with a flowchart) and Random Forest (RF) Classifiers. In the experiment section of this question, I used the downloaded online source code for Edge detection method using Structured Edge in Matlab to perform analysis by adjusting various parameters.

(Source: EE569 – Discussion lecture 4)

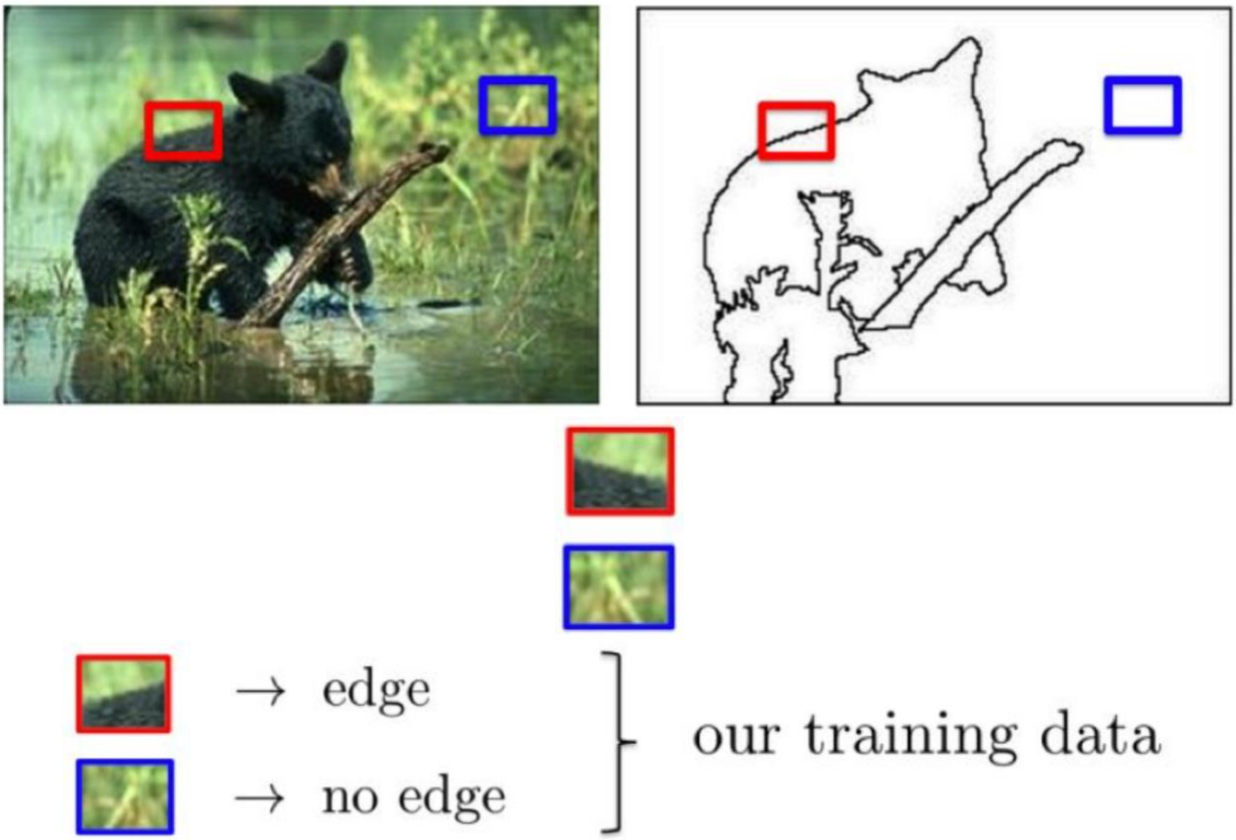
The images used for the dataset are from the Berkeley Segmentation Dataset and Benchmarks. It contains the original image and several separate subjects i.e. the ground truth images labelled by human beings.

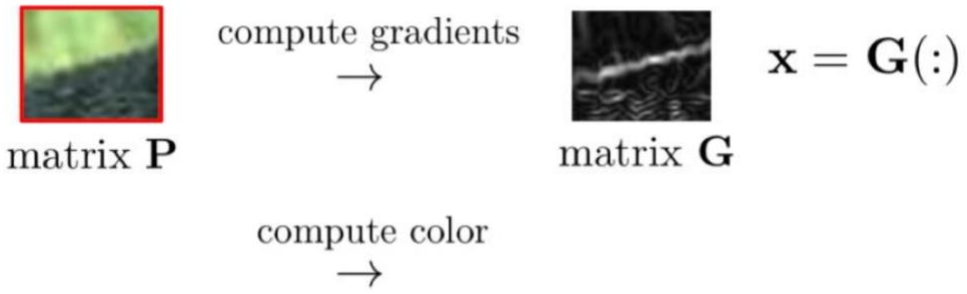
Berkeley Segmentation Dataset 500

- 500 images (200 train + 100 validation + 200 test)
- Each image was annotated by five subjects on average
- Served as evaluation of both “Segmentation” and “Contour”

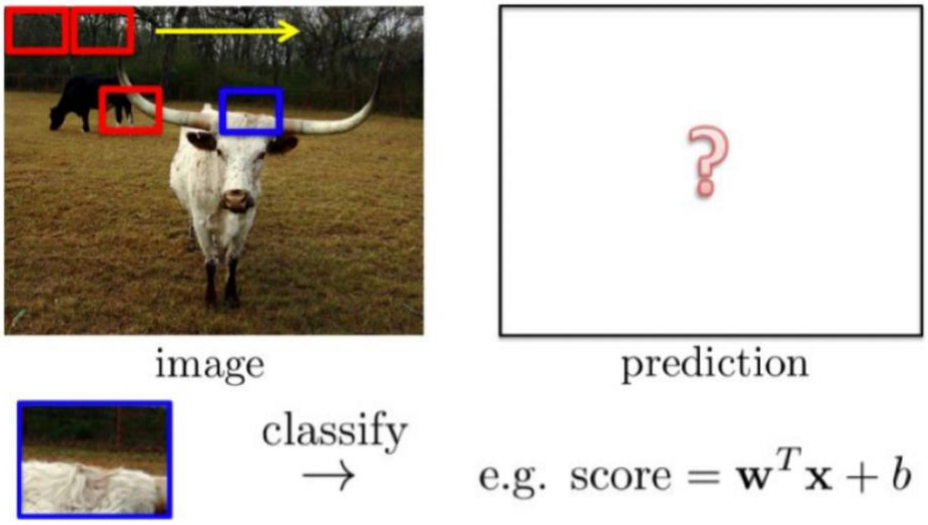
Training:

Firstly, we need to subtract a lot of blocks. Prepare the training data and labels. We call each such label as an image patch. We need to then prepare the positive and active samples, means we need to prepare the edge and non-edge samples. So as to make the system understand what is an edge and what is not. After subtracting all these images of small blocks, we have to extract or express in a mathematical way say a vector. So that vector is a feature vector of that small patch. And then we train our feature vector with our labels by the structured random forest. Structured random forest is like random forest and it has introduced some dimension reduction techniques and how to adjust the random forest to the edge detection problem. Now the model knows the feature of an edge or non-edge in a small patch.

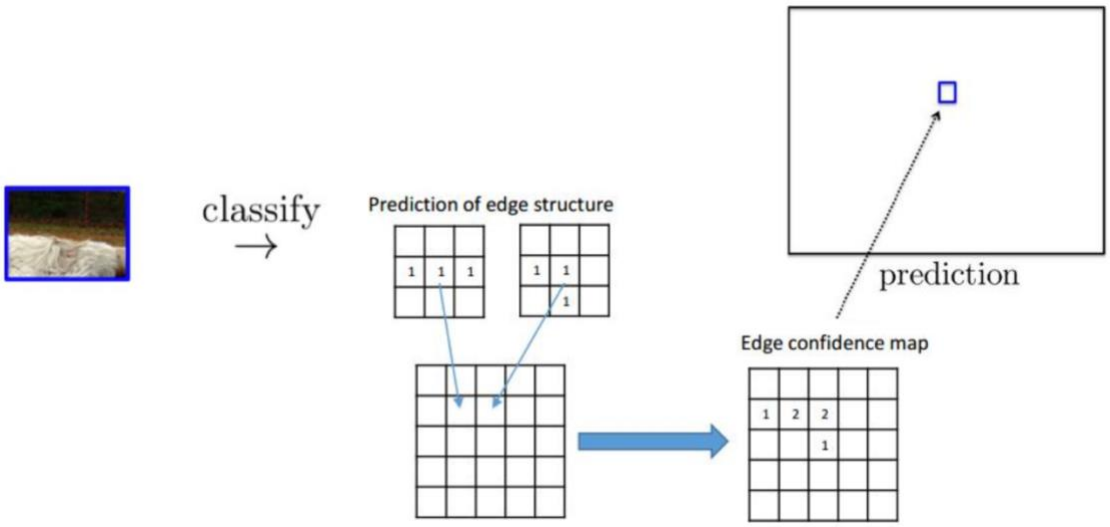


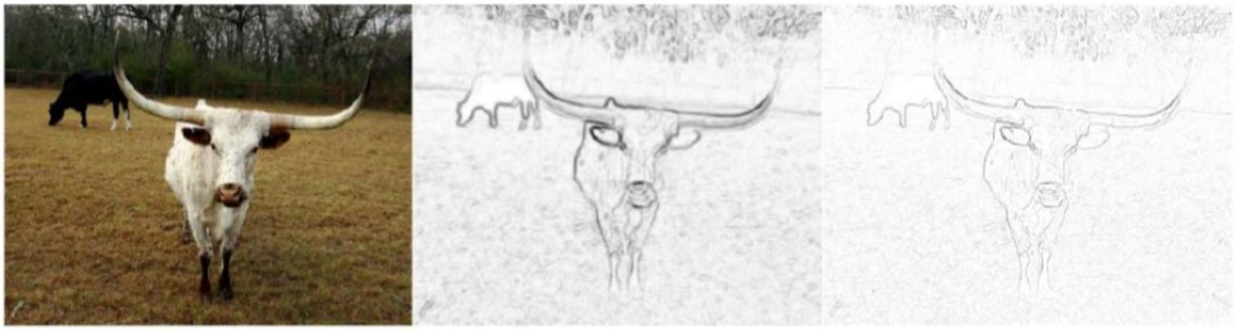


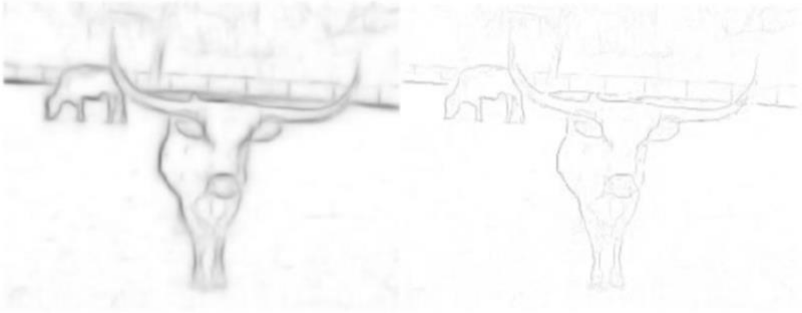
Testing:
Now we can apply the model to the new patches which are part of the testing images. For the testing image, we repeat the same procedure above that is, to extract the small image patches and we apply the standard we learned before so that we can predict if it is edge or not. Example, this is the straight forward linear determination used in the algorithm. There may be other non-linear ones.



For the classification part, there are some accumulation process. For example, in the testing image, for each pixel, the model predicts the edge pattern. If the edge exists, it is 1 else 0 for non-edge. Later, it accumulates all the pixels values which predicted for edge pattern exists or not. The final values calculated at each pixel are also called as edgeness score. Later we can apply NMS to the score, to get a better visual image which has simple edges displayed.







Applied the Structured Edge (SE) detector to extract edge segments from a color image with online source codes (released MATLAB toolbox: https://github.com/pdollar/edges).
(Source code: https://pdollar.github.io/toolbox/)

We can apply the SE detector to the RGB image directly without converting it into a grayscale image. Also, the SE detector generates a probability edge map. To obtain a binary edge map, we need to binarize the probability edge map with a threshold.

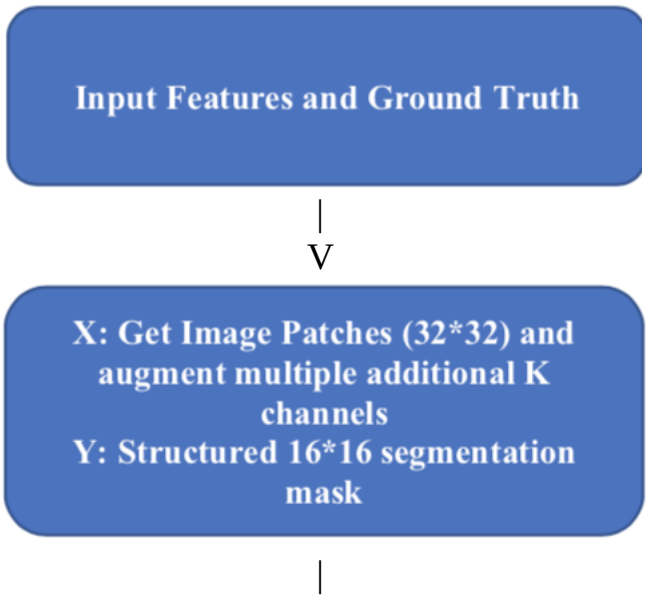
We also need to change the different parameters and hence we need to compare the different output results.

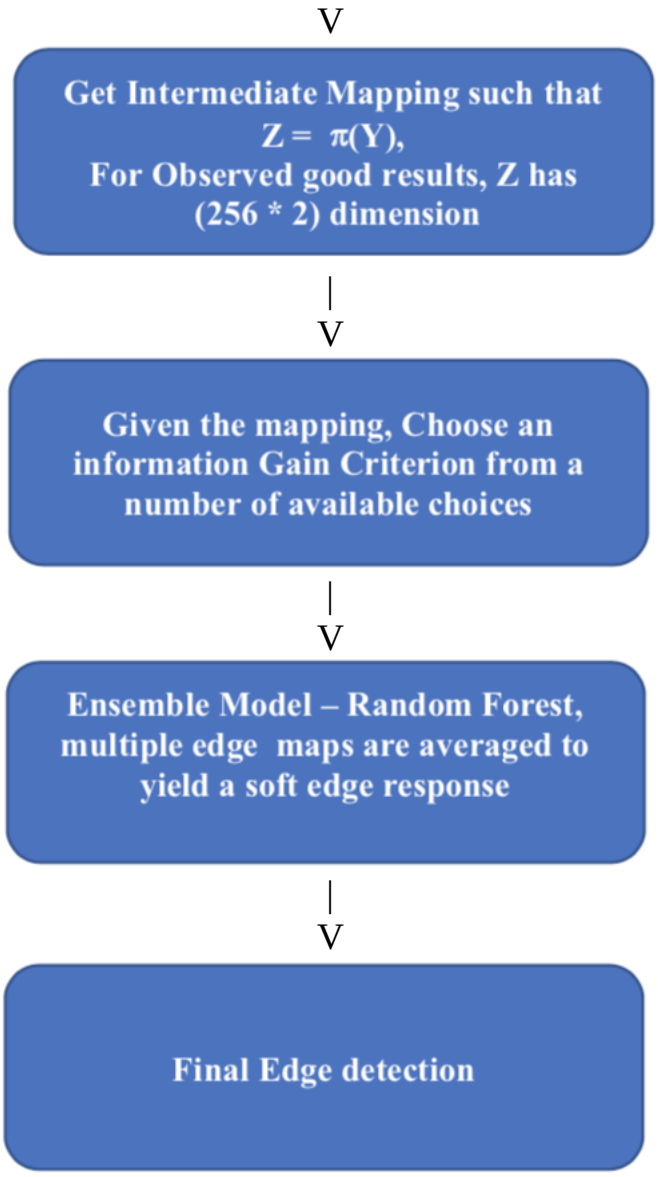
(Source: https://pdollar.github.io/files/papers/DollarPAMI15edges.pdf)

1. Structured Edge detection algorithm

Structured Forests for Fast Edge Detection was a research work originally published by Microsoft Research. Edge detection is a critical component of many vision systems, including object detectors and image segmentation algorithms. Patches of edges exhibit well-known forms of local structure, such as straight lines or T-junctions. In this paper one takes the advantage of the structure present in local image patches to learn both an accurate and computationally efficient edge detector. We formulate the problem of predicting local edge masks in a structured learning framework applied to random decision forests. The novel approach to learning decision trees robustly maps the structured labels to a discrete space on which standard information gain measures may be evaluated. The result is an approach that obtains real-time performance that is orders of magnitude faster than many competing state-of-the-art approaches and also the contour detection is very less noisy and efficient, while also achieving state-of-the-art edge detection results on the BSDS500 Segmentation dataset. Finally, we show the potential of the approach as a general-purpose edge detector by showing the learned edge models generalize well across datasets.

The SE detection algorithm flow chart:





The above flow chart explains Structured Edge Detection algorithm. The important step to check is the algorithm uses Random Forest using Decision Trees to compute the outputs.

We have the Berkeley dataset which gives us the ground truth and the input images file with the features.

There are two important elements which we use: X- Get the image patches for all the input images (32x32) and get together multiple K channels

Y- This denotes the structured 16x16 segmentation mask

Get the intermediate mapping such that Z contains mapping from Y to get better estimate of similarity. For observed good results, Z has (256x2) dimension.

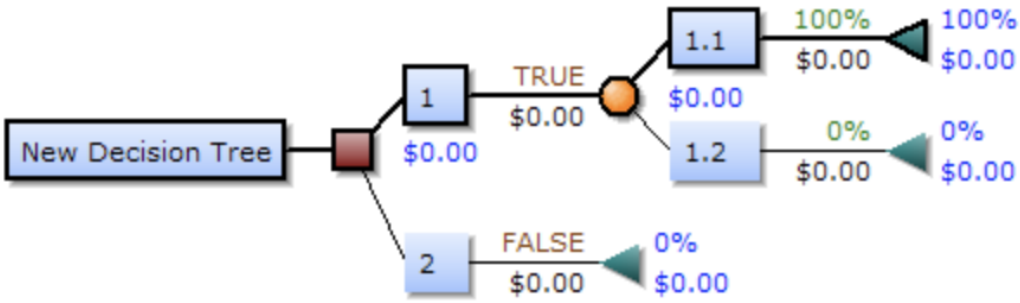
Now, given the mapping, choose an information gain from the number of available choices.
Ensemble model-random forest, multiple edge maps are averaged to yield a soft edge response. This sort of ensemble training is also used and proven the best in the field of Data Science, hence, it is not limited to only Image Processing applications.
Also, we can give a detailed explanation on Decision Trees Implementation and the Random Forest Classifier to get better insight on the whole topic.
The performance of such a robust algorithm can be evaluated by various metrics like Precision, Recall and F Measure. When we compare this Structured Edge algorithm with the others, this machine learning approach is far more advanced and gives better results than the other methods we used above for edge detection.

2. The Random Forest Classifier

The Random Forest (RF) classifier is used in the SE detector. The RF classifier consists of multiple decision trees and integrate the results of these decision trees into one final probability function. the process of decision tree construction and the principle of the RF classifier is explained below:

(Source: https://en.wikipedia.org/wiki/Decision_tree
https://en.wikipedia.org/wiki/Random_forest)

The process of decision tree construction:
A decision tree is a decision support tool that uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements.
Decision trees are commonly used in operations research, specifically in decision analysis, to help identify a strategy most likely to reach a goal, but are also a popular tool in machine learning.
Decision tree elements are shown below:



Drawn from left to right, a decision tree has only burst nodes (splitting paths) but no sink nodes (converging paths). Therefore, used manually, they can grow very big and are then often hard to draw fully by hand.

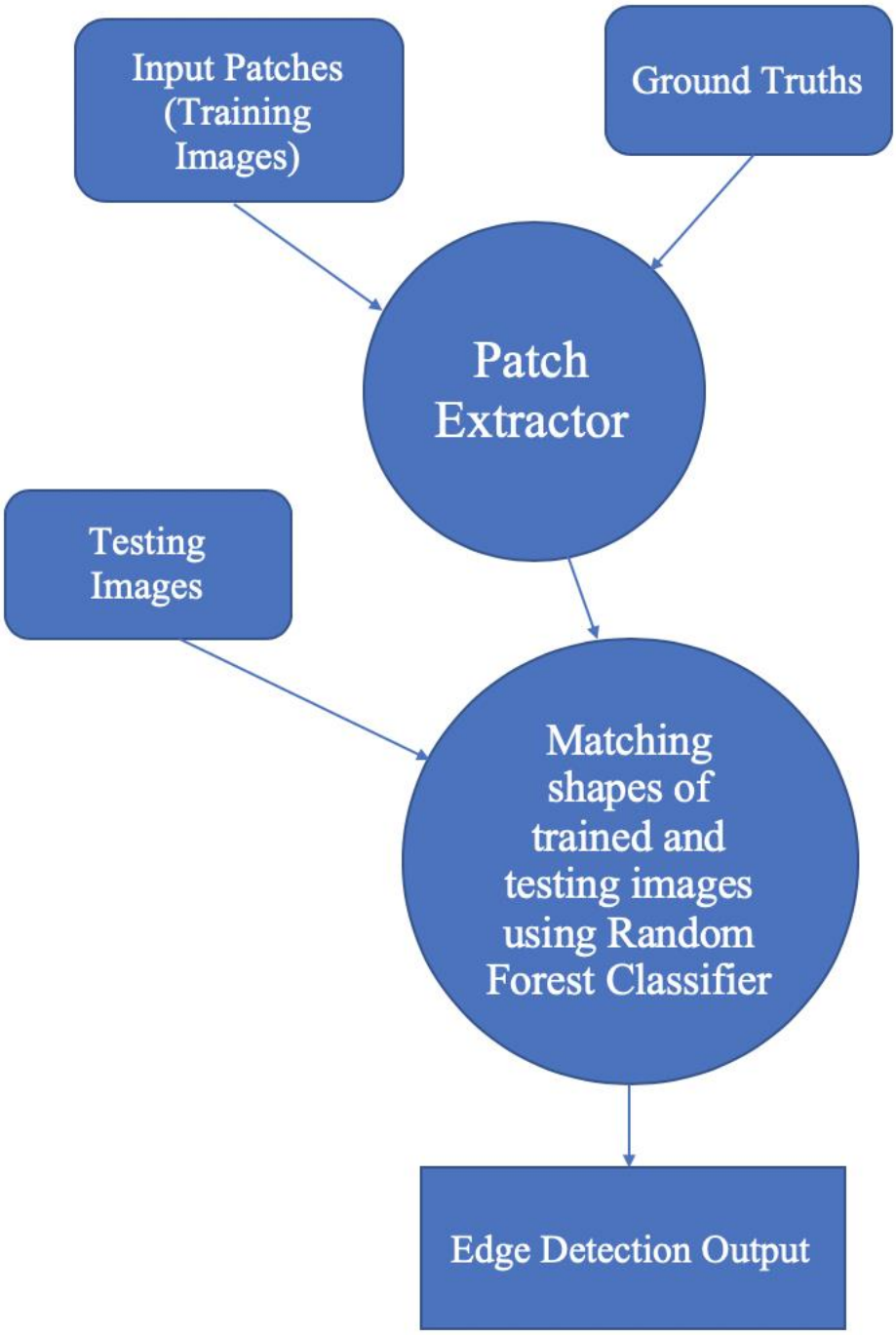
A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.
In decision analysis, a decision tree and the closely related influence diagram are used as a visual and analytical decision support tool, where the expected values (or expected utility) of competing alternatives are calculated.
A decision tree consists of three types of nodes:

1. Decision nodes – typically represented by squares
2. Chance nodes – typically represented by circles
3. End nodes – typically represented by triangles

The principle of RF classifier:
Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

Decision trees are a popular method for various machine learning tasks. In particular, trees that are grown very deep tend to learn highly irregular patterns: they overfit their training sets, i.e. have low bias, but very high variance. Random forests are a way of averaging multiple deep decision trees, trained on different parts of the same training set, with the goal of reducing the variance. This comes at the expense of a small increase in the bias and some loss of interpretability, but generally greatly boosts the performance in the final model.

II. Approach and Procedure



The structured edge algorithm is explained above in the flowchart. The Random Forest Classifier takes input training images and ground truths to construct a forest having multiple decision trees. As the decision trees in the forest have high variability, they tend to overfit the data. Hence, an ensemble of the decision trees is used wherein the prediction of each decision tree is combined to generate the final output.

Structure edge detector considers the neighborhood of a pixel to detect the presence or absence of an edge. The edge patches formed from the patch extractor are classified into sketch tokens to take into consideration all the variability of different edge patterns of the input patches.
After a considerable amount of decision trees are added in the Random Forest Classifier, the testing images are given to the classifier. Random Forest uses the divide and conquer algorithm where it classifies the input by moving to the left or right of the current node and stops
after reaching the leaf node.
A set of decision trees are trained in such a way that they give optimal split of data. The Random Forest Classifier matches the testing image patches to the trained classifier to make a decision about the presence of an edge.
The performance of such a robust algorithm can be evaluated by various metrics like Precision, Recall and F Measure. When we compare this Structured Edge algorithm with the others, this machine learning approach is far more advanced and gives better results than the other methods we used above for edge detection.

A random forest is a neat approach and very efficient. They run on the divide and conquer logic that's the heart that helps to improve their performance. The main idea of ensemble approach is to put a set of weak learners to form a strong learner. The basic element of a random forest is a decision tree. The decision trees are weak learners that are put together to form a random forest.
A decision tree classifies an input that belongs to space A as an output that belongs to a space B. The input or output space can be complex in nature. For example, we can have a histogram as the output. In a decision tree, an input is entered at the root node. The input then traverses down the tree in a recursive manner till it reaches the leaf node. Each node in the decision tree has a binary split function with some parameters. This split function decides whether the test sample should progress towards the right child node or the left child node. Often, the split function is complex.

Steps for implementing the Random Forest algorithm using Decision Trees:

(Source: http://dataaspirant.com/2017/05/22/random-forest-algorithm-machine-learing/)

Training:

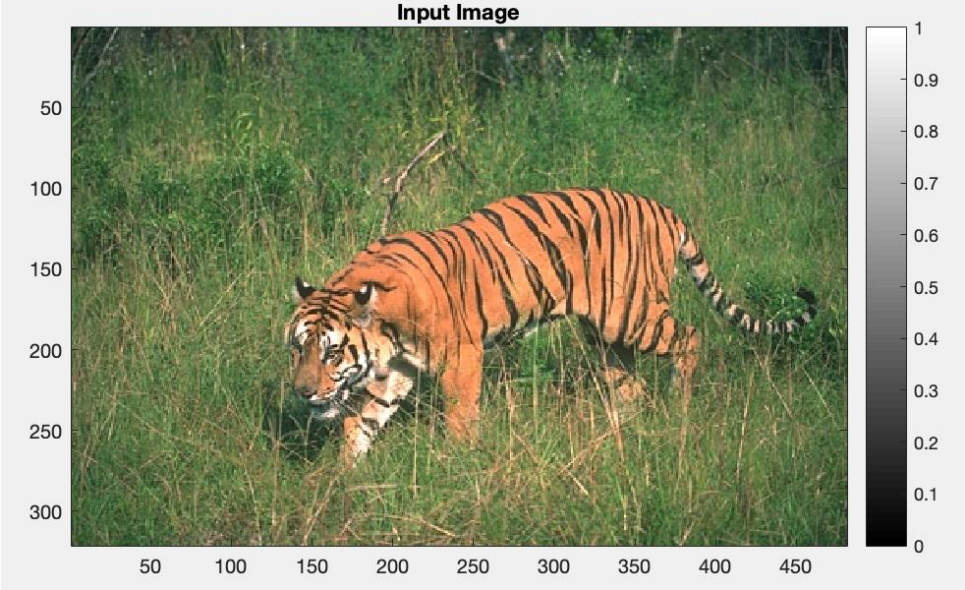
- Randomly select "K" features from total of "M" features such that K << M
- From the selected K features, calculate the node "d" using the best split point criteria

- 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 numbers of decision trees

Testing:

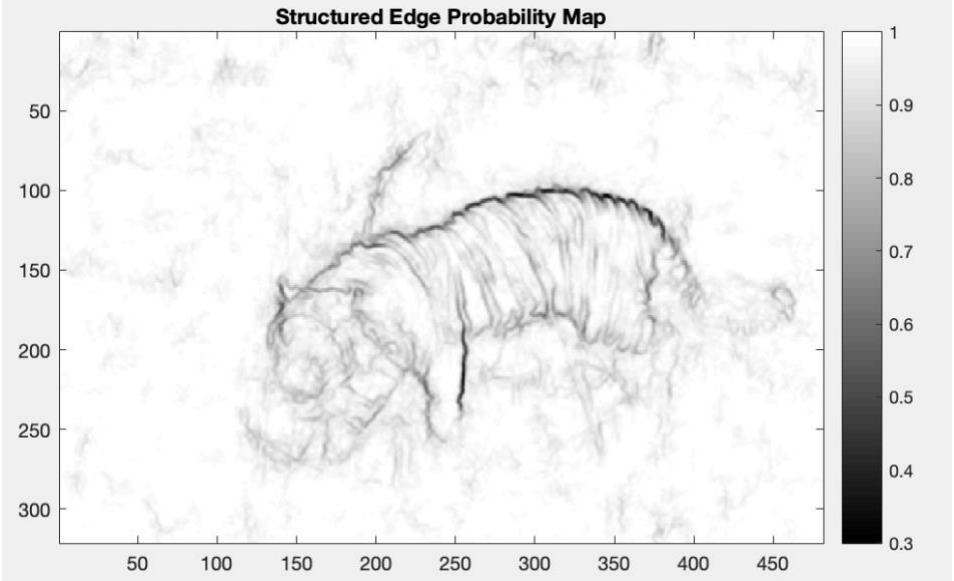
- Get the test feature and use the decision rules developed for every decision tree to predict the outcome
- Find the means of the outputs if they are continuous and consider them as the final prediction

III. Experimental Results



Input image tiger.raw

Structured Edge Detection with Bad Parameters for Tiger

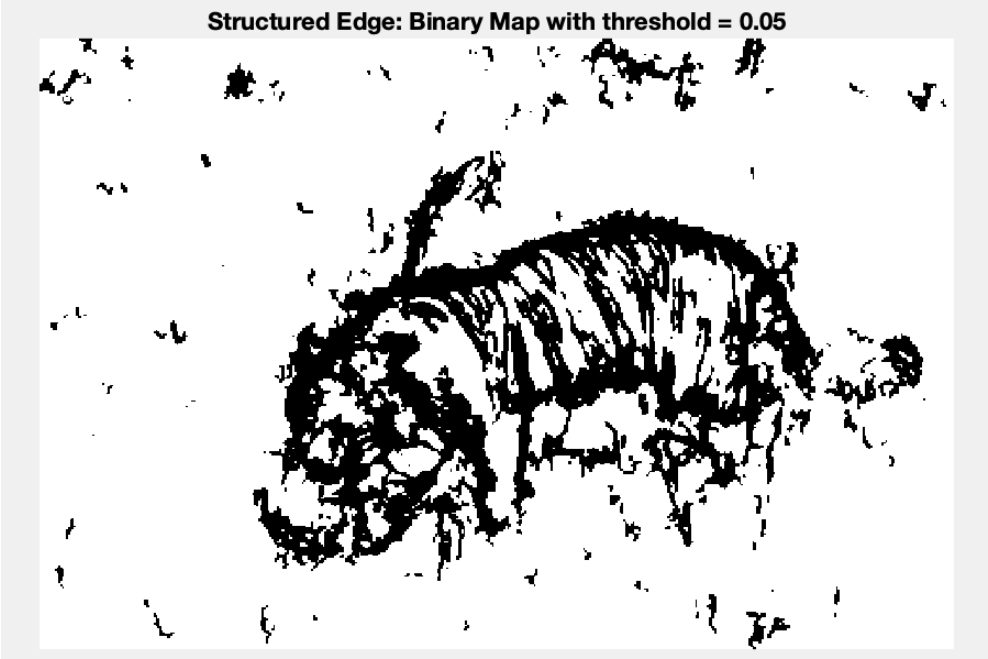


Structured Edge Probability Map of Tiger.jpg

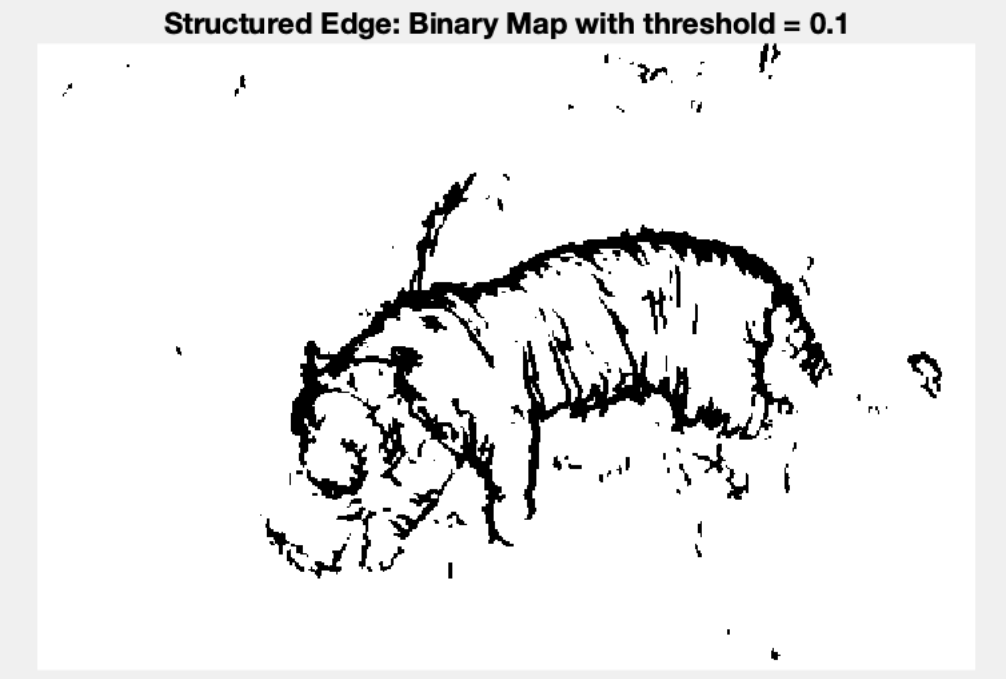
Parameters chosen

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

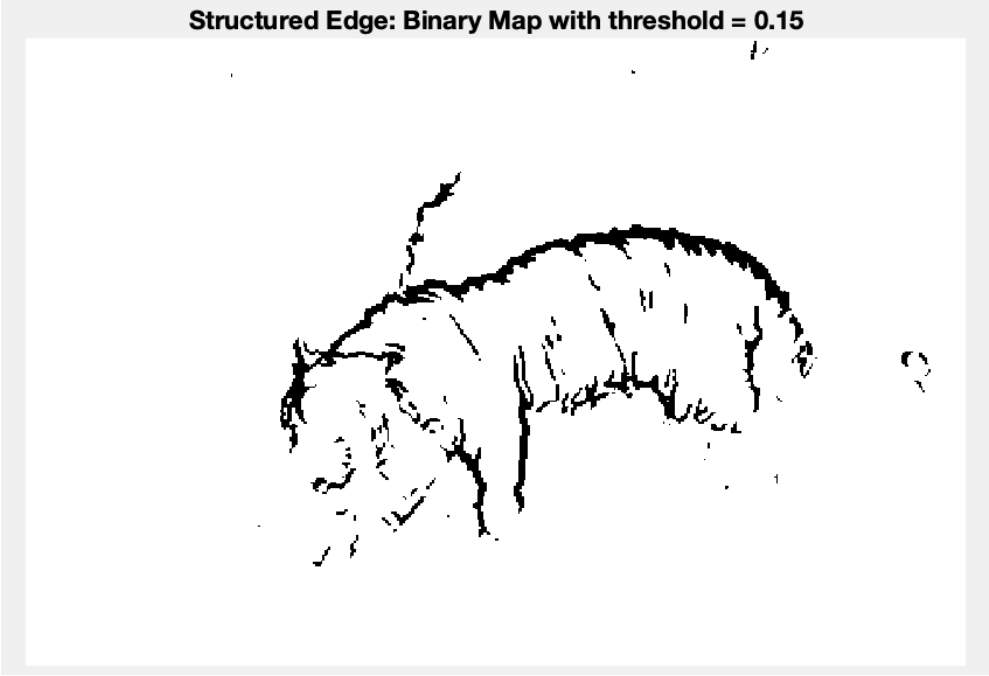
Threshold mentioned below is the probability and the binary map corresponds to the binary edge map



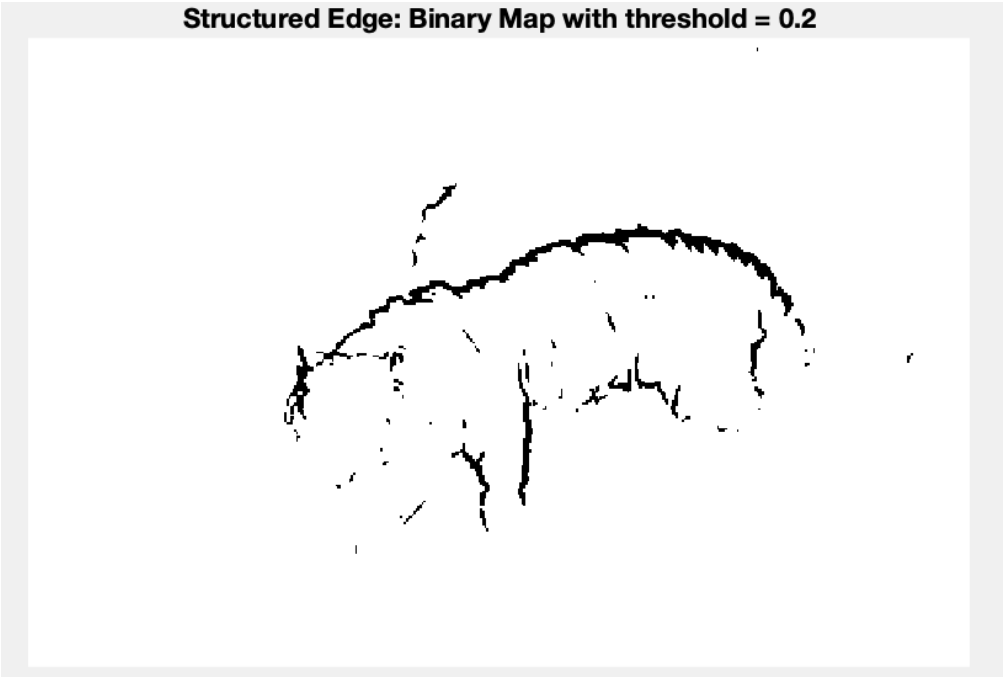
Structured Map: Binary map using threshold = 0.05



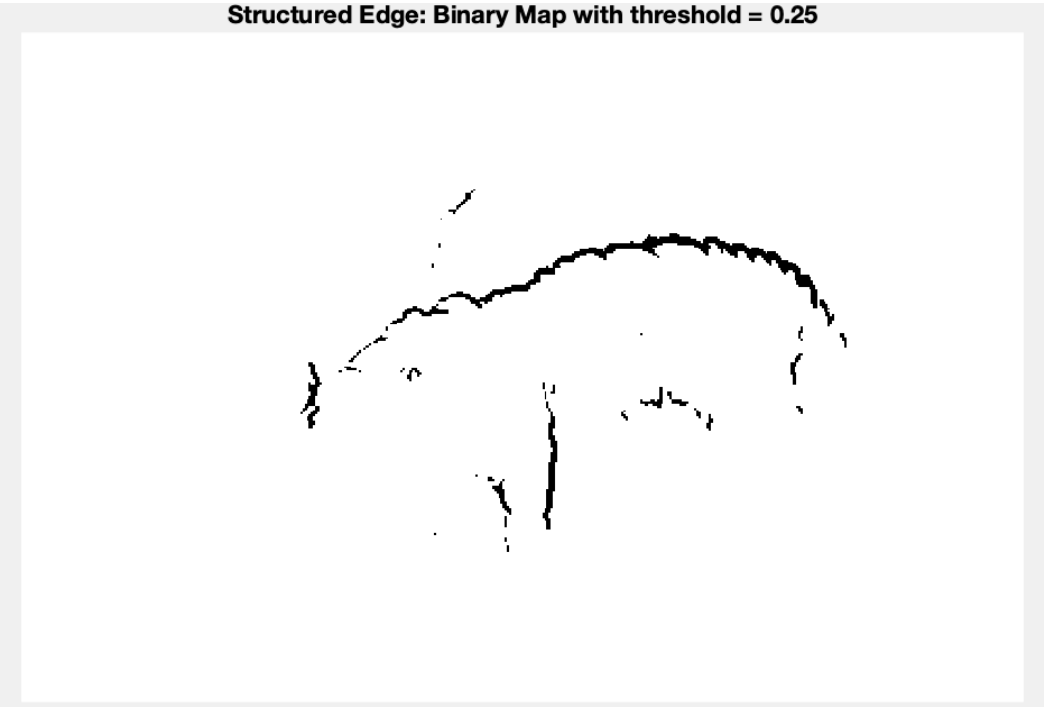
Structured Map: Binary map using threshold = 0.10



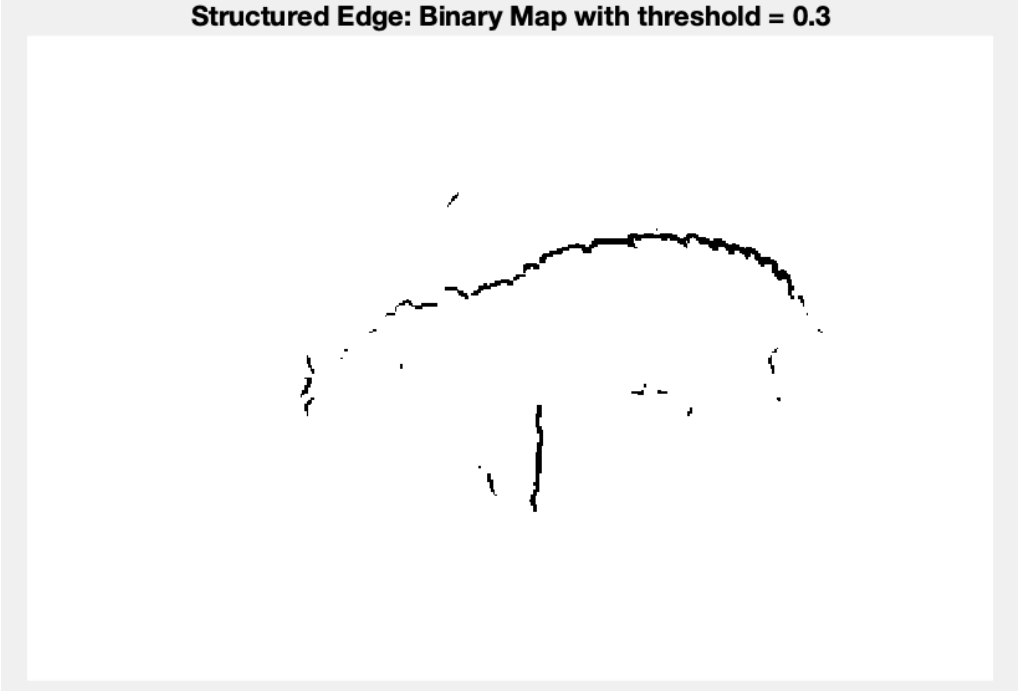
Structured Map: Binary map using threshold = 0.15



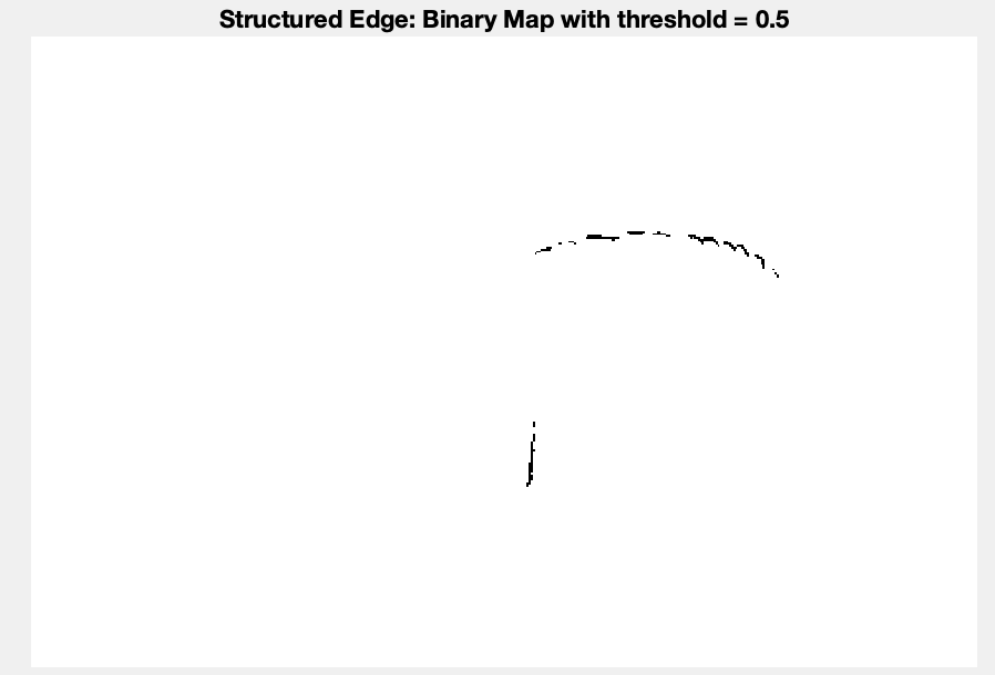
Structured Map: Binary map using threshold = 0.20



Structured Map: Binary map using threshold = 0.25

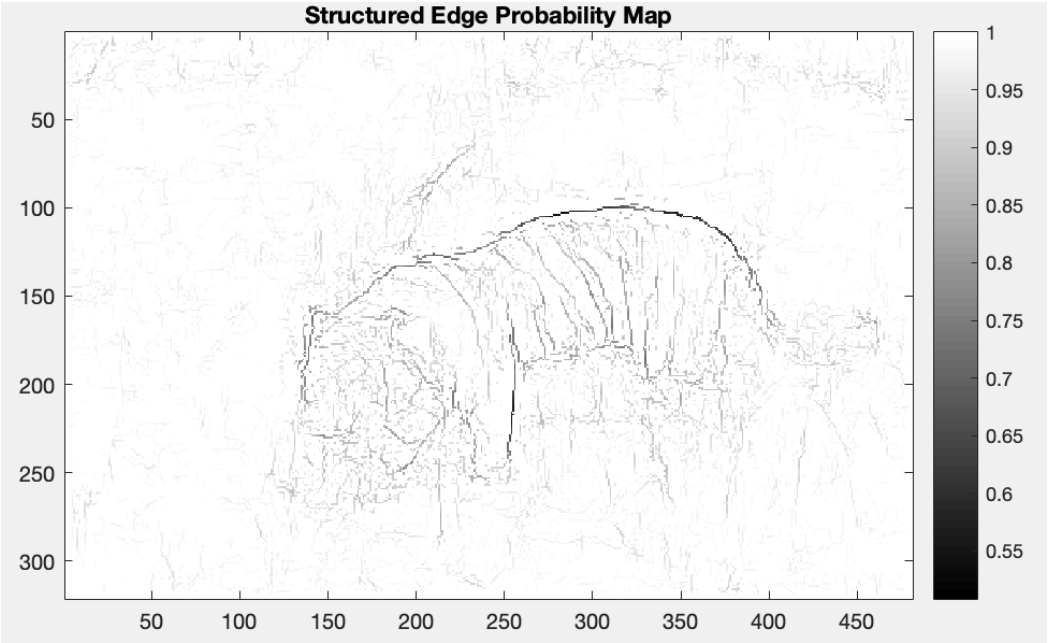


Structured Map: Binary map using threshold = 0.30



Structured Map: Binary map using threshold = 0.50

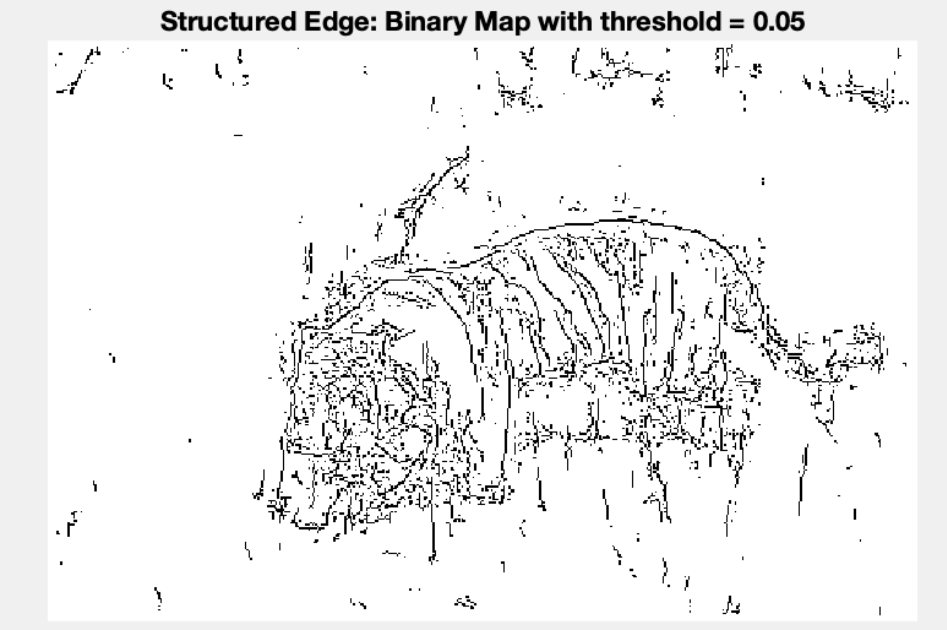
Structured Edge Detection with Best Parameters for Tiger



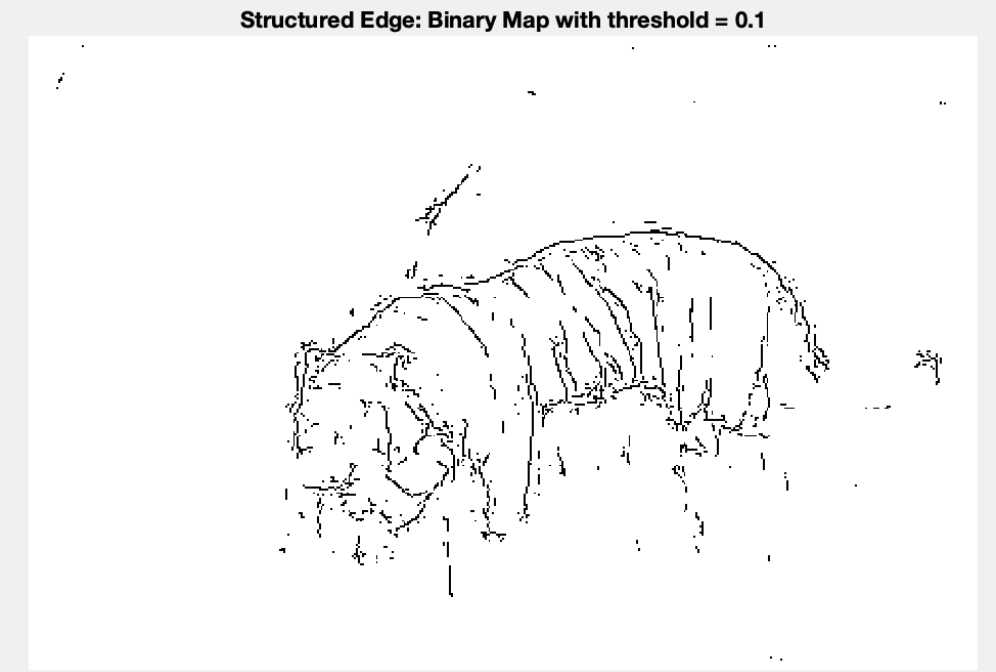
Structured Edge Probability Map of Tiger.jpg

Parameters chosen

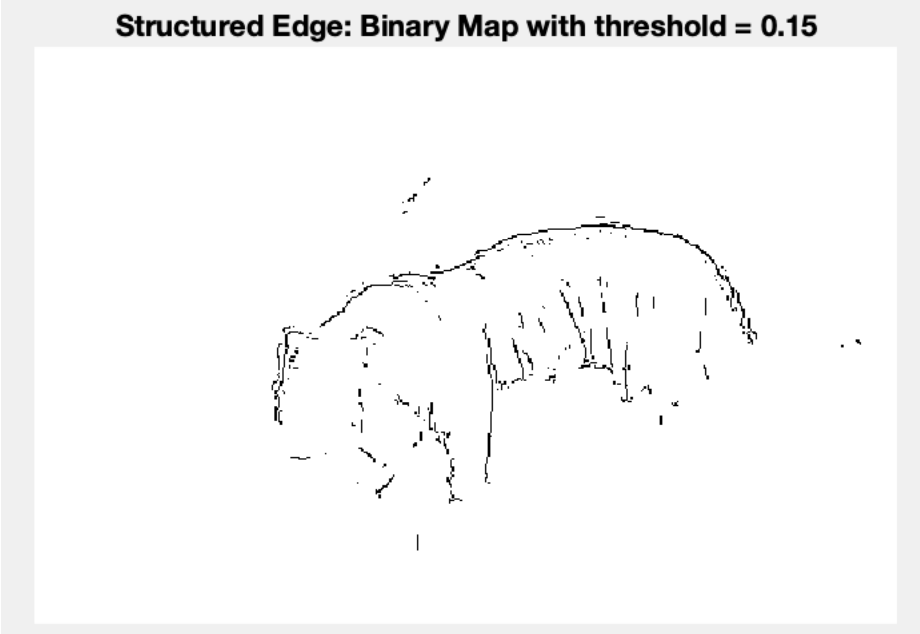
```
%% set detection parameters (can set after training)
model.opts.multiscale=1;          % for top accuracy set multiscale=1
model.opts.sharpen=0;             % for top speed set sharpen=0
model.opts.nTreesEval=1;          % for top speed set nTreesEval=1
model.opts.nThreads=4;            % max number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



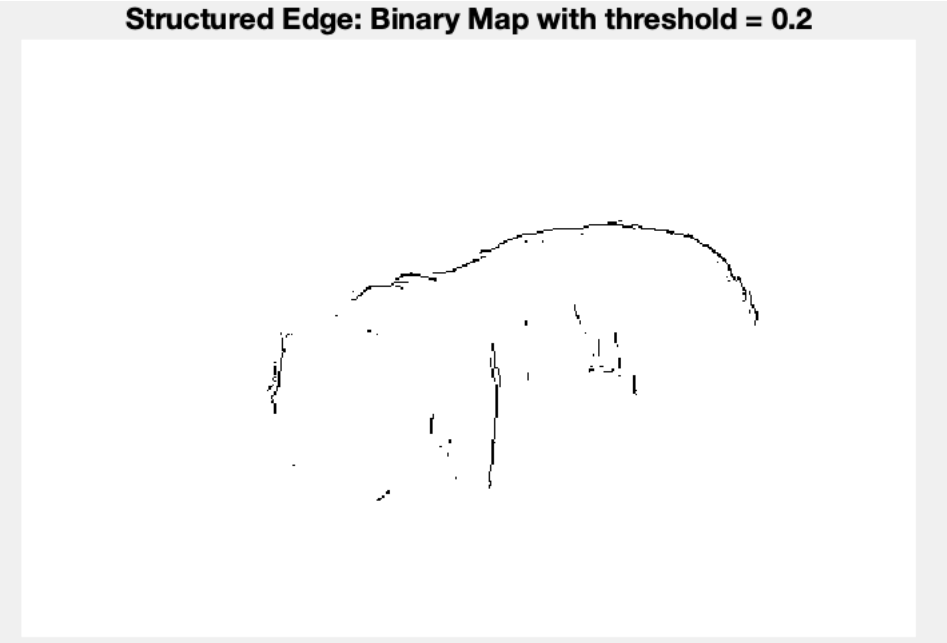
Structured Map: Binary map using threshold = 0.05



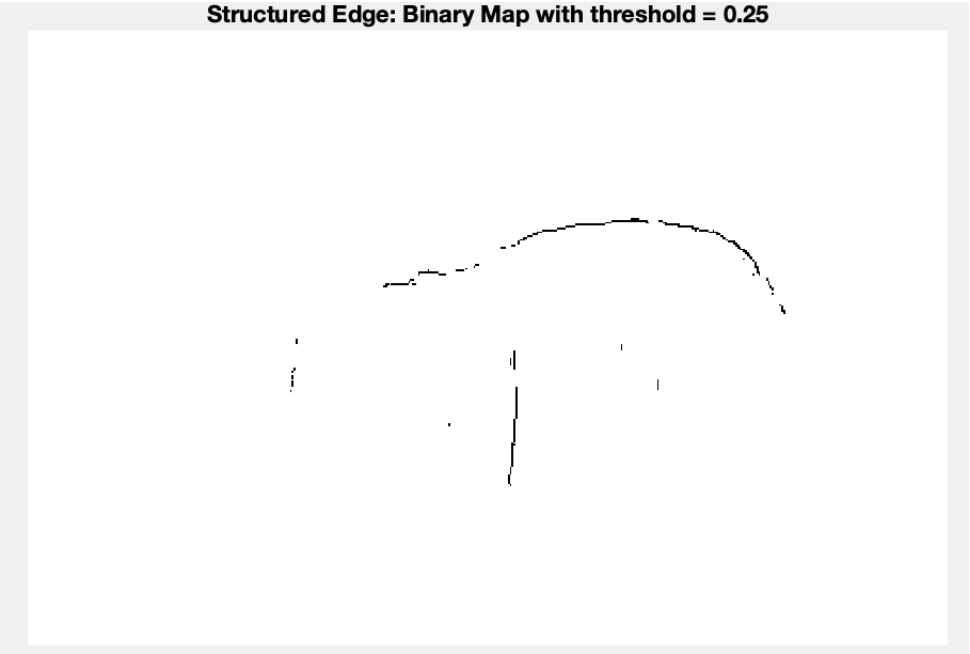
Structured Map: Binary map using threshold = 0.10



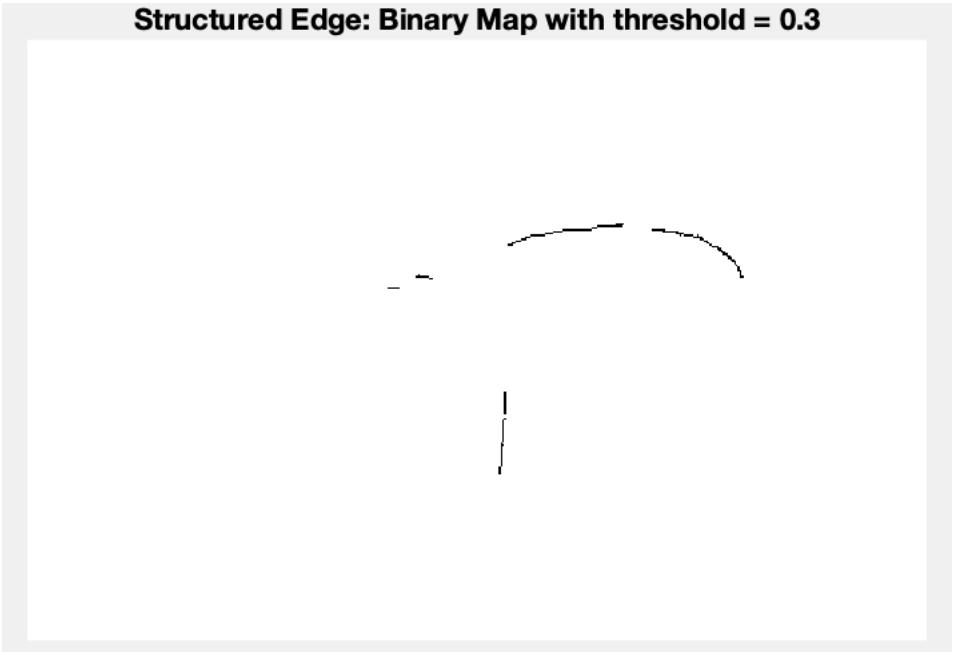
Structured Map: Binary map using threshold = 0.15



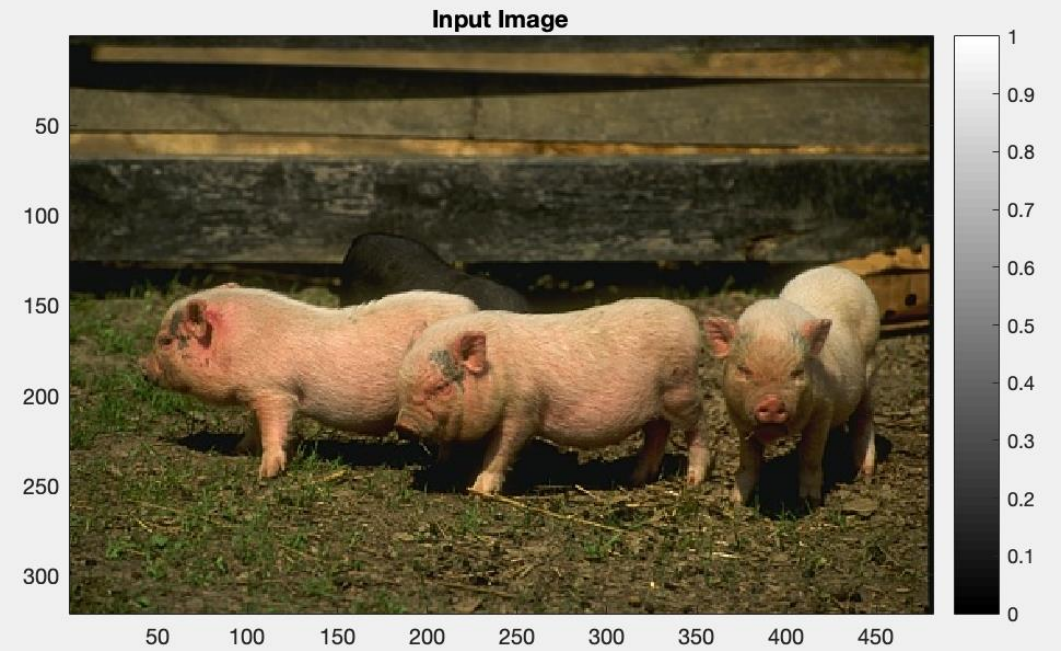
Structured Map: Binary map using threshold = 0.20



Structured Map: Binary map using threshold = 0.25

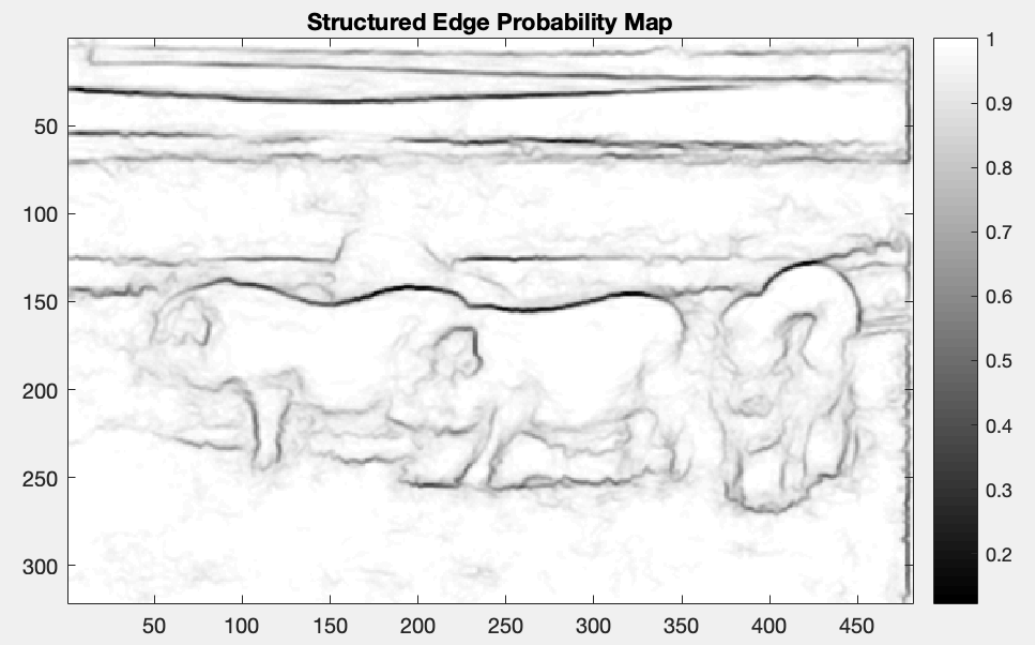


Structured Map: Binary map using threshold = 0.30



Input image pig.raw

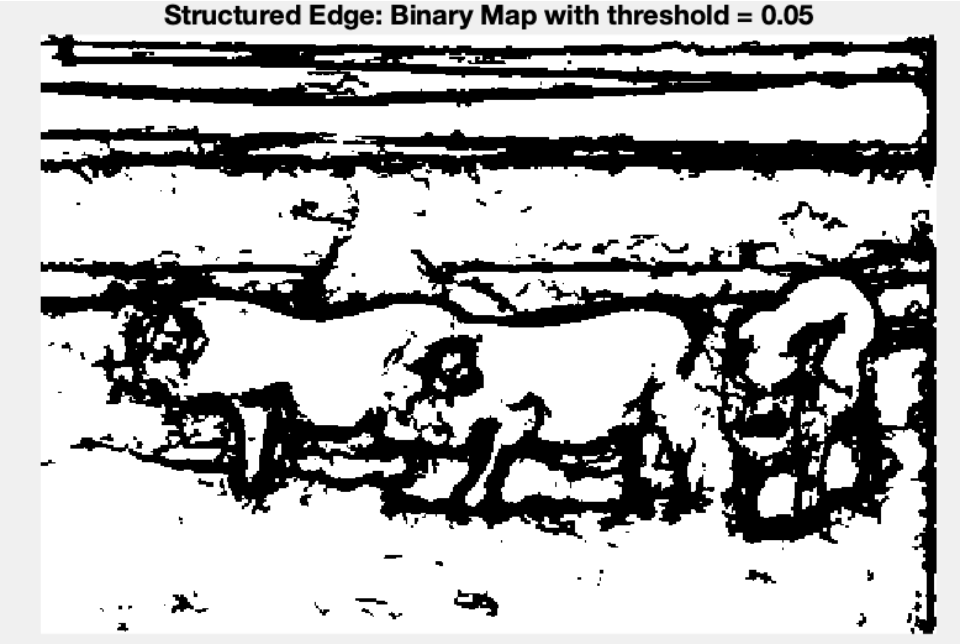
Structured Edge Detection with Bad Parameters for Pig



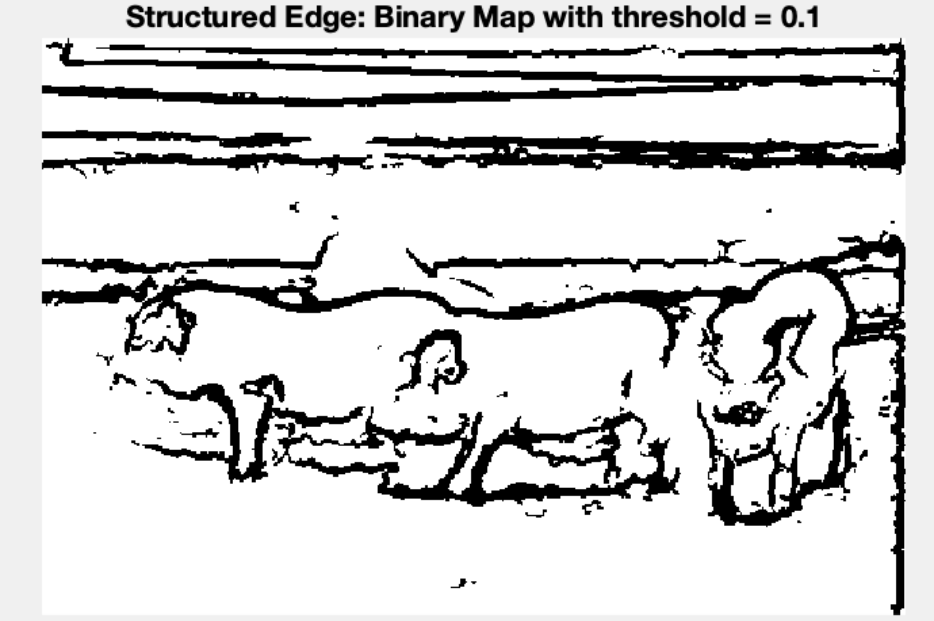
Structured Edge Probability Map of Pig.jpg

Parameters chosen

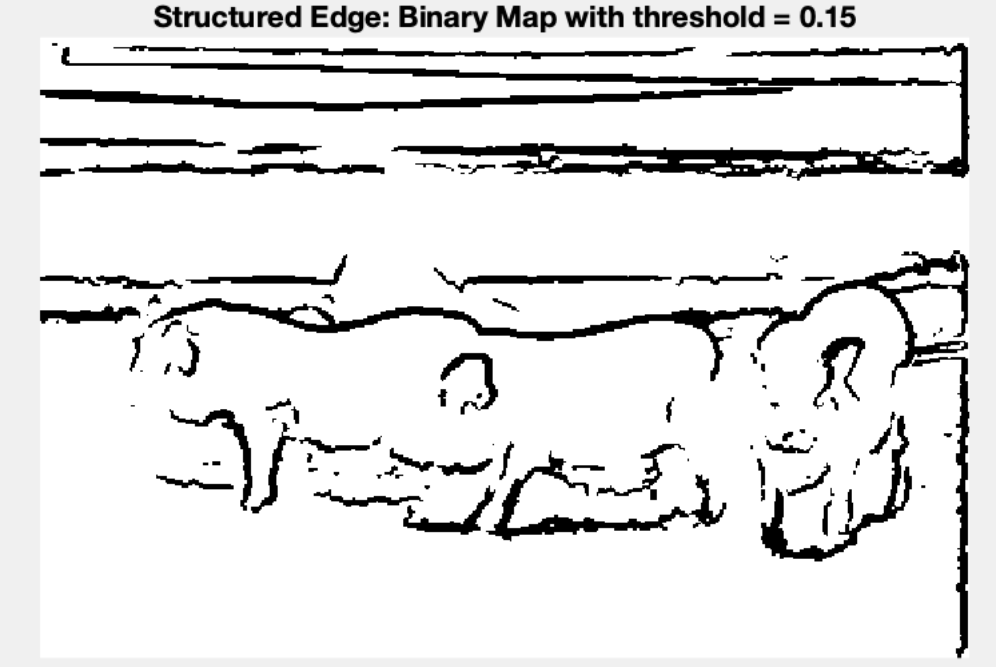
```
%% set detection parameters (can set after training)
model.opts.multiscale=0;          % for top accuracy set multiscale=1
model.opts.sharpen=2;             % for top speed set sharpen=0
model.opts.nTreesEval=2;          % for top speed set nTreesEval=1
model.opts.nThreads=4;            % max number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



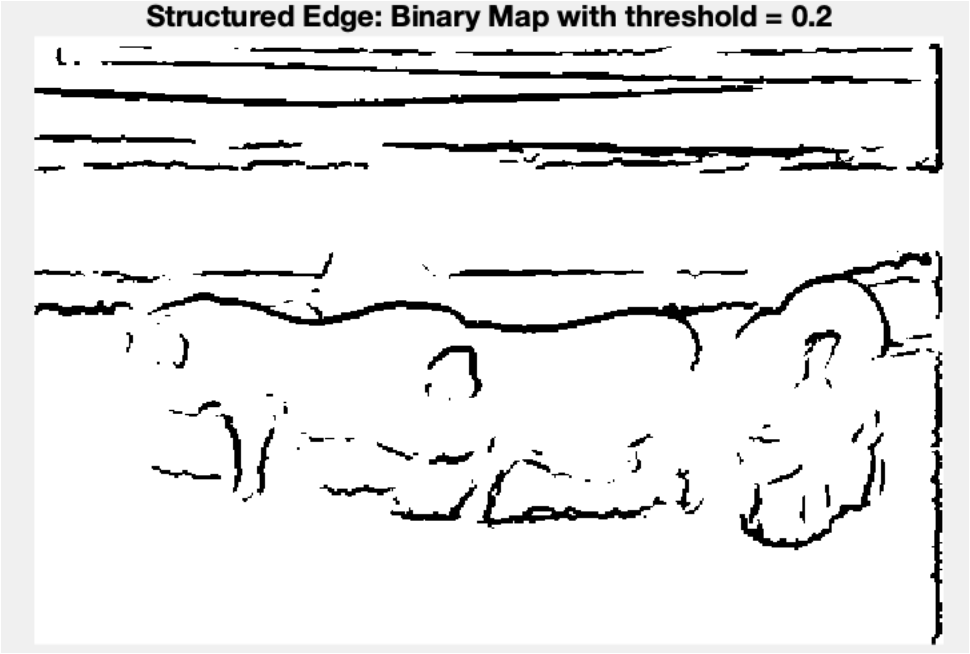
Structured Map: Binary map using threshold = 0.05



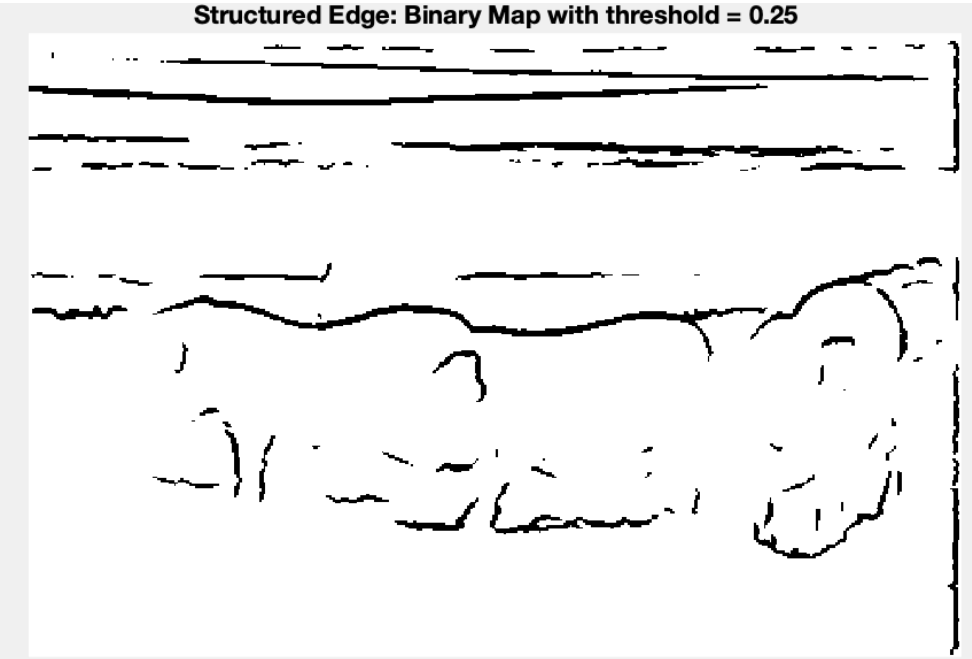
Structured Map: Binary map using threshold = 0.10



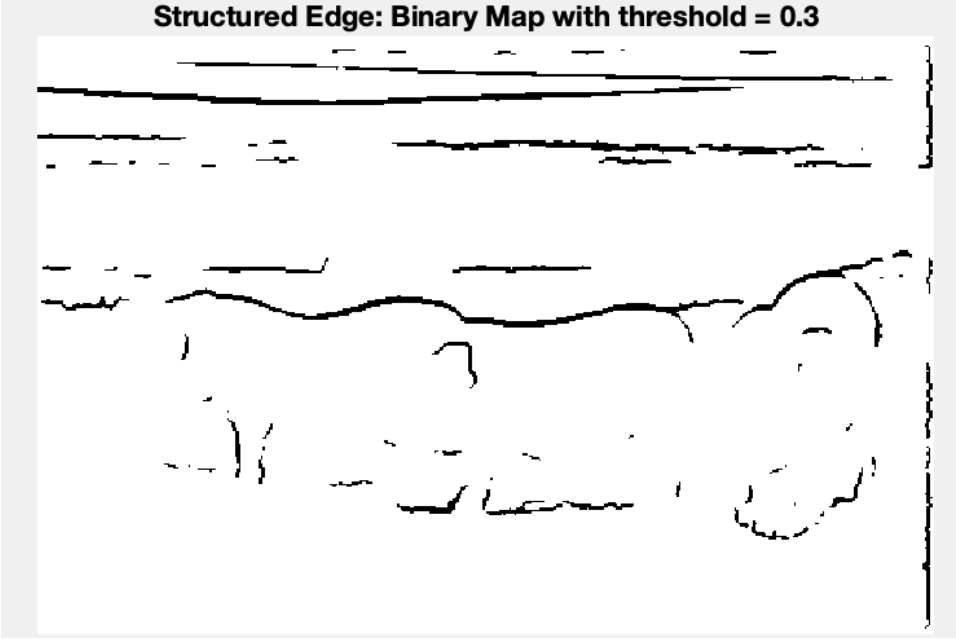
Structured Map: Binary map using threshold = 0.15



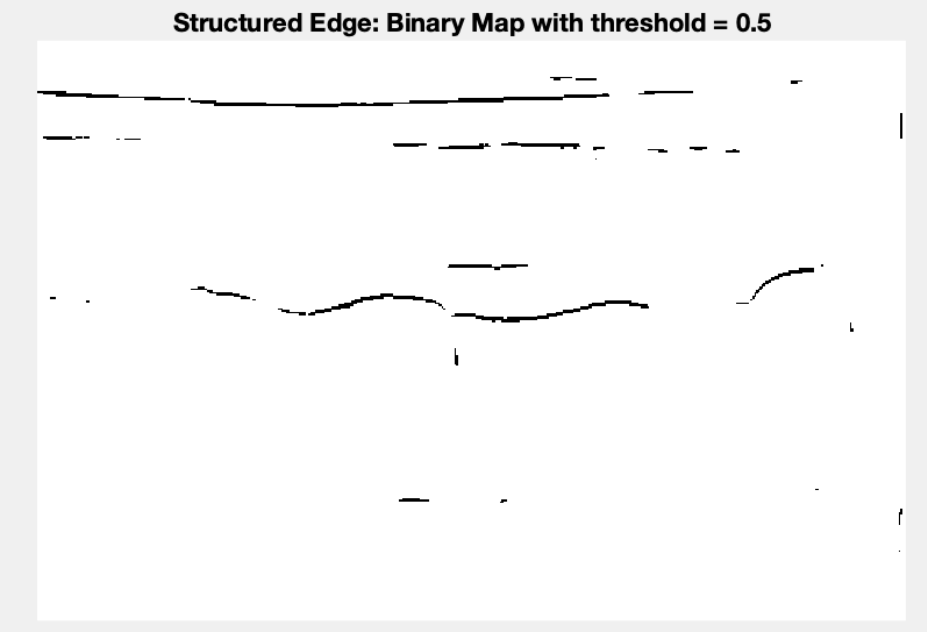
Structured Map: Binary map using threshold = 0.20



Structured Map: Binary map using threshold = 0.25

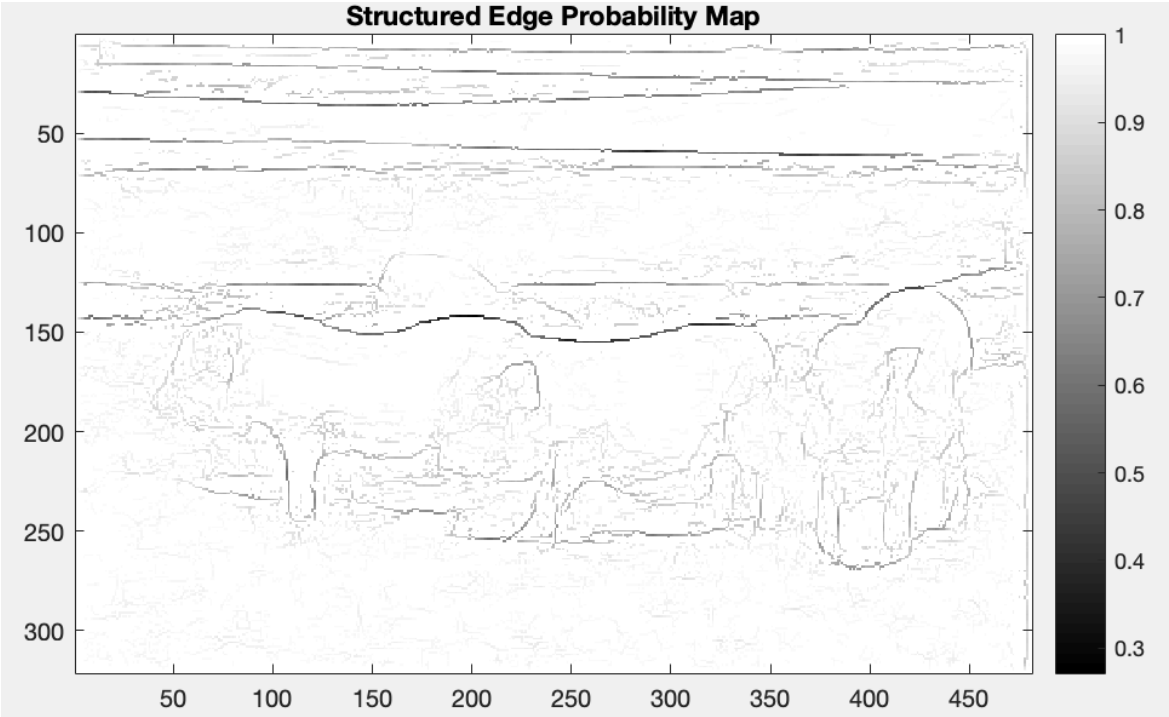


Structured Map: Binary map using threshold = 0.30



Structured Map: Binary map using threshold = 0.50

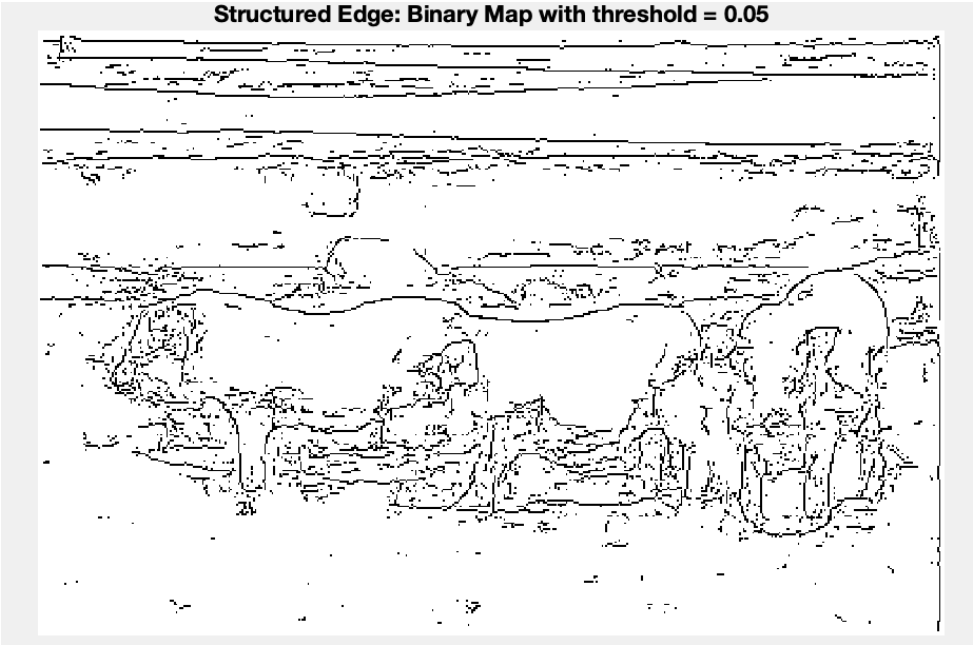
Structured Edge Detection with Best Parameters for Pig



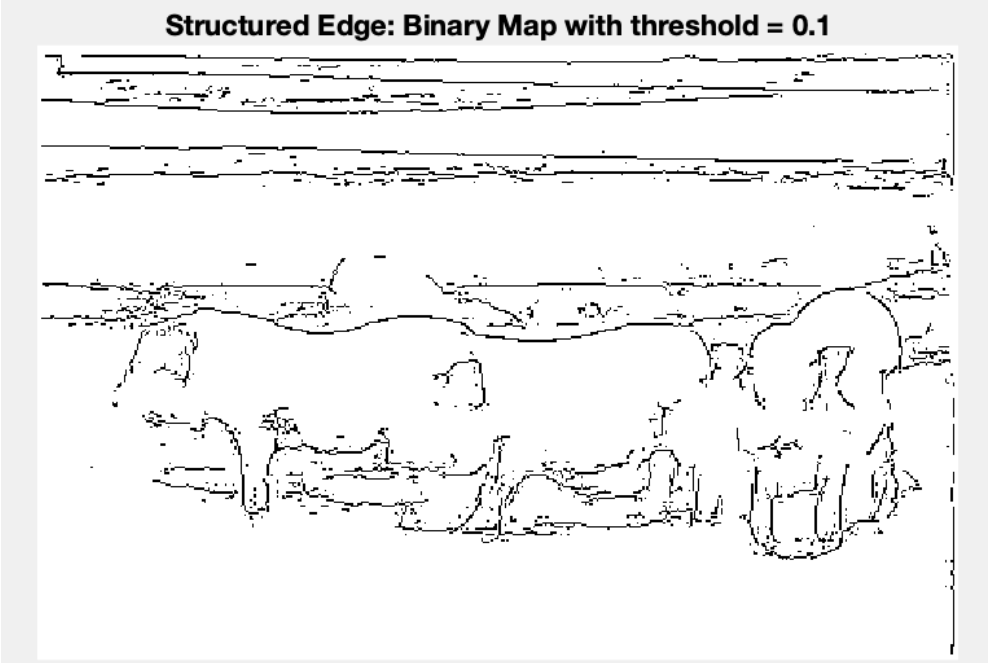
Structured Edge Probability Map of Pig.jpg

Parameters chosen

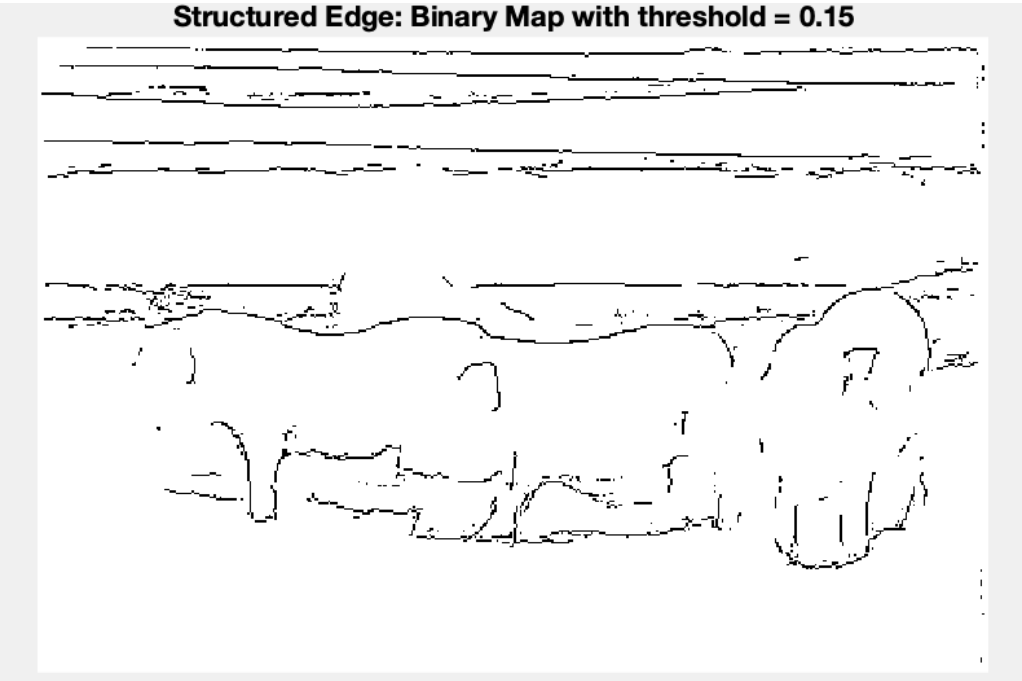
```
%% set detection parameters (can set after training)
model.opts.multiscale=1;          % for top accuracy set multiscale=1
model.opts.sharpen=0;             % for top speed set sharpen=0
model.opts.nTreesEval=1;          % for top speed set nTreesEval=1
model.opts.nThreads=4;            % max number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



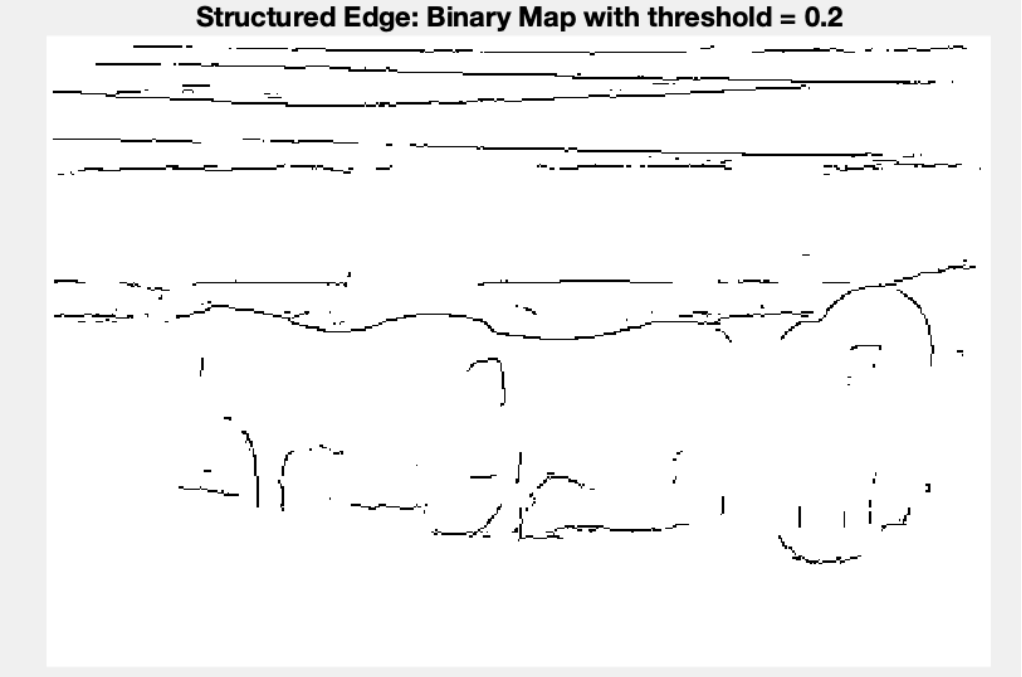
Structured Map: Binary map using threshold = 0.05



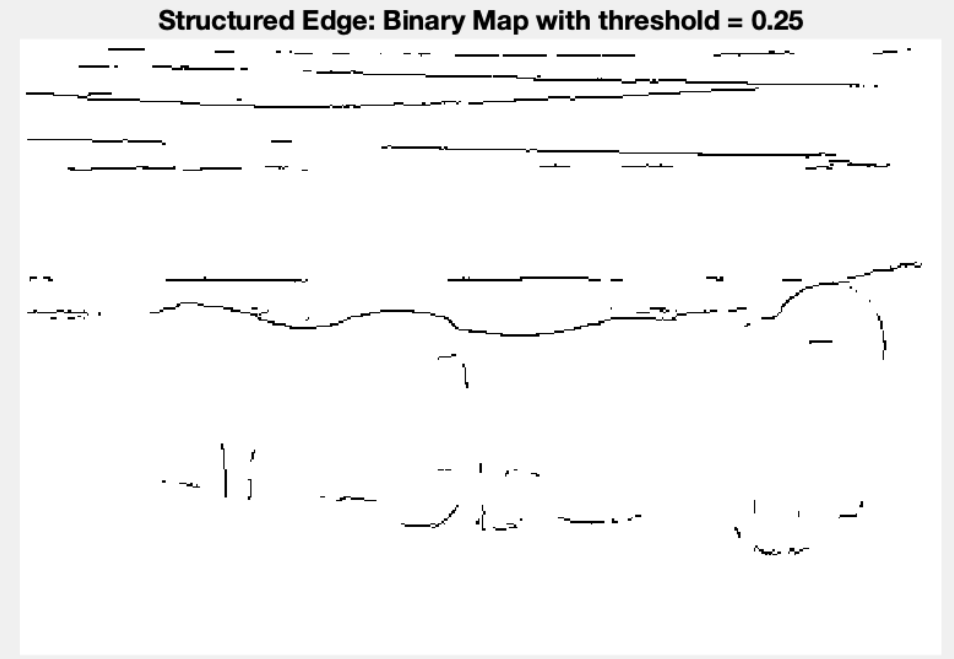
Structured Map: Binary map using threshold = 0.10



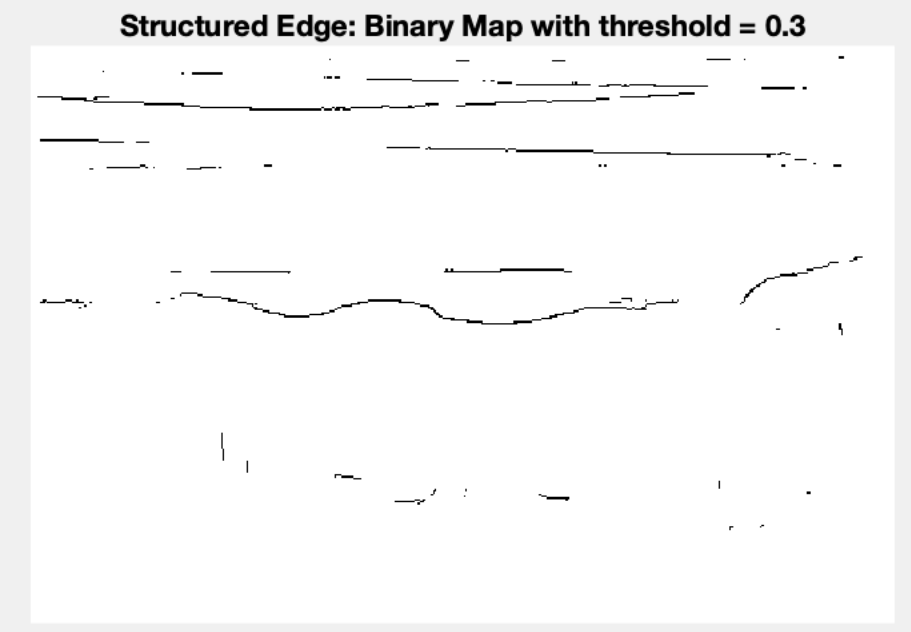
Structured Map: Binary map using threshold = 0.15



Structured Map: Binary map using threshold = 0.20



Structured Map: Binary map using threshold = 0.25



Structured Map: Binary map using threshold = 0.30

IV. Discussion

I have applied the SE detector to *Tiger* and *Pig* images and we can see the results above. The model used training structured labels with the random forest and decision tree. There were 2 main challenges faced: the output spaces were complex and were

of high dimensions, the information gained from the labels wasn't always well defined and stated and hence it was difficult to train the decision trees. However, the output spaces were reduced by using Intermediate mapping.

The parameters chosen were changed for me to have a better understanding of what the code does and how these parameters affect the data and the output. I referred the paper from Microsoft on SE to give me idea of the various parameters used in the model. The parameters I used to change and observe output changes were:

- Multiscale – it is used for top level accuracy. I observed the changes for 0 and 1. I came to the conclusion of setting it to 1, for better accuracy of getting the edges.
- Sharpen – it is used to set top speed. It also causes sharpening of image which helps to improve accuracy. Thus, I selected this parameter as 1 when I need sharp edge and better accuracy and as 0 when I needed high speed.
- nTreesEval – it is used to set top speed. This denotes the number of decision trees. I think it's value should be kept 1 so as to help for high speed.
- nThreads – this denotes maximum number of threads for evaluation. Since this parameter mainly deals with complexity time and computation speed, I did not change this value much. It should be kept at 1.
- nms – this denotes non-maximal suppression. This is used to thin the edges. I observed that when this value was 0, it caused thick edges in the image and when it was changed to 1, there were thin edges and a better simpler edge map is created. Thus, it's value should be 1 for better accuracy.
- Threshold/probability – I tried different values like 0.05, 0.10, 0.15, 0.20, 0.25, 0.30, 0.50 for both the best and worst cases for the above parameters and observed the changes in the results shown above.

Best parameters output gives more accurate output in lesser computational time. However, at times it caused a trade-off between the two measures.
For the best threshold parameter, I infer that for the tiger and pig images, it should be 0.10-0.15 for both.

On comparison of the visual results of the Sobel detector, Canny detector and the SE detector, we can say that SE proves to be more robust, less noisy, simple and clean output and also has very less computation time. The edges are very sharp in SE and also simple clear output compared to Sobel or Canny, wherein the edges are thicker or complex or look false-positive. Especially because of the texture like grass in the background of tiger or the fencing in the background in the pig image, various high frequency components are misunderstood to be edges while they aren't. Thus, SE gives better accuracy.

The binary edge map from SE is quite thicker. This results in coverage of less details of the edges as compared to Canny edge detector that give a very fine edge detection due to the use of NMS and double thresholding post Canny edge function which can be seen in the figures above. However, if we look at the probability edge map on the other side, for SE we can differentiate between objects based on the probability edge map which is something not possible in a Canny edge detector output.

(d) Performance Evaluation

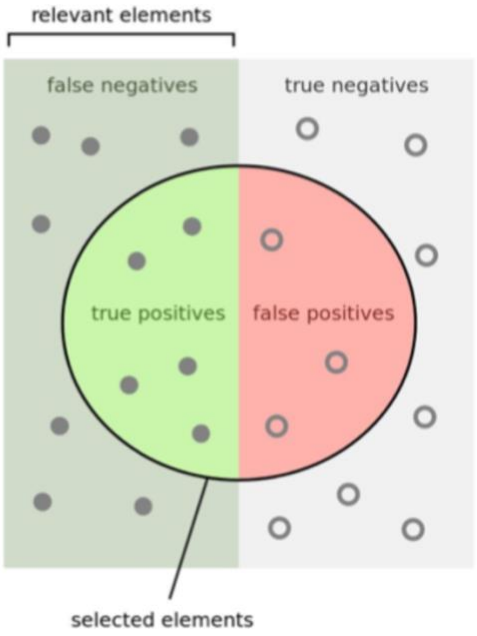
I. Abstract and Motivation

Perform quantitative comparison between different edge maps obtained by different edge detectors. The ultimate goal of edge detection is to enable the machine to generate contours of priority to human being. For this reason, we need the edge map provided by human (called the ground truth) to evaluate the quality of a machine-generated edge map. However, different people may have different opinions about important edge in an image. To handle the opinion diversity, it is typical to take the mean of a certain performance measure with respect to each ground truth, e.g. the mean precision, the mean recall, etc.

To evaluate the performance of an edge map, we need to identify the error. All pixels in an edge map belong to one of the following four classes:

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.

- True positive: Edge pixels in the edge map coincide with edge pixels in the ground truth.
- True negative: Non-edge pixels in the edge map coincide with non-edge pixels in the ground truth.
- False positive (false alarm): Edge pixels in the edge map correspond to the non-edge pixels in the ground truth.
- False negative (missing): Non-edge pixels in the edge map correspond to the true edge pixels in the ground truth.



Clearly, pixels in (1) and (2) are correct ones while those in (3) and (4) are error pixels of two different types to be evaluated. The performance of an edge detection algorithm can be measured using the F measure, which is a function of the precision and the recall. F measure is needed to be calculated because if we change the threshold for the true or false prediction then we do not get a good recall and precision at the same time. There is some balance always such that when the recall increases the precision decreases, and vice versa. Hence, we require F measure. Higher the F measure, better is the result.

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

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

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

- Recall = True Positive / (True Positive + False Negative)
- Precision = True Positive / (True Positive + False Positive)
- Mean recall and mean precision (based on each groundtruth)
- F-measure:

$$F_1 = 2 \cdot \frac{1}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

One can make the precision higher by decreasing the threshold in deriving the binary edge map. However, this will result in a lower recall. Generally, we need to consider both precision and recall at the same time and a metric called the F measure is developed for this purpose. A higher F measure implies a better edge detector.

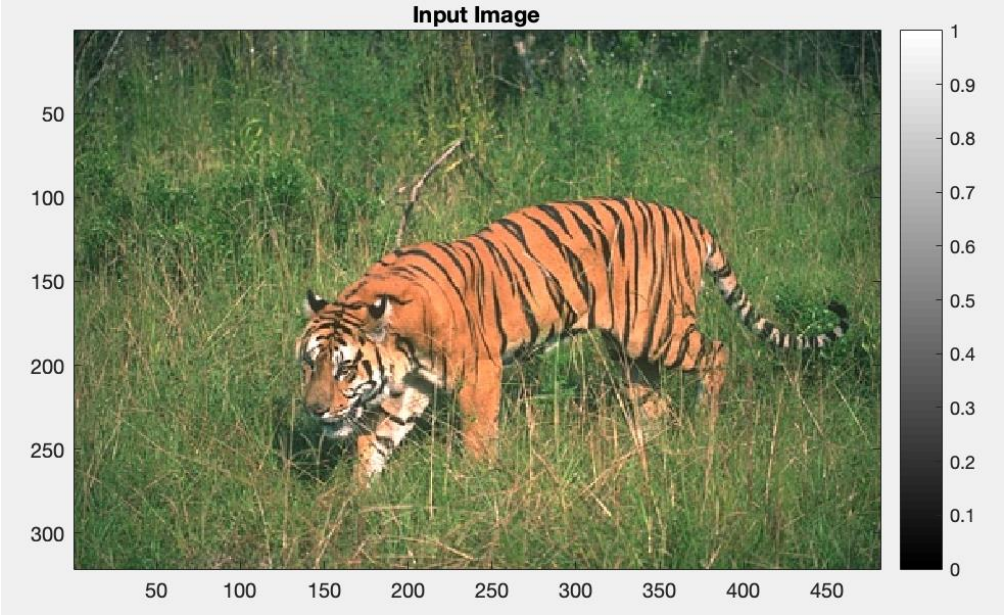
I have performed the calculation for precision, recall and F measure for all the 3 detectors – Sobel, Canny, Structured Edge.

II. Approach and Procedure

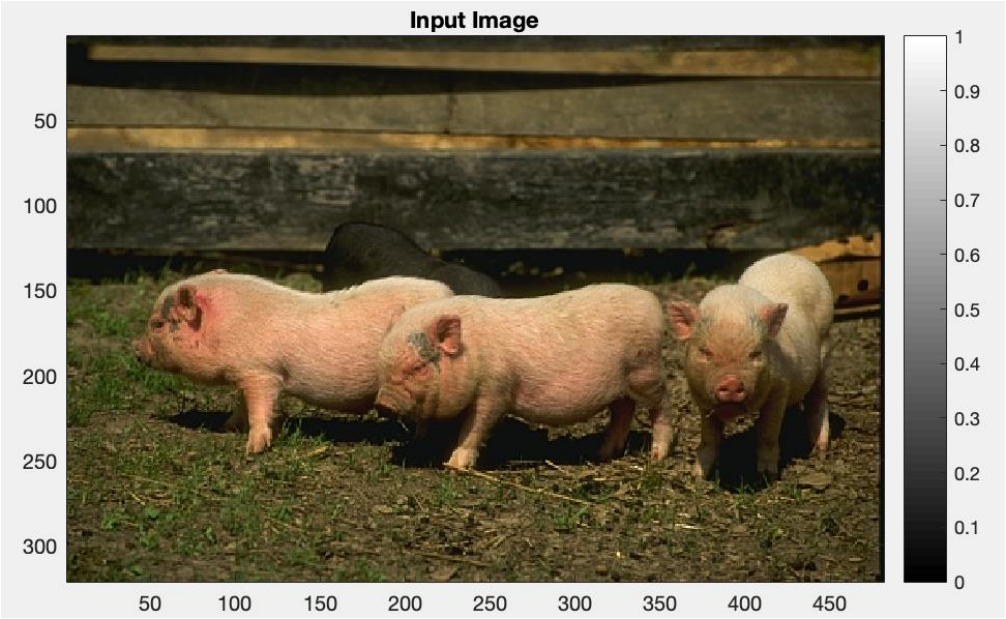
I have done my coding using the online source codes (released MATLAB toolbox: https://github.com/pdollar/edges).
(Source code: https://pdollar.github.io/toolbox/)

In MATLAB, performance evaluation is done by:
[thrs, cntR, sumR, cntP, sumP, V] = edgesEvalImg(BinaryMap, ground truths .mat file, varagin);
where,
thrs - the threshold values
P = cntP./sumP
R = cntR./sumR
F = 2 * ((P*R)/(P+R))

III. Experimental Results

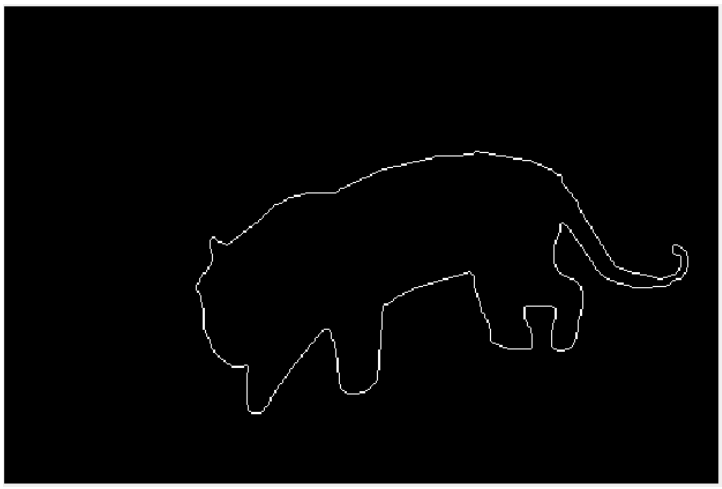


Input image tiger.raw

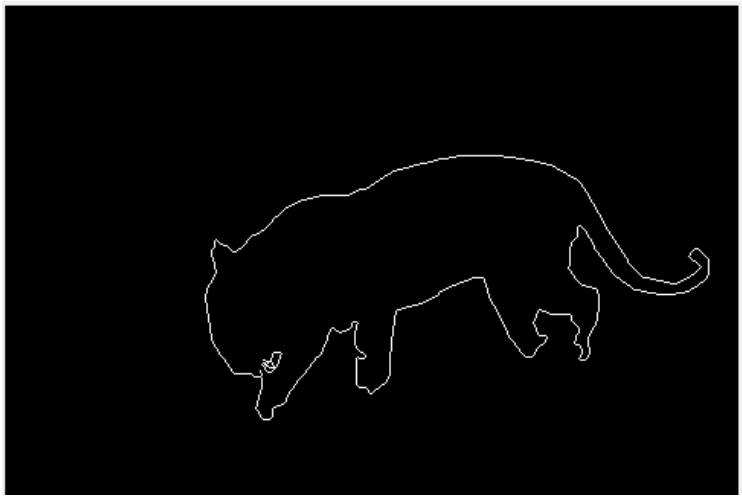


Input image pig.raw

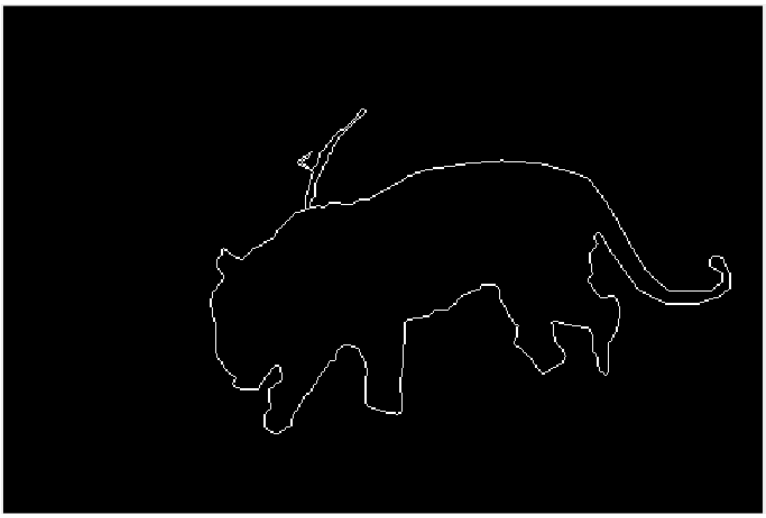
Given Ground Truth images of Tiger in .mat file



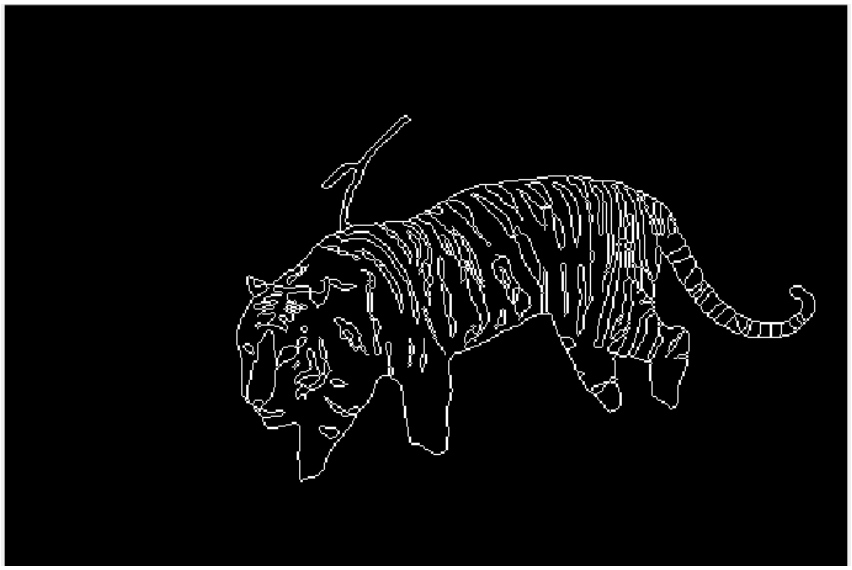
Tiger – Ground Truth Image – 1



Tiger – Ground Truth Image - 2



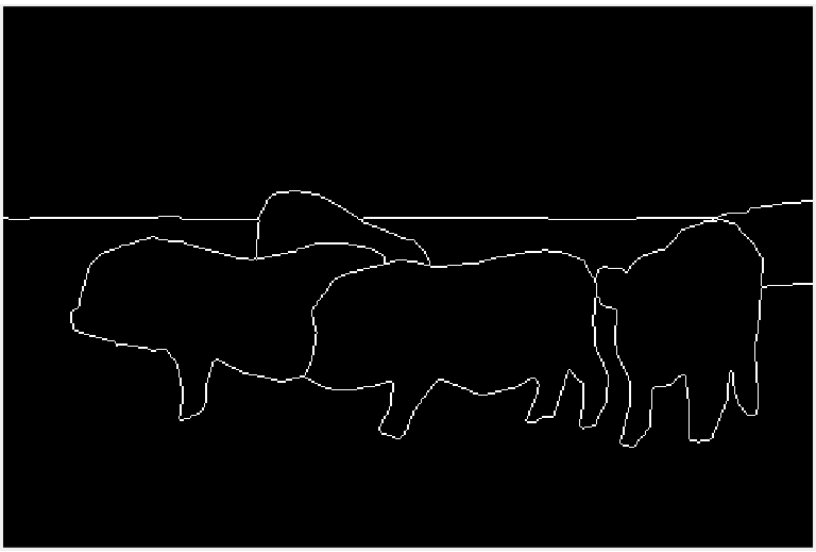
Tiger – Ground Truth Image - 3



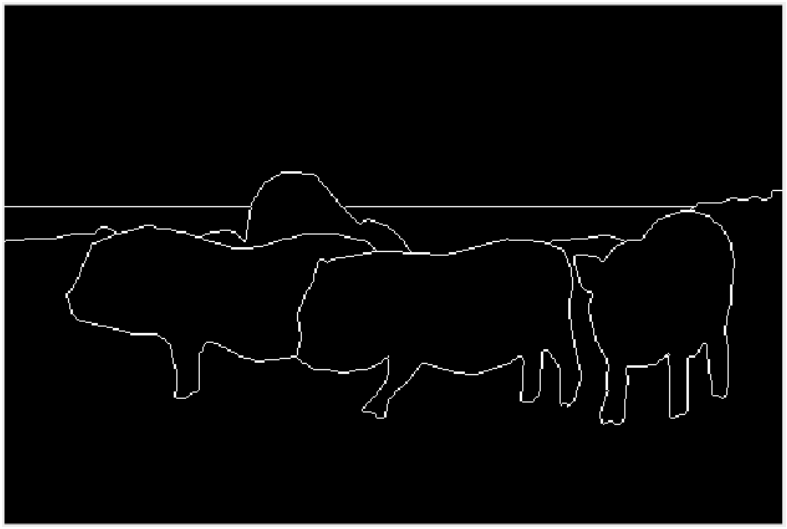
Tiger – Ground Truth Image - 4

Tiger – Ground Truth Image - 5

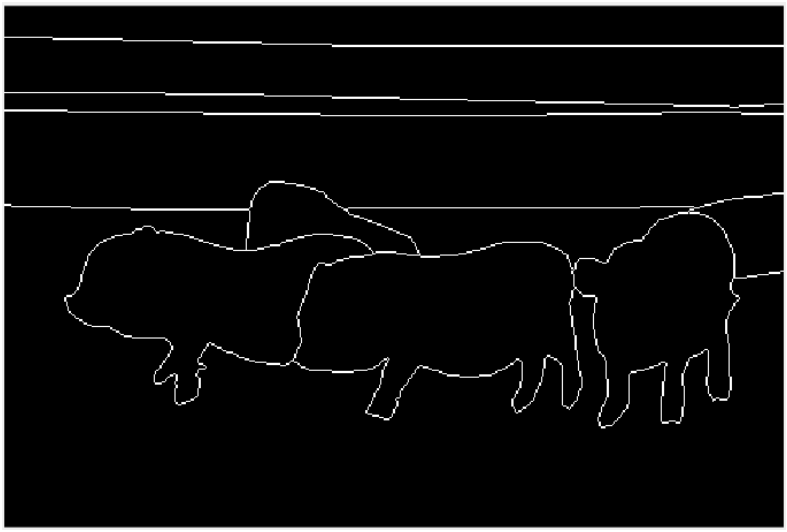
Given Ground Truth images of Pig in .mat file



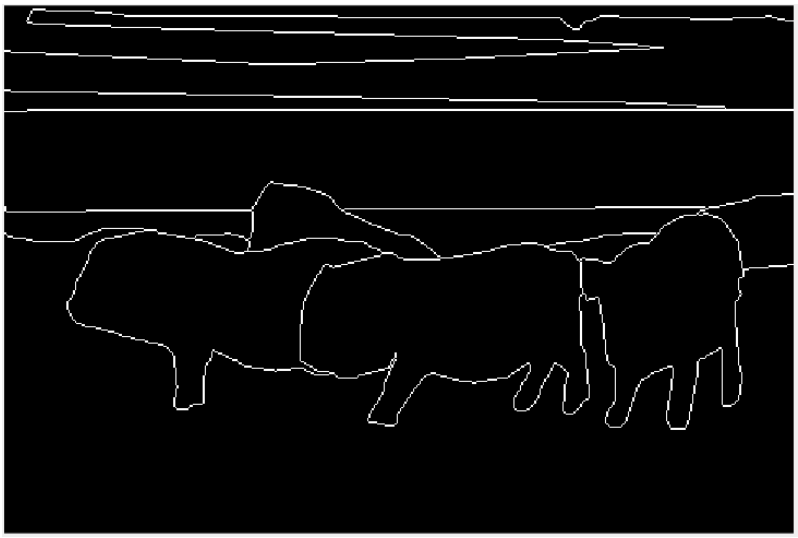
Pig – Ground Truth Image – 1



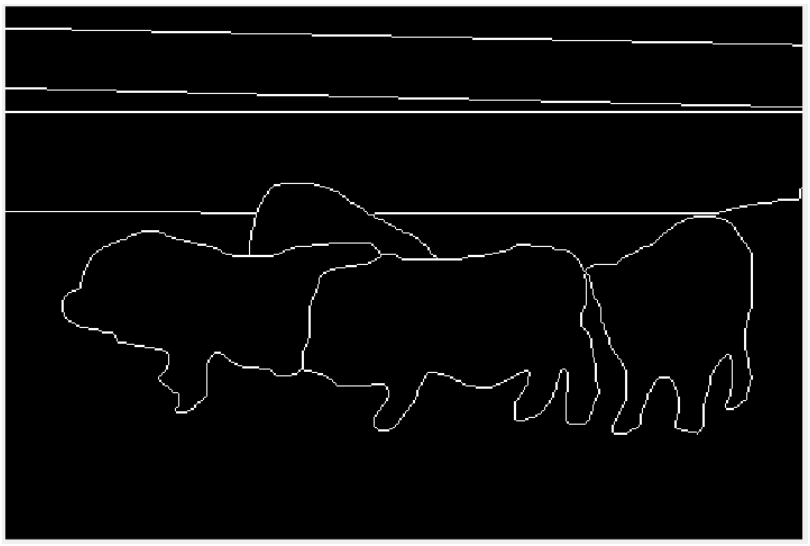
Pig – Ground Truth Image - 2



Pig – Ground Truth Image - 3



Pig – Ground Truth Image - 4



Pig – Ground Truth Image – 5

Table for Tiger image using Structured Edge Detector

Ground Truth Number	Precision	Recall	F-measure	Threshold
Ground Truth 1	0.6347	0.3456	0.4475	0.1
Ground Truth 2	0.5160	0.4460	0.4785	0.1
Ground Truth 3	0.3380	0.7075	0.4575	0.1
Ground Truth 4	0.7239	0.6849	0.7039	0.1
Ground Truth 5	0.4596	0.4227	0.4404	0.1
Mean	0.5344	0.5213	0.5278	0.1

Table for Tiger image using Canny Edge Detector

Ground Truth Number	Precision	Recall	F-measure	Threshold
Ground Truth 1	0.4911	0.2706	0.3489	0.4
Ground Truth 2	0.4262	0.2803	0.3537	0.4
Ground Truth 3	0.2160	0.6102	0.3476	0.4
Ground Truth 4	0.5011	0.5017	0.5896	0.4
Ground Truth 5	0.3713	0.3152	0.3513	0.4
Mean	0.4011	0.3953	0.3982	0.4

Table for Tiger image using Sobel Edge Detector

Ground Truth Number	Precision	Recall	F-measure	Threshold
Ground Truth 1	0.3973	0.1929	0.2237	0.9
Ground Truth 2	0.2480	0.2138	0.2349	0.9

Ground Truth 3	0.1595	0.3137	0.2312	0.9
Ground Truth 4	0.3518	0.3176	0.3516	0.9
Ground Truth 5	0.2372	0.2063	0.2301	0.9
Mean	0.2601	0.2489	0.2543	0.9

Table for Pig image using Structured Edge Detector

Ground Truth Number	Precision	Recall	F-measure	Threshold
Ground Truth 1	0.2683	0.7985	0.4014	0.1
Ground Truth 2	0.2988	0.7635	0.4295	0.1
Ground Truth 3	0.4976	0.7940	0.6118	0.1
Ground Truth 4	0.7395	0.8010	0.7690	0.1
Ground Truth 5	0.5188	0.7852	0.6248	0.1
Mean	0.4646	0.7884	0.5847	0.1

Table for Pig image using Canny Edge Detector

Ground Truth Number	Precision	Recall	F-measure	Threshold
Ground Truth 1	0.2009	0.5111	0.3041	0.4
Ground Truth 2	0.2058	0.5027	0.3128	0.4
Ground Truth 3	0.3257	0.5322	0.4929	0.4
Ground Truth 4	0.4491	0.5382	0.4723	0.4
Ground Truth 5	0.3972	0.5197	0.4313	0.4
Mean	0.3277	0.5208	0.4023	0.4

Table for Pig image using Sobel Edge Detector

Ground Truth Number	Precision	Recall	F-measure	Threshold
Ground Truth 1	0.1548	0.3541	0.1917	0.9
Ground Truth 2	0.1727	0.3129	0.1943	0.9
Ground Truth 3	0.2316	0.3528	0.2654	0.9
Ground Truth 4	0.2617	0.3724	0.3421	0.9
Ground Truth 5	0.2410	0.3371	0.3124	0.9
Mean	0.2124	0.3459	0.2632	0.9

IV. Discussion

The ground truth edge maps of tiger and pig images are shown above. Using these, I have evaluated the quality of the edge maps and can compare the performance of Sobel edge detector, Canny edge detector and Structured edge detector.

Precision is basically the agreement among several dimensions. Higher the precision, lower is the difference between the ground truth and the predicted truth. Recall is basically the sensitivity. It denotes the probability that the edge is correctly detected by the algorithm. F-measure is a combination of precision and recall. This gives a better understanding of the model or detector as it consists of both the parameters. Higher the F measure, better is the model. We calculate the precision and recall for each ground truth (saved in .mat format) separately using the function provided by the SE software package and, then, compute the mean precision and the mean recall. Finally, we calculate the F measure for each generated edge map based on the mean precision and the mean recall. The table showing the precision and recall for each ground truth and their means and F measure is shown for both the images tiger and pig and for all the detectors used – Sobel, Canny, SE. F measure for Sobel is almost half of SE. Even the Canny F measure is lower, however, by a small value. Thus, the performance of the detectors based on these values is that SE is supreme and has better outputs as we had compared and told above as well for the reasons stated.

The F measure is image dependent and can be proved from the above results in the tables. It is easier for the pig image to get a higher F measure than the tiger image. This is because, tiger image has more high frequency components than the pig image that is grass in the image. Also, the stripes on the tiger's body are more textures as compared to the pig. Hence, we observe in the output that there are more noise components in tiger image than the pig image. The pig image is made of simpler edges. Thus F-Measure depends on the High frequency components present in the image and also based on the method of Edge Detection.

To understand F measure, we can say that F-Measure is the harmonic mean of Precision and Recall. This is because Harmonic Mean is best when it comes to ratios. There exists an inverse relation between precision and recall. If one value is high and the other value is low, it badly affects the F-Measure Score Value. To get a Maximum F-Measure, both Precision and Recall values need to be the same.

Task: If the sum of precision and recall is a constant, show that the F measure reaches the maximum when precision is equal to recall.
If Precision and Recall add up to a constant, Precision (P) + Recall (R) = Constant (C)

$R = C - P$
Substituting in formula for F measure
$F = 2 * (P * R) / (P + R)$
Therefore, $F = 2*(P) (C - P) / C$.
To find the maximum, find the first derivative with respect to P and equate it to 0
If, $F' = 0$
$F = (2 * P * C - 2 * P * P) / C$
F' is $2C - 4P = 0$
Hence, $P = C/2$
Therefore, $R = C/2$ and $R = P$
Hence, we will get the maximum F-score when Recall = Precision.

Problem 2: Digital Half-toning

(a) Dithering

I. Abstract and Motivation

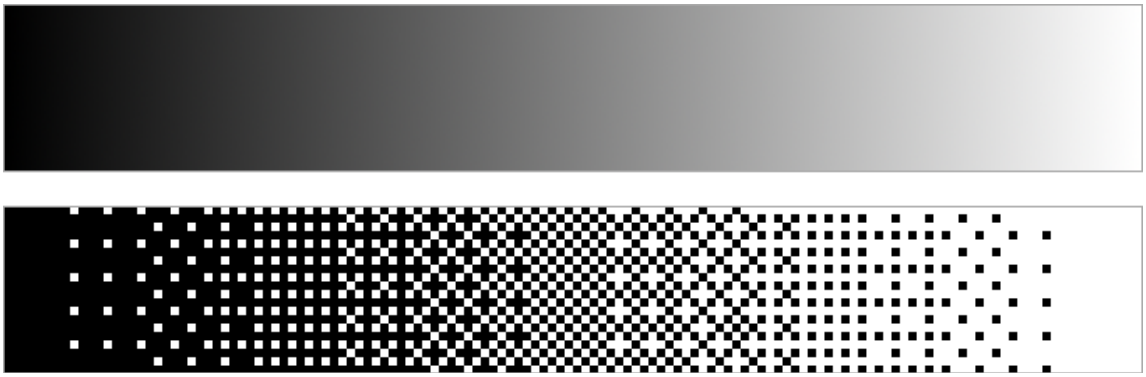
Half-toning is used in the printing industry to produce visual reproductions (render) to an image on the newspapers, magazines, PDF files, fax that is printed with binary inks, white ink (background) and black ink (foreground). Such gray scale images are represented by black i.e. 0's and white i.e. 1's. Human perception is like continuous tone because of high resolution dots in dpi measurement like 600 dip, 900 dpi. It is difficult for human beings to differentiate two dots very close to each other. Thus, high density of dots in a local cell can be rendered as a gray scale pixel. Digital Halftoning is mainly done because of the property of human eyes acting as a Low Pass filter which makes the image of black and white dots appear as a gray scale image as a continuous tone. Digital half-toning is about how we can represent a grayscale image with only black and white pixels. Since some printers have only black ink, we need to approximate the gray scale using only black color.

Half-toning first step is Color space transformation (from additive color space to subtractive color space) i.e. RGBW → CMYK

Digital half-toning is done by 2 methods: Dithering and Error Diffusion. In dithering, we can use random and fixed/constant thresholding and dithering matrix for converting the original input image into a binary image.

The principle of digital half-toning is to attain satisfactory color reduction and image rendering. This is achieved by the dithering method. Digital half-toning gives the illusion of continuous image with more than 2 intensity levels like grayscale image (0-255 intensity levels) by using only 2 binary pixel values i.e. 0 and 1. We need to add noise (blue noise) before quantizing or thresholding to reduce the monotonicity in the local varying region. However, it is not easy to add noise, but adopt adaptive

threshold values, so dithering method uses multiple thresholds. In fixed thresholding method, when the pixel value of the original input image is less than the fixed or constant set threshold, the output image pixel is a black dot. In random thresholding method, when the pixel value of the original input image is less than the set threshold, the output image pixel is a black dot.



(Source: http://alex-charlton.com/posts/Dithering_on_the_GPU/)

In Dithering matrix method, an index matrix is created and placed on the original image and the output pixel value is thresholded depending on the pixel values of output image and the Bayer matrix. A variety of Bayer's matrices are used like 2x2, 4x4, 8x8, 16x16, 32x32. When the pixel value in the original image is greater than the Bayer matrix value, a black dot is placed at that value.

II. Approach and Procedure

- Binarize grayscale → compare with threshold

$$output(i,j) = \begin{cases} 0 & 0 \le input(i,j) < thre(i,j) \\ 255 & thre(i,j) \le input(i,j) < 256 \end{cases}$$

1. Fixed thresholding

For performing fixed thresholding, a threshold of intensity 127 is used to covert gray scale image into binary image. Each pixel gets the value 0 when the pixel value is smaller than the threshold and mapped to 255 elsewise. Say the original input image is denoted as F(i,j) and the output image is denoted as G(i,j) and T/const is the constant or fixed threshold.

$$G(i,j) = \begin{cases} 0 & \text{if } 0 \le F(i,j) < T \\ 255 & \text{if } T \le F(i,j) < 256 \end{cases}$$

- **Constant threshold:**

$$op(i,j) = \begin{cases} 0 & 0 \le ip(i,j) < const \\ 255 & const \le ip(i,j) < 256 \end{cases}$$

pros: fast, low memory
cons: loss of info

This method is fast and requires lesser space in memory, however, it causes loss of information.

2. Random thresholding

In order to break the monotones in the result from the fixed thresholding, we may use a random threshold. Say the original input image is denoted as F(i,j) and the output image is denoted as G(i,j). Random function is random number generator for a particular distribution generally uniform distribution. Since the threshold is random, output will not be the same every time.

The algorithm can be described as:

- For each pixel, generate a random number in the range 0 ~ 255, so called $rand(i,j)$
- Compare the pixel value with $rand(i,j)$. If it is greater, then map it to 255; otherwise, map it to 0, i.e.

$$G(i,j) = \begin{cases} 0 & if\ 0 \le F(i,j) < rand(i,j) \\ 255 & if\ rand(i,j) \le F(i,j) < 256 \end{cases}$$

- **Random threshold:**

$$op(i,j) = \begin{cases} 0 & 0 \le ip(i,j) < rand(i,j) \\ 255 & rand(i,j) \le ip(i,j) < 256 \end{cases}$$

pros: include more details
cons: introduce random noise

A choice of random threshold between 0 to 255 could be uniformly distributed random variable generated for every pixel. Also, as the threshold is random, the output will not be the same every time. This method implementation is better than fixed thresholding as it includes more details. The con is that it introduces random

noise in the image. We use the random thresholds instead of fixed threshold values in order to reduce the quantization errors.

Algorithm implemented in C++:

- Read the input image "bridge.raw" whose dimensions are height_size, width_size, BytesPerPixel
- For every pixel, threshold the image using random and fixed thresholding.
- For each pixel, generate a random number in the range 0~255 using rand ()
- Compare each pixel value/intensity with the random number generated by rand ()
- If pixel value is less than the random number, map to 0 else map to 255
- For the fixed method, use 127 instead of random number and repeat the method above
- Write the computed image data array on output.raw file using the fwrite() function

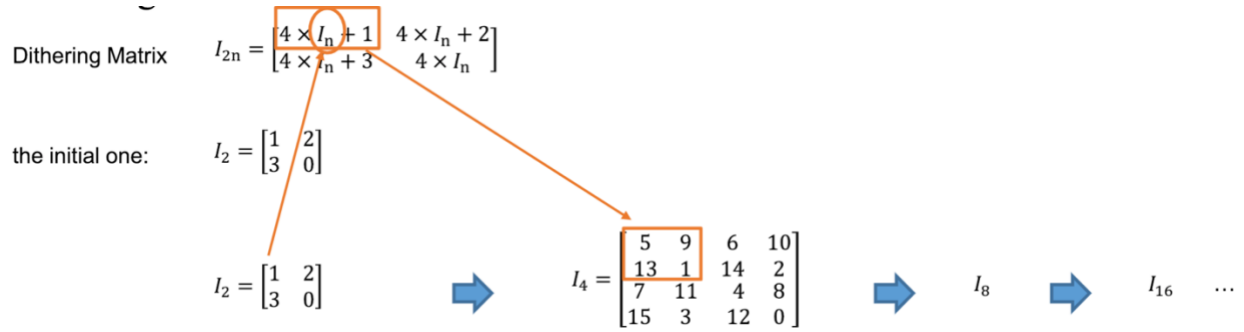
3. Dithering Matrix

In order to reduce the noisy outputs obtained by the thresholding method, a smarter method called dithering is used. This is also known as ordered dithering. This algorithm implements dithering by using pre-calculated threshold from different ordered matrices which store the pattern of thresholds. Dithering parameters are specified using an Index matrix of 2x2 size. It is given below,

$$I_2(i,j) = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix}$$

This value of the index matrix is used to calculate the likelihood of a dot or pixel to be turned on. For example, 0 indicates the pixel most likely to be turned on, whereas 3 indicates the pixel that is least likely to be turned on. The generalized formula of the Bayer Index Matrix (using recursion) is given below,

$$I_{2n}(i,j) = \begin{bmatrix} 4 * I_n(x,y) + 1 & 4 * I_n(x,y) + 2 \\ 4 * I_n(x,y) + 3 & 4 * I_n(x,y) \end{bmatrix}$$



This is used to generate the threshold matrix which is used to compare with the input image. It is used to calculate the 4x4, 8x8, 16x16 and 32x32 matrices from the 2x2 matrix. The index matrix can then be transformed into a threshold matrix T for an input gray image with normalized pixel values (dynamic range between 0 and 255) by the formula below,

$$\text{threshold matrix} \quad T(x,y) = \frac{I(x,y) + 0.5}{N^2} \times 255$$

where, N^2 is the number of pixels in the matrix. Multiplication with 255 is done because of the normalization. As the input image matrix is very large than threshold matrix, the threshold matrix is used as a mask on the entire input image and repeated periodically across the full image. The final thresholding of every grayscale image pixel is given below,

$$output(i,j) = \begin{cases} 0 & 0 \le input(i,j) \le T(i\ mod N, j\ mod N) \\ 255 & T(i\ mod N, j\ mod N) < input(i,j) < 256 \end{cases}$$

Here, input(i,j) is the normalized pixel intensities. But the output is large, so we should repeat the threshold. We need to use some operation for compression. Hence, in the code I have used the Modular operation so as to save the memory.

Finally, the outputs are compared for different threshold matrices like I2, I8, I32.

Dithering-4 intensity levels is similar to 2-gray scale such that each gray scale intensity levels are mapped to 4 unique intensity levels instead of 2 binary levels. The threshold values are given below:

$$G(i,j) = \begin{cases} 0, & if\ 0 \le F(i,j) \le \dfrac{T(i\%N, j\%N)}{2} \\ 85, & if\ \dfrac{T(i\%N, j\%N)}{2} < F(i,j) \le T(i\%N, j\%N) \\ 170, & if\ T(i\%N, j\%N) < F(i,j) \le 127 + \dfrac{T(i\%N, j\%N)}{2} \\ 255, & if\ 127 + \dfrac{T(i\%N, j\%N)}{2} < F(i,j) \le 255 \end{cases}$$

Algorithm implemented in C++:

- Read the input image "bridge.raw" whose dimensions are height_size, width_size, BytesPerPixel using fread () function
- The input image is normalized for values between 0-1 to implement the dithering method
- 2x2 index matrix is created
- Using recursive functions, find dithering matrices 4x4, 8x8, 16x16, 32x32 for window sizes/ different indexes 4, 8, 16 and 32 using the formula above
- Obtain the threshold matrices for I2, I4, I8, I16, I32
- For every pixel in the image, the input and threshold matrix are compared and set the pixel to 0 or 1 accordingly using decision rule
- Write the computed image data array on output.raw file using the fwrite() function

4. Experimental Results

The I2, I4, I8 matrices generated using the Bayer index matrix formula are shown below:

1	2
3	0

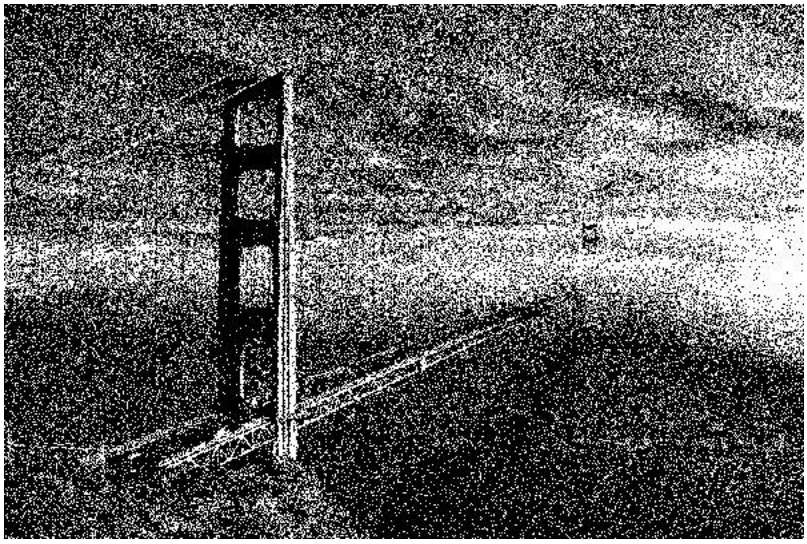
5	9	6	10
13	1	14	2
7	11	4	8
15	3	12	0

21	37	25	41	22	38	26	42
53	5	57	9	54	6	58	10
29	45	17	33	30	46	18	34
61	13	49	1	62	14	50	2
23	39	27	43	20	36	24	40
55	7	59	11	52	4	56	8
31	47	19	35	28	44	16	32
63	15	51	3	60	12	48	0

2×2 4×4 8×8

Similarly, I16 and I32 matrices were created using the above formula and matrices, however, the matrix is too large, hence I did not include it in the report. The matrix is produced and printed when the code is run using cout and can be checked there.

Input Image bridge.raw

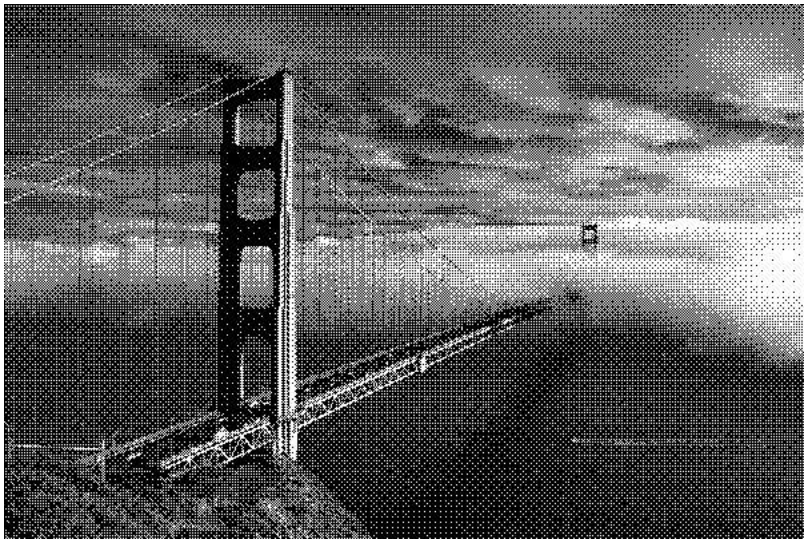


Output Image using random thresholding

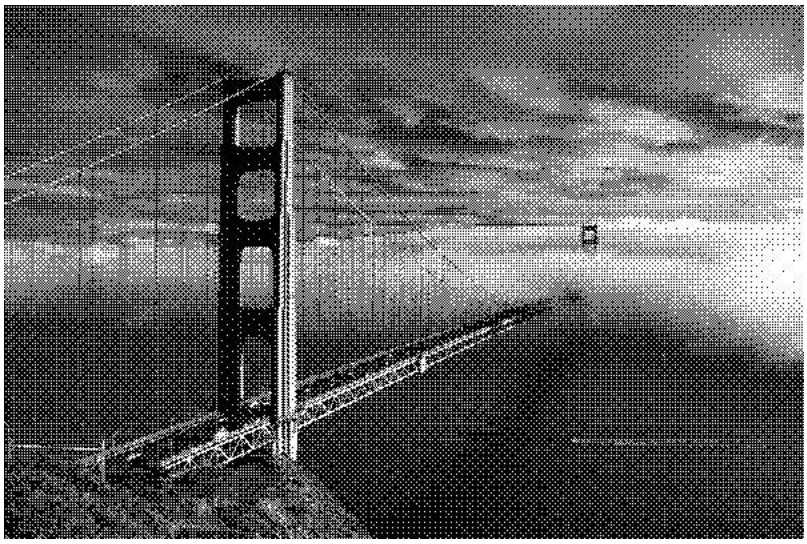
Output Image using fixed thresholding

Output Image using 2x2 I2 index dithering matrix

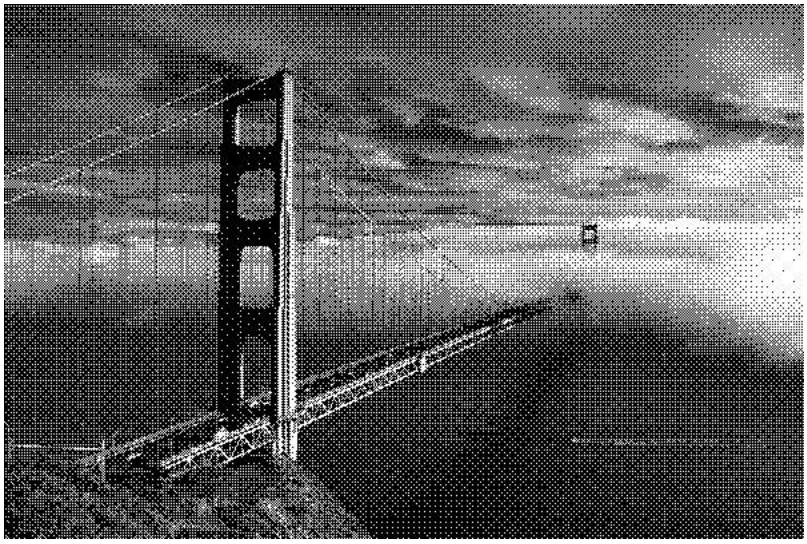
Output Image using 4x4 I4 index dithering matrix



Output Image using 8x8 I8 index dithering matrix



Output Image using 16x16 I16 index dithering matrix



Output Image using 32x32 I32 index dithering matrix

5. *Discussion*

The dithering outputs are very good as compared to the random and fixed thresholding. The features like the lines of the bridge, the clouds, the sea and the boat sailing are well distinguished in the dithering output.

Similarly, 8x8 index matrix dithering is better compared to 4x4 and 2x2 index matrices and resembles the input image more closely. The I2 and I4 dithered images have lots of dots in them because of the density difference between the black dots. The I16 and I32 dithered images are also images which give a better output and close to the gray input image, however, we do not see much significant change in these compared to the 8x8 dithered image. Hence, in practice I8 matrix is used widely because of its unique pattern. We can thus, make a conclusion that the output caused by dithering using I8 matrix gives a good output and applying higher index dithered matrix like I32 would not cause much change and stays the same.

The fixed threshold image has only two gray levels and hence can be seen as a black and white image. It causes massive loss of information. In order to reduce the false contours in the fixed threshold image, random thresholding is used. In random thresholding, the image has only two levels but looks like a gray image from a distance. However, since it uses Gaussian or uniform distribution, it results into distorted output.

In order to reduce the noise content in the image of random thresholding, dithering is used. Bayer matrix of varied sizes distribute the quantization error in different

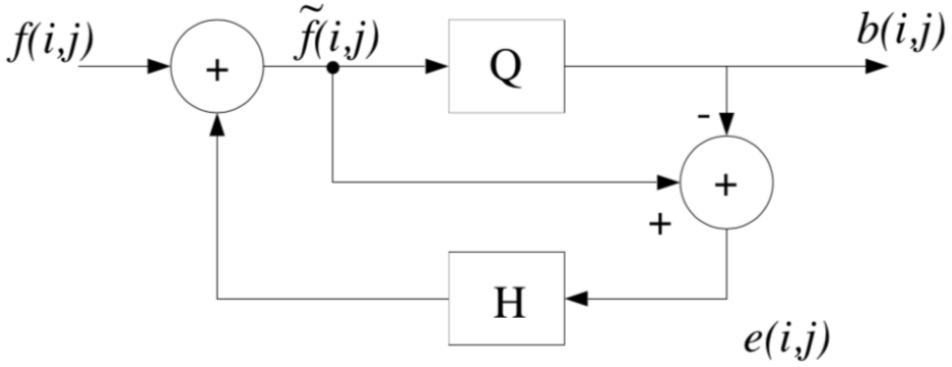
manner for each pixel value. As the size/order of the index matrix increases, the probability of the black pixel being turned on increases and the density of dots keeps increasing. Halftoning using dithering matrix is used a lot however, it possesses undesired patterns which if looked upon closely, it shows distortion.

(b) Error Diffusion

I. Abstract and Motivation

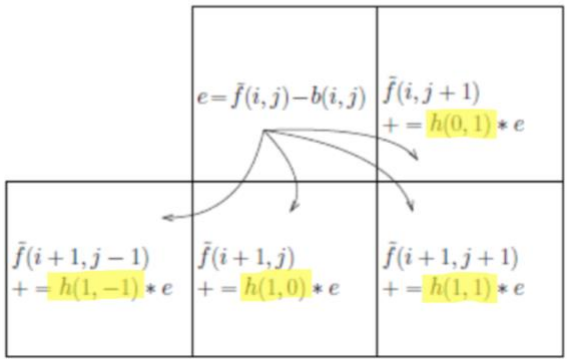
Error diffusion is another method of Half-toning technique where the error of the present pixel is distributed/diffused to the neighboring pixels that have not been processed yet basically the pixels after the center pixel while scanning the image. The pixels before the center pixel are not disturbed. Remove the DC low frequency noise and diffuse the error to the neighbor and update the neighbor pixel values. This helps to maintain the average intensity of the binary image close to gray scale image. This is also called as area diffusion as the effect of the algorithm at one pixel location affects the pixels at other locations. Here the error is calculated by the algorithm at each pixel and it is diffused on the upcoming pixels.

Error diffusion is mainly used to convert multi-level image into binary image. This technique enhances the edges in the image i.e. the lines of the bridge in the output obtained by error diffusion method are more evident than in the output of the dithering or thresholding method. It enhances the small details and features in the image. Here the quantization error is diffused in forward pixels that are yet to be processed. In this way, we produce better local intensity and enhance edges.



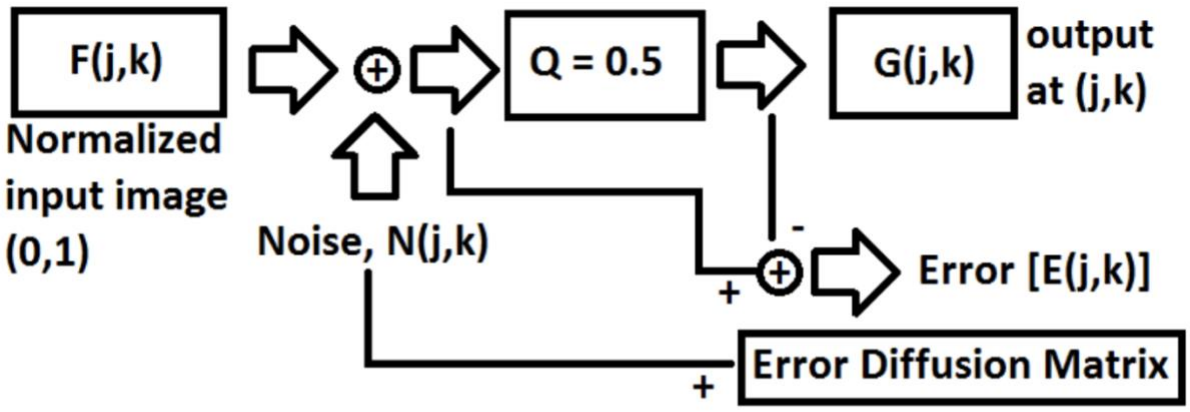
(Source: https://engineering.purdue.edu/~bouman/ece637/notes/pdf/Halftoning.pdf)

- Initialize $\tilde{f}(i,j) \leftarrow f(i,j)$
- For each pixel:
  - Binarize $$b(i,j) = \begin{cases} 255 & \text{if } \tilde{f}(i,j) > T \\ 0 & \text{otherwise} \end{cases}$$
  - Diffuse error:



where, h is the weighted factor multiplied with error and gives the output value for the pixel value. So we traverse over every pixel value in the image and calculate in a similar manner.

There are 3 different error diffusion methods: Floyd-Steinberg (FS), Jarvis, Judice and Ninke (JJN) and Stucki. The distribution of error is different for each type. The error diffusion algorithm is stated below in the diagram,

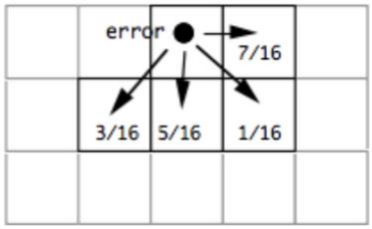


Q with threshold = 0.5
if F(j,k) >= T then G(j,k) is 255 (white)
if F(j,k) < T then G(j,k) is 0 (black)

In this algorithm, the input pixel F(j,k) is modified by adding some noise first N(j,k) and then it undergoes processing with past quantization error. This modified pixel is named as G(j,k). This modified pixel is then quantized to a binary value by using a quantizer Q with threshold T. The error generated is given as E(j,k) = F(j,k)− G(j,k). This error is then diffused to the future pixels by using several filters and constructing the error diffusion matrix namely Floyd Steinberg, Jarvis, Judice and Ninke, Stucki. Also, in each case we scan the pixels in 2 different manners like raster parsing and serpentine parsing and compare the results from both. The filters in each of these methods is given as follows:

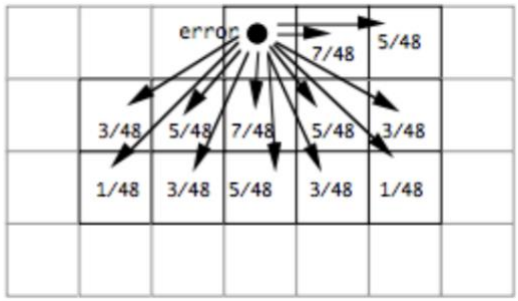
- **Floyd-Steinberg**

$$\frac{1}{16}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 7 \\ 3 & 5 & 1 \end{bmatrix}$$



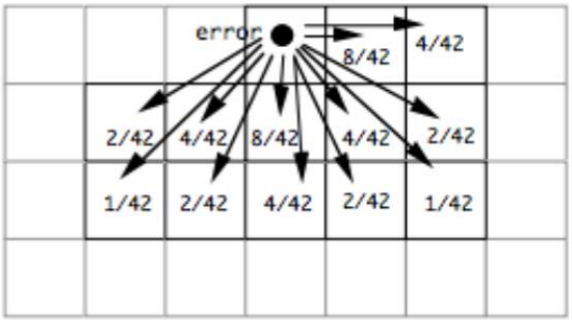
- **JJN**

$$\frac{1}{48}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{bmatrix}$$



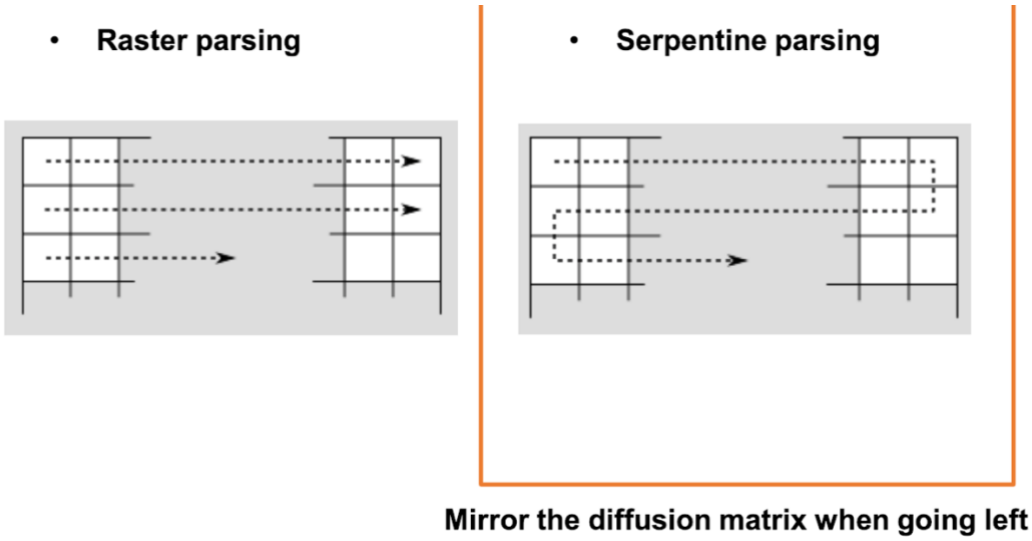
- **Stucki**

$$\frac{1}{42}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix}$$



Since all the filter coefficients sum to one, the local average value of the output image is approximately equal to the local average value of the input image. The error diffusion methods implemented make use of serpentine and raster scanning. The serpentine method helps in diffusing the error evenly in both the sides of the image rather than it accumulating in one corner.

In serpentine scanning, for even rows, the scanning is done from left to right whereas for odd rows it is done from right to left. Hence, the masks are also changed accordingly using mirroring effect for scanning odd rows. However, in the raster scanning method, the scanning of pixels is done from left to right only column wise and then moving to the next row and repeating the same pattern. We scan through the next pixels only the ones who are unvisited and perform the matrix operation.



II. Approach and Procedure

Each pixel in the input image is thresholded by using some thresholding value which is say, 127. The error is then calculated by subtracting the original value of the pixel

with the new value of the pixel. The error diffusion is then performed to the neighboring pixels of the center pixel.

$$b(i,j) = \begin{cases} 255, & if\ \tilde{f}(i,j) > T \\ 0, & otherwise \end{cases}$$

$$e(i,j) = \tilde{f}(i,j) - b(i,j)$$

$$\tilde{f}(i,j) = f(i,j) + \sum_{k,l} h(k,l)e(i-k, j-l)$$

The 3 error diffusion matrices for serpentine scanning while scanning from left to right,

A. Floyd Steinberg's error diffusion matrix,

$$h = \frac{1}{16}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 7 \\ 3 & 5 & 1 \end{bmatrix}$$

B. Jarvis, Judice and Ninke (JJN) error diffusion matrix,

$$h = \frac{1}{48}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{bmatrix}$$

C. Stucki's error diffusion matrix,

$$h = \frac{1}{42}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix}$$

The 3 error diffusion matrices for serpentine scanning while scanning from right to left,

A. Floyd Steinberg's error diffusion matrix,

$$h = \frac{1}{16}\begin{bmatrix} 0 & 0 & 0 \\ 7 & 0 & 0 \\ 1 & 5 & 3 \end{bmatrix}$$

B. Jarvis, Judice and Ninke (JJN) error diffusion matrix,

$$h = \frac{1}{48}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 7 & 0 & 0 & 0 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{bmatrix}$$

C. Stucki's error diffusion matrix,

$$h = \frac{1}{42}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 4 & 8 & 0 & 0 & 0 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \end{bmatrix}$$

Algorithm implemented in C++:

- Read the input image “bridge.raw” whose dimensions are height_size, width_size, BytesPerPixel using fread () function
- The input image is normalized for values between 0-1
- Binarize the temporary image which has pixel values transferred from the original image
- Calculate the error diffusion matrix using Floyd-Steinberg’s (FS) matrix with and without serpentine scanning
- For raster scanning, assign a value to the output pixel by using thresholding. If the value of input image <= 0.5 or 127 depends on normalizing, assign a value of 255 or else assign a value of 0
- For serpentine scanning, if the row is even, check by using the modular 2 operation, assign a value to the output pixel by using thresholding. If the value of input image <= 0.5 or 127 depends on normalizing, assign a value of 255 or else assign a value of 0
- Calculate the error by subtracting new value assigned (binarized value) to the pixel after thresholding from the old value (error = output – input)
- Diffuse the error to the future neighboring pixels as shown in the matrices above
- Repeat the procedure for scanning of odd rows when modular 2 operation fails
- Repeat the procedure with Jarvis, Judice and Ninke (JJN) and Stucki matrices
- Write the computed image data array on output.raw file using the fwrite() function

III. Experimental Results

Input Image bridge.raw

Output Image using Floyd- Steinberg's error diffusion matrix without serpentine scanning

Output Image using Floyd- Steinberg's error diffusion matrix with serpentine scanning

Output Image using Jarvis, Judice and Ninke's error diffusion matrix without serpentine scanning

Output Image using Jarvis, Judice and Ninke's error diffusion matrix with serpentine scanning

Output Image using Stucki's error diffusion matrix without serpentine scanning

Output Image using Stucki's error diffusion matrix with serpentine scanning

IV. Discussion

We have implemented the 3 error diffusion methods: Floyd-Steinberg (FS), Jarvis, Judice and Ninke (JJN) and Stucki. We can compare the results for these 3 methods from above, and also compare the results for the random and fixed thresholding and dithering matrix with the error diffusion method. The main challenge was to reproduce the distinct features from the given image say, the thin lines of the bridge.

Floyd-Steinberg's error diffusion method only diffuses the error to neighboring pixels. This results in very fine-grained dithering. It gives a decent output when looked at from a distance. This is the fastest as the matrix is small and hence lesser computations as FS has a 3x3 mask. It provides low contrast and is less sharp.
Jarvis, Judice and Ninke's error diffusion method diffuses the error to pixels one step further and has 5x5 mask, thus affecting a larger neighborhood. The dithering looks coarser, and the runtime is also slower. There is also no loss of information and JJN provides a sharp output.
Stucki's error diffusion method also diffuses the error to pixels one step further and has 5x5 mask, thus affecting a larger neighborhood. This method is slightly faster, and its output looks sharp and clean. I would prefer using the Stucki matrix error diffusion method as it gives very good results compared to FS and JJN as we can see in the results above.

JJN and Stucki provide a better output visually than FS because of affecting a larger neighborhood of pixels. It is difficult to distinguish between the Stucki and JJN as they provide a quite similar output, however, the weights associated are higher in

Stucki than JJN. The JJN method provides stronger edges as compared to the other methods and hence is beneficial to use when the edge information is of great importance as compared to other portions of the image like shown in the bridge example above where thin lines of the bridge are an important part of the image and can be used as a distinguishing factor for the 3 methods. FS method gives a blurred image compared to others, JJN and Stucki images are less blurry. The fine details of the image are very well preserved when Stucki matrix is used as compared to any other matrix. If we calculate and compare the PSNR values for the 3 methods, JJN and Stucki have a higher value as compared to FS.

Trying all these 3 methods, we still compare and see that artifacts do not disappear and are still present. Without the serpentine scanning, in raster scanning, we see that the error is diffused only in one direction i.e. from left to right and artifacts go on accumulating at the right-side end. Whereas, when we use serpentine scanning, the error is diffused, artifacts are distributed evenly and no accumulation occurs in any of the cases of matrices, as the filters are flipped while moving from left to right and right to left respectively. Since high frequency noise such as blue noise is added in this technique, all the methods have large amount of noise introduced in the images.

Out of all half-toning methods, error diffusion is the best method as we can observe in output as well as it is smoother and does not induce any kind of pattern like dithering matrices. I would prefer using the Error Diffusion over Dithering method as it preserves the details and patterns in the output image. The Error Diffusion method gives a better contrast performance as compared to the Dithering method of Half-toning. The Error Diffusion methods gives an output image close to the original gray scale image if looked from a distance. Hence, the error diffusion methods are used a lot in digital Half-toning. However, error diffusion methods tend to display streaking artifacts.

However, other than the methods used above, we can implement other ideas to get better results, like Sierra dithering, Atkinson, Burkes, Sierra Lite, Two-row dithering. These methods prove to be faster and simpler and clean. The Burkes error diffusion matrix much better output by removing all lower frequency components and preserving blue noise by defined matrix is a novel way. The image displays more details than JJN method. The cascade inverse halftoning algorithm works quite well as it improves performance by cascading 2 stages, as image has patterns that is rich of edges and the cascade algorithms is designed to reserve edges. Atkinson developed a dithering algorithm named as Atkinson Dithering. In this type of dithering, only a small portion of the error is propagated to the future pixels. Normally only three quarters of the error is diffused to the future pixels. By

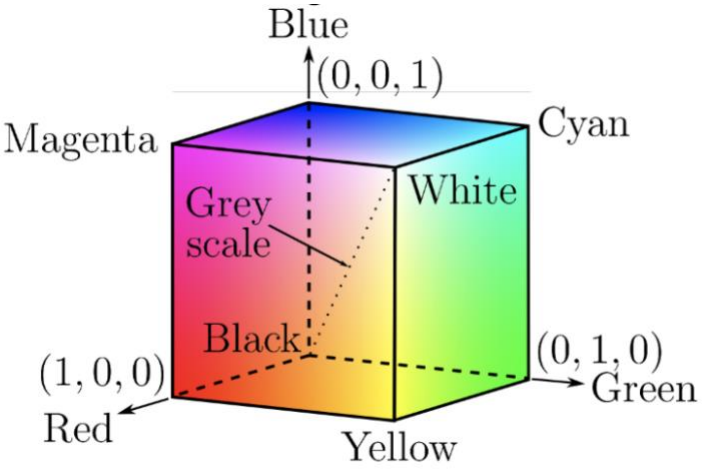
propagating only a part of the error, the speckles in the image are reduced, but the continuous dark and light portions of the image may become blur as well.

(c) Color Halftoning with Error Diffusion

I. Abstract and Motivation

The error diffusion algorithm was earlier restricted to gray scale images but in recent years has been extended to the color images. Here, we use the error diffusion method which we used in the previous task for color half-toning. There are 2 methods – Separable error diffusion and MBVQ based error diffusion.

In color images, we have 24 bits (8 bit per RGB i.e. 3 channels = 8x3 = 24) so we have 2^24 combinations of color distributed in the image which has millions of colors, hence by color half-toning we match and reduce them to only 8 colors. We apply the Floyd-Steinberg's error diffusion serpentine scanning method for the color half-toning to 3 separate channels RGB/CMY. MBVQ performs better than Separable error diffusion method.



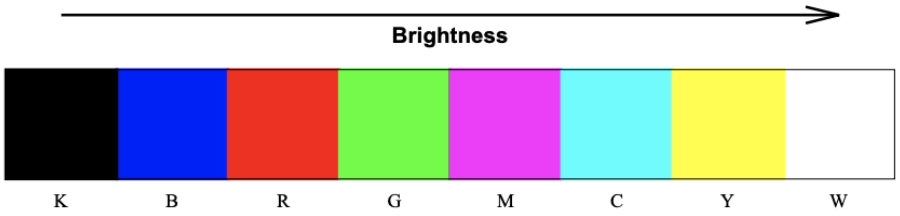
(Source: https://pdfs.semanticscholar.org/c67f/37a2ab36bab46bb632b65dc8dc3866f7c80e.pdf)

Separable Error Diffusion:

One simple idea to achieve color halftoning is to separate an image into CMY three channels and apply the Floyd-Steinberg error diffusion algorithm to quantize each channel separately. Then, you will have one of the following 8 colors, which correspond to the 8 vertices of the CMY cube at each pixel:

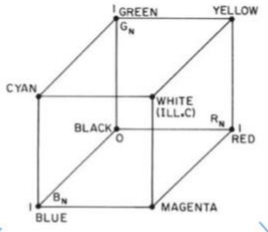
W = (0,0,0), Y = (0,0,1), C = (0,1,0), M = (1,0,0),
G = (0,1,1), R = (1,0,1), B = (1,1,0), K = (1,1,1)

Note that (W, K), (Y, B), (C, R), (M, G) are 4 complementary color pairs.



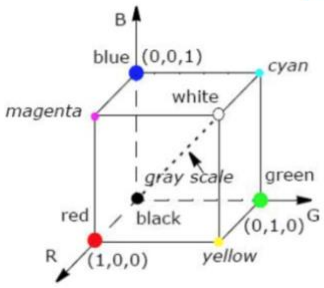
The color space is given below where the 8 colors are mapped to the vertices of the 3d cube.

- Color space



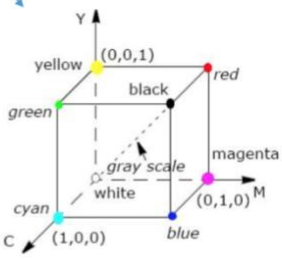
(W, K), (Y, B), (C, R), (M, G)
are complementary color pairs

RGB Cube
Representation



K = (0,0,0)
B = (0,0,1)
G = (0,1,0)
R = (1,0,0)
C = (0,1,1)
M = (1,0,1)
Y = (1,1,0)
W = (1,1,1)

CMY Cube
Representation



W = (0,0,0)
Y = (0,0,1)
C = (0,1,0)
M = (1,0,0)
G = (0,1,1)
R = (1,0,1)
B = (1,1,0)
K = (1,1,1)

MBVQ-based Error Diffusion:

The full form of MBVQ is Minimal Brightness Variation Quadruples. Shaked et al. [5] proposed a new error diffusion method, which can overcome the shortcoming of the separable error diffusion method. He claimed that the human vision is much more sensitive to brightness changes than to color changes. To reduce halftone noise, select from within all halftone sets by which the desired color may be rendered, the

one whose brightness variation is minimal. We can render any color with simply using 4 colors. Hence, we can approximate any color with 4 colors instead of 8. This will save up resources with different colors requiring different quadrants.

II. Approach and Procedure

Separable Error Diffusion:

Separate an image into CMY three different color channels. This is done because in printers, printing is done usually in the CMY space. Apply the Floyd-Steinberg error diffusion algorithm to quantize each channel separately.

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

The method is for every pixel value, find the closest of the 8 vertices in the cube above. So we first convert the RGB image into CMY image. Separate the CMY image into 3 C, M, Y channels. Perform the Floyd Steinberg's error diffusion method on these separate C, M, Y channels. After the processing is done, combine the C, M, Y channels and convert the image into RGB color image.

Algorithm implemented in C++ :

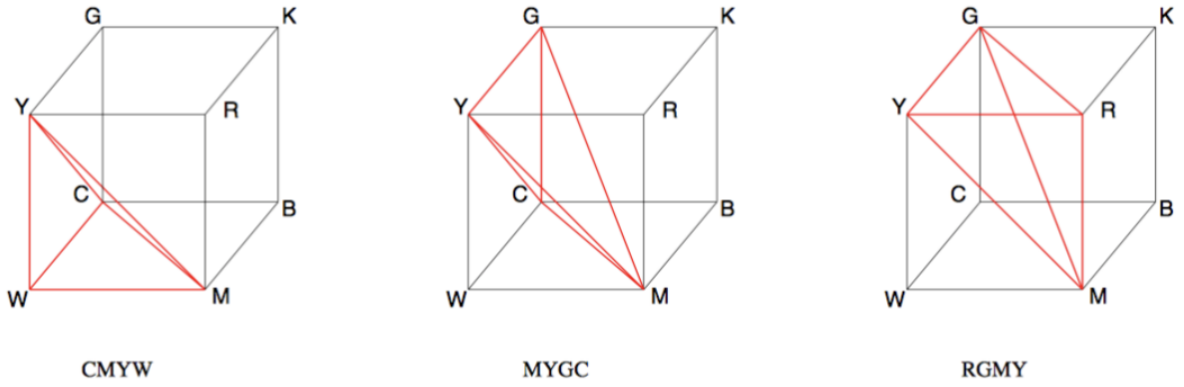
- Read the color input image "bird.raw" whose dimensions are height_size, width_size, BytesPerPixel using fread () function
- The input image is normalized for values between 0-1
- Get C, M, Y values using the R, G, B values in the color image
- Calculate the error diffusion matrix using Floyd-Steinberg's (FS) matrix with and without serpentine scanning
- For raster scanning, assign a value to the output pixel by using thresholding. If the value of input image <= 0.5 or 127 depends on normalizing, assign a value of 255 or else assign a value of 0
- For serpentine scanning, if the row is even, check by using the modular 2 operation, assign a value to the output pixel by using thresholding. If the value

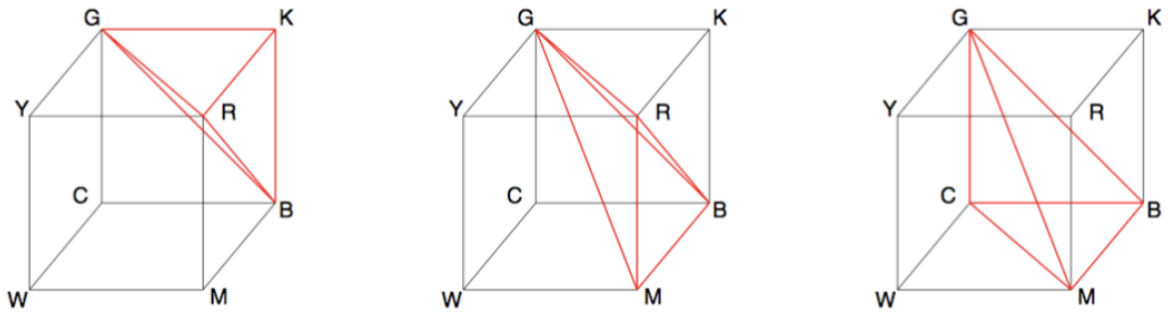
of input image <= 0.5 or 127 depends on normalizing, assign a value of 255 or else assign a value of 0

- Calculate the error by subtracting new value assigned (binarized value) to the pixel after thresholding from the old value (error = output – input)
- Diffuse the error to the future neighboring pixels as shown in the matrices above
- Repeat the procedure for scanning of odd rows when modular 2 operation fails
- Combine the C, M and Y channels into the output image
- Covert this CMY image into RGB output image
- Write the computed image data array on output.raw file using the fwrite() function

MBVQ-based Error Diffusion:

The human vision is much more sensitive to brightness changes than to color changes. So we actually need to map the points inside the cube to the points with the smallest brightness changes. The cube below is divided into 6 different regions. And inside each region, we have the minimum brightness changes. Hence the name minimum brightness variation. So we find in one of these 6 regions or tetrahedrons the point that contains the RGB pixel value. The given pixel lies in one of the 6 quadruples and hence we first need to find in which quadruple does the pixel belong. MBVQ corresponds to the 4 points of the quadruple. So each region is defined by 4 vertices which is MBVQ. Next point is to find the vertex point inside this one region which is closest to the RGB value plus error. We use the error diffusion method. The RGB value and the error will also correspond to some point in the cube and then we find one of the 4 vertices that closest to the point. Third step is to find the quantization error like in the error diffusion method and next step is to diffuse it to the future neighboring pixels.





For each pixel (i, j) in the image do:

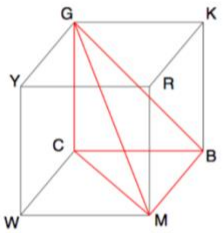
1. Determine MBVQ($RGB(i, j)$).
2. Find the vertex $v \in$ MBVQ which is closest to $RGB(i, j)+e(i, j)$.
3. Compute the quantization error $RGB(i, j) + e(i, j) - v$.
4. Distribute the error to "future" pixels.

Step 1: Determine MBVQ' pseudo code is shown below.

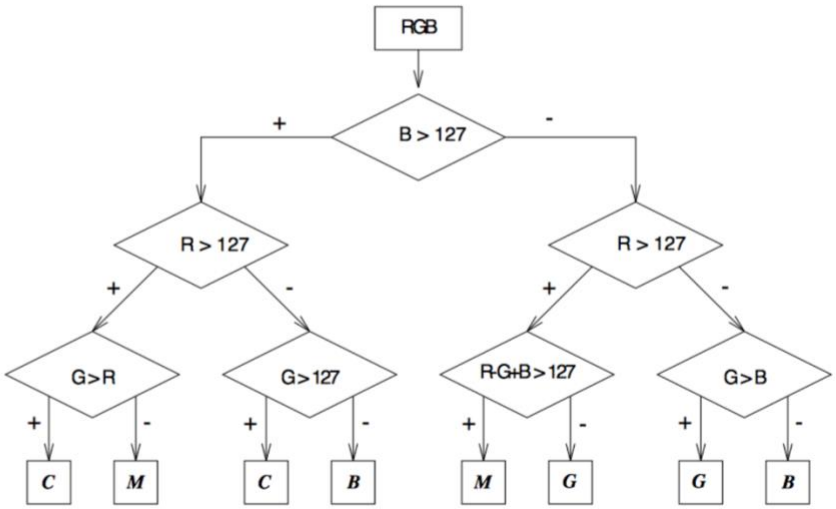
```
pyramid MBVQ(BYTE R, BYTE G, BYTE B)
{
  if((R+G) > 255)
    if((G+B) > 255)
      if((R+G+B) > 510)     return CMYW;
      else                  return MYGC;
    else                    return RGMY;
  else
    if(!((G+B) > 255))
      if(!((R+G+B) > 255)) return KRGB;
      else                  return RGBM;
    else                    return CMGB;
}
```

Step 2: To determine the closet vertex

- (e.g. for CMGB Quadruple)



CMGB



The MBVQ method makes sure that the White and Black dots are used less whereas the saturated color dots are used more. Hence, the color saturation in the MBVQ method is better.

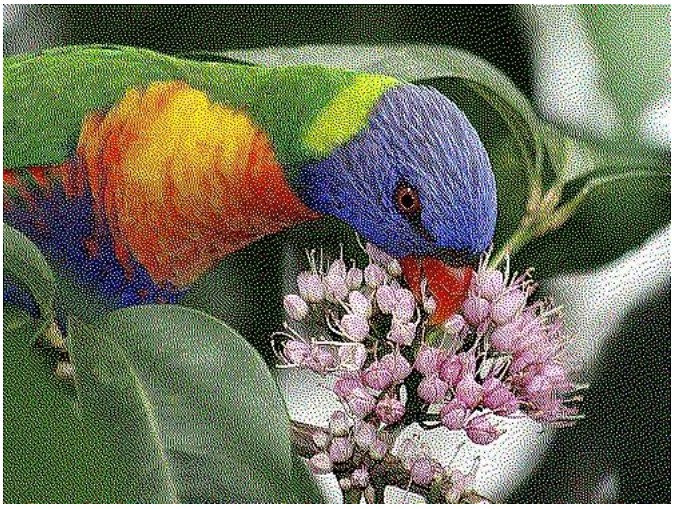
Algorithm implemented in C++ :

- Read the color input image "bird.raw" whose dimensions are height_size, width_size, BytesPerPixel using fread () function
- The input image is normalized for values between 0-1
- Get C, M, Y values using the R, G, B values in the color image
- Calculate the error diffusion matrix using Floyd-Steinberg's (FS) matrix with and without serpentine scanning
- For raster scanning, assign a value to the output pixel by using thresholding. If the value of input image <= 0.5 or 127 depends on normalizing, assign a value of 255 or else assign a value of 0
- For serpentine scanning, if the row is even, check by using the modular 2 operation, assign a value to the output pixel by using thresholding. If the value of input image <= 0.5 or 127 depends on normalizing, assign a value of 255 or else assign a value of 0
- Get the R, G, B values for every pixel location
- Using the above algorithm/code, find the tetrahedron/quadruple in which the pixel lies
- Find the Euclidian distance of the pixel from the 4 vertices of the quadruple

- Select the vertex which corresponds to the minimum distance compared to others using the C, M, Y values
- Calculate the error by subtracting new value assigned (binarized value) to the pixel after thresholding from the old value (error = output – input)
- Diffuse the error to the future neighboring pixels as shown in the matrices above
- Repeat the procedure for scanning of odd rows when modular 2 operation fails
- Combine the C, M and Y channels into the output image
- Covert this CMY image into RGB output image
- Write the computed image data array on output.raw file using the fwrite() function

III. Experimental Results

Input Image bird.raw



Output Image using Separable error diffusion with serpentine scanning

Output Image using MBVQ-based error diffusion with serpentine scanning

IV. Discussion

The main difference in the two methods mentioned above is: Separable error diffusion finds the closest vertex of all the 8 vertices in the cube whereas, MBVQ based error diffusion finds the closest vertex in the quadruple out of 4 vertices as the output pixel. While calculating the closest vertex, norm is not specified hence might create an ambiguity there in MBVQ based on which norm we prefer while calculating closest vertex.

The error diffusion method used in the two techniques is the same that is Floyd Steinberg's serpentine error diffusion method.

The main shortcoming of the Separable error diffusion method is that it does not exploit the inter color correlation. This leads to color artifacts and poor color rendering. This method does not converge and for high sharpness and low artifacts, it shows patchy or blurry output. When the error is distributed to the three channels separately, the compensation is done in error distribution, but when the channels are combined the error increases and causes artifacts in the output image. The output from the Separable method is noisy but sharper because of the error diffusion in the future pixels. To overcome these shortcomings, we use the MBVQ based error diffusion method.

Comparing the results above, we can see that the images look similar, however, MBVQ based method gives the output image which is better than the Separable method in terms of preserving the color and the brightness.

Looking at the outputs, we observe that on comparing the intensity and brightness, the MBVQ method gives better results than Separable method. One of the reasons is there are more colors used. All channels are considered simultaneously when applying the error diffusion method and hence the output image is closer to the input image.

Also, MBVQ method reduces the halftone noise. It also selects the output vertex in a manner that there is minimal brightness variation.

The high frequency pattern on the neck of the bird, especially the red in yellow color is very well preserved in the MBVQ based error diffusion as compared to the Separable error diffusion. The texture and different patterns come out much better in MBVQ.

Since in MBVQ based method, we use 8 colors to output the image, it creates higher density in the regions which are dark, and the density is lower in lighter color regions.

MBVQ code takes a little more time as compared to Separable one in terms of the run time as MBVQ has high pixel density, retains local averaging and reduces noise which results in more time.