Subject: EE569

Homework 2

Name: Shreyansh Dnyanesh Daga

USC ID: 6375-3348-33

Email: sdaga@usc.edu

Date: 10/13/2013

Problem: 1 Special Effects Filter

Motivation:

Many applications which offer services in imaging processing, have various new, attractive features by use of which we can manipulate and modify our basic image to the desired image with use of various special effects present in the applications. The underlying implementation of these special effects filters forms the motivation of this problem. We discuss how certain basic special effect filters work and we will see their implementation as well. Understanding of working of these filters puts forward many new and different ways to implement them to get more new special effects.

Approaches and Procedure and results:

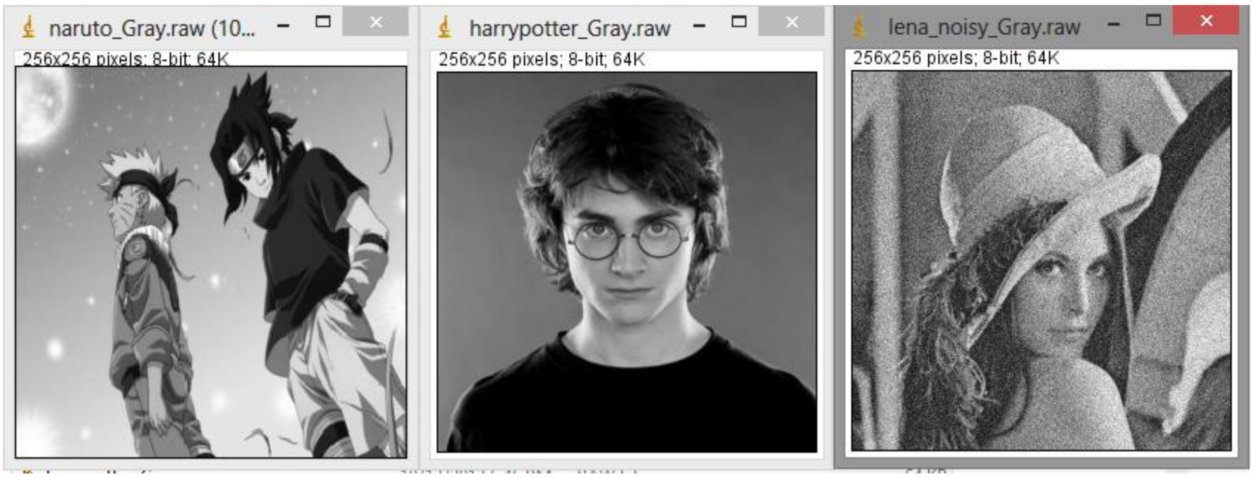
P1.A Color to Gray scale conversion

For this part of the problem, we are given 3 different RGB images which we have to convert to gray scale images. Gray scale images represent the luminance information of the color in the image. The luminance information or the luminance channel is given by the following formula,

$$Y = 0.299 \times R + 0.587 \times G + 0.114 \times B$$

Here the R, G and B are the respective components of the Red, Green and Blue channel in the color image. Thus after implementing this, we get a gray scaled image of the color image as shown in the results.

Results:


naruto_Gray.raw (10...
256x256 pixels; 8-bit; 64K
harrypotter_Gray.raw
256x256 pixels; 8-bit; 64K
lena_noisy_Gray.raw
256x256 pixels; 8-bit; 64K

Grayscale conversion results

P1.B Pencil Sketch using edge detection

For this part of the problem, we have to generate a pencil sketch of the given color image. Pencil sketch filters represent the original image as a pencil drawn image.

Pencil sketch is mostly done by artists, where they highlight the edges of the object in the image, hence to get a pencil image, we must detect edges in the image. Also pencil sketches are mostly done using a gray scale color, so the image looks a gray scaled pencil sketch of the original image.

To detect edges, we must perform edge detection on the luminance information of the image as the luminance of any image is more sensitive to the eye, and the sketch “feels” real.

For this we convert the RGB color image into a gray scale image as seen before, and implement the following methods to detect edges.

1) Edge detection using 1st order derivative.

In this method, we are performing the edge detection using the 1st order differential of the image to get the edge points.

The color image is converted into gray scale image.

The derivative is taken across rows and columns. To get the gradient map

Grows(j,k) = I(j,k) – I(j-1,k)

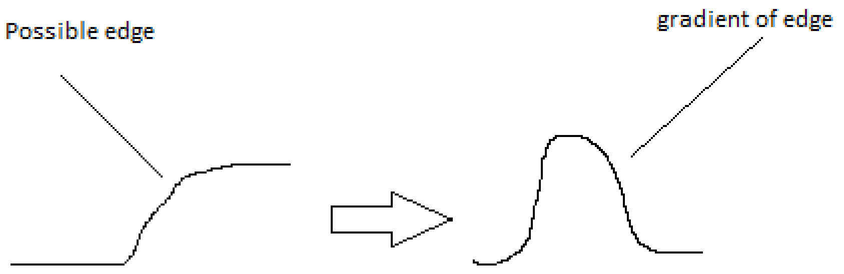
Gcols(j,k) = I(j,k) – I(j,k-1)

This is the backward difference method to obtain the derivative in discrete form.

Gradient(j,k) = Grows(j,k)^2 + Gcols(j,k)^2

This will give us the gradient map.

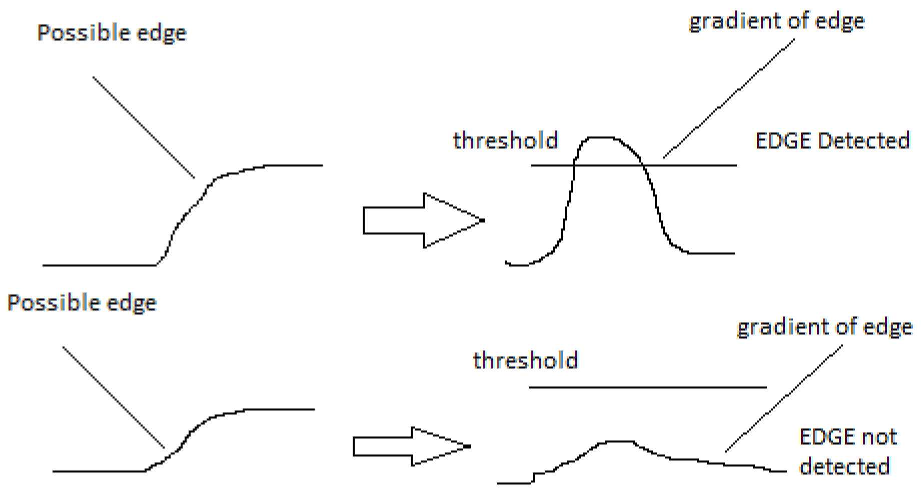
Now an edge will have a following gradient for change in pixel intensities;



Thus we need to decide a threshold for this inorder to get the correct edge values.

For example more prominent the edge, more will be its gradient and less prominent the edge, less will be its gradient. Hence deciding this threshold is of most important.

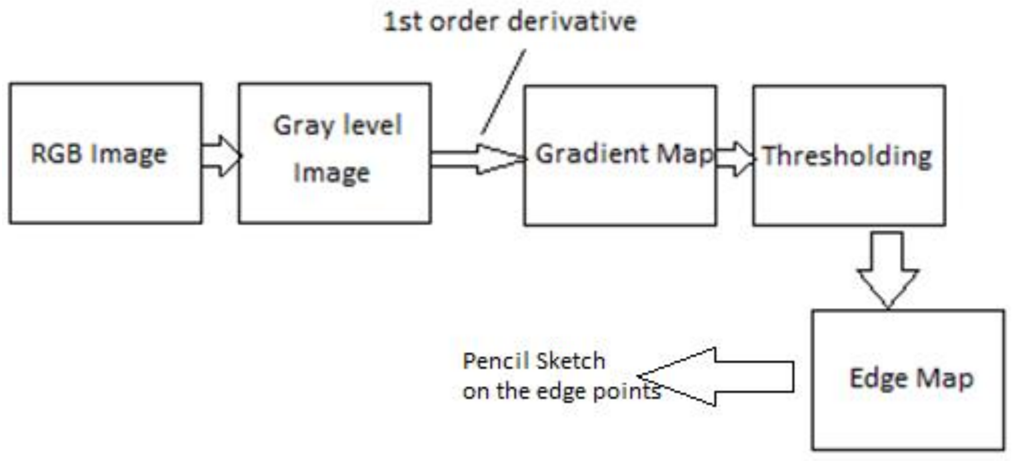
This can be explained as follows



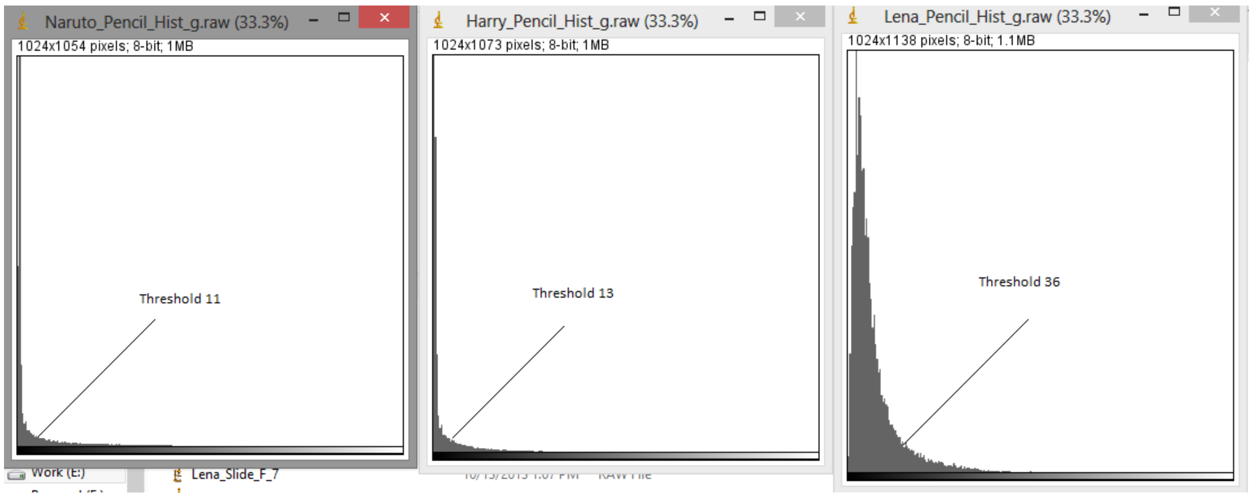
Thus using a threshold, we decide if it's an edge point or not.

Thus we get an edge map after this step where we can replace these edge points with the original gray scale value to get the pencil sketch

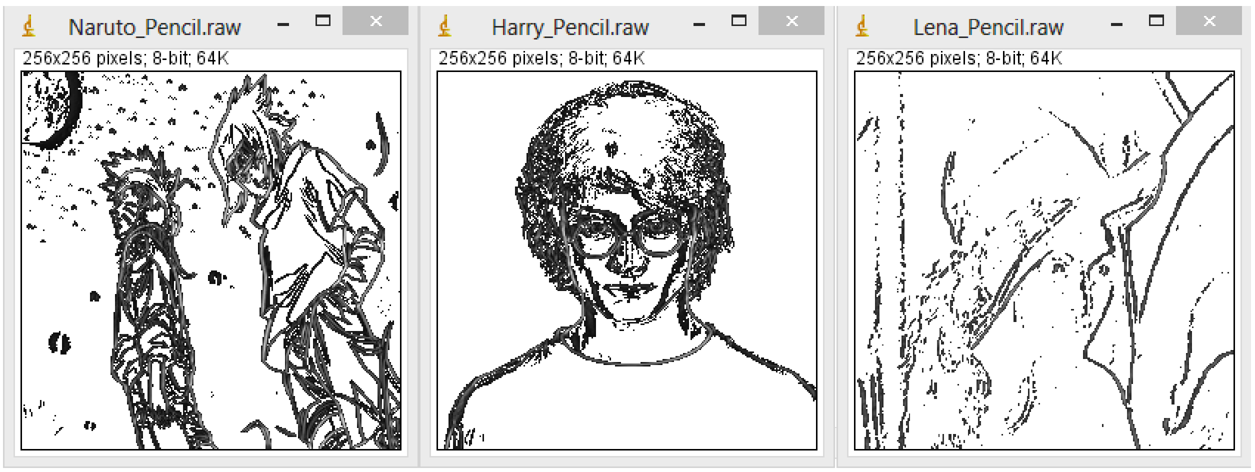
The entire method is described as follows



Result:



Determining threshold from histogram of gradient of 1st order


Naruto_Pencil.raw
256x256 pixels; 8-bit; 64K
Harry_Pencil.raw
256x256 pixels; 8-bit; 64K
Lena_Pencil.raw
256x256 pixels; 8-bit; 64K

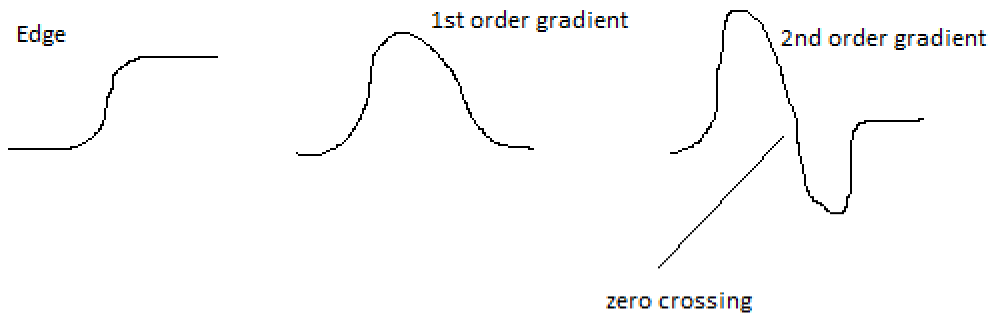
Pencil Sketch from Edge map using 1st order

2) Edge detection using 2nd order derivative

This method involes taking the 2nd derivative which will give us the zero crossing if the edge is present or not. This method is more effective than the first order, however results show that the earlier methid gives better performance than this method.

This is also know as Laplacian operation and this is performed along with gaussian filtering know as laplacian of gaussian(LoG)

In this method, the second order derivative will give us a much easy to analyze gradient map where threshlding is easy to be done as, the thresholding is done on the knee points of the histogram of the gradient map. Once thresholding is done, we need to confirm for the zero crossings as the 2nd order derivative will have zero crossings on the gradient map.



The method for second order derivative is summarized as follows

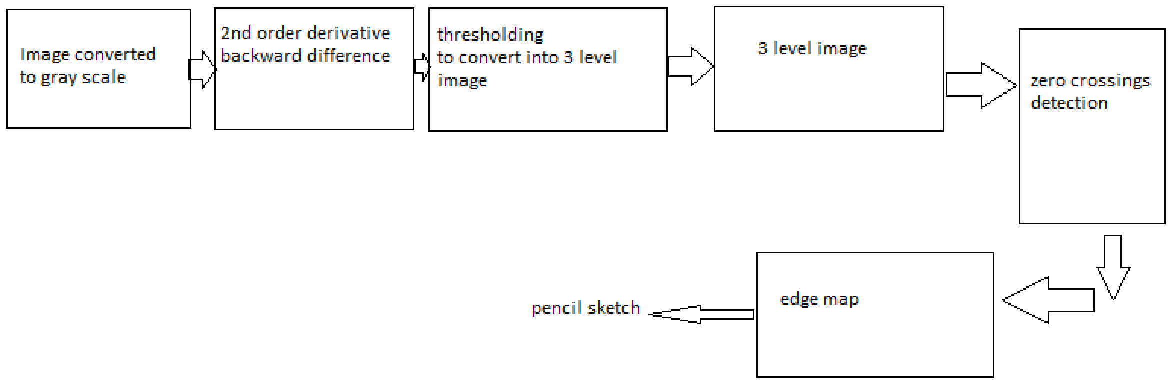
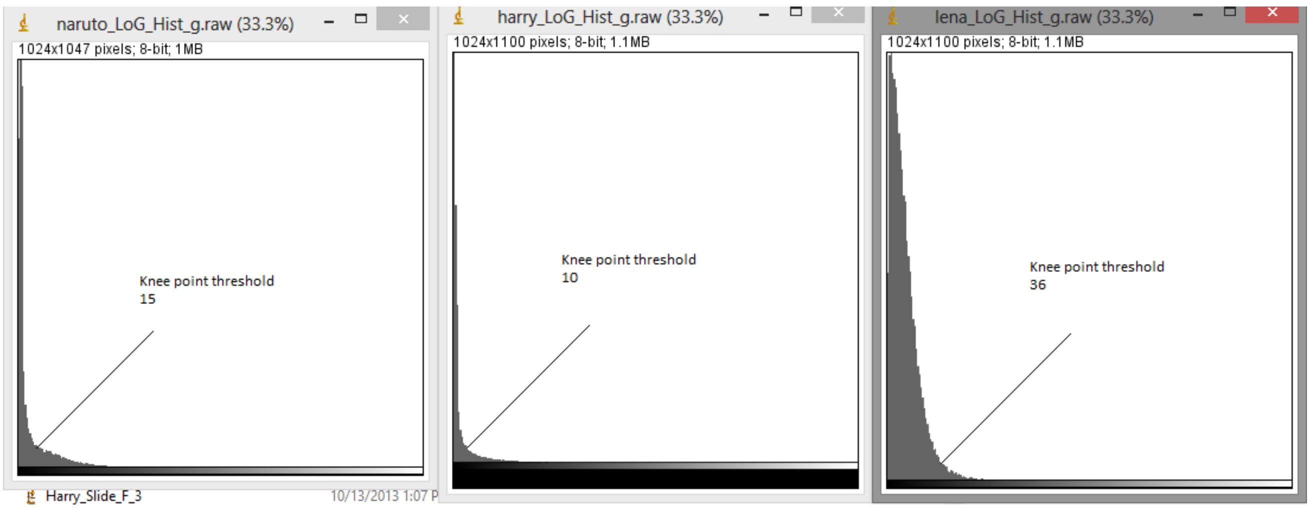


Image converted to gray scale
2nd order derivative backward difference
thresholding to convert into 3 level image
3 level image
zero crossings detection
edge map
pencil sketch

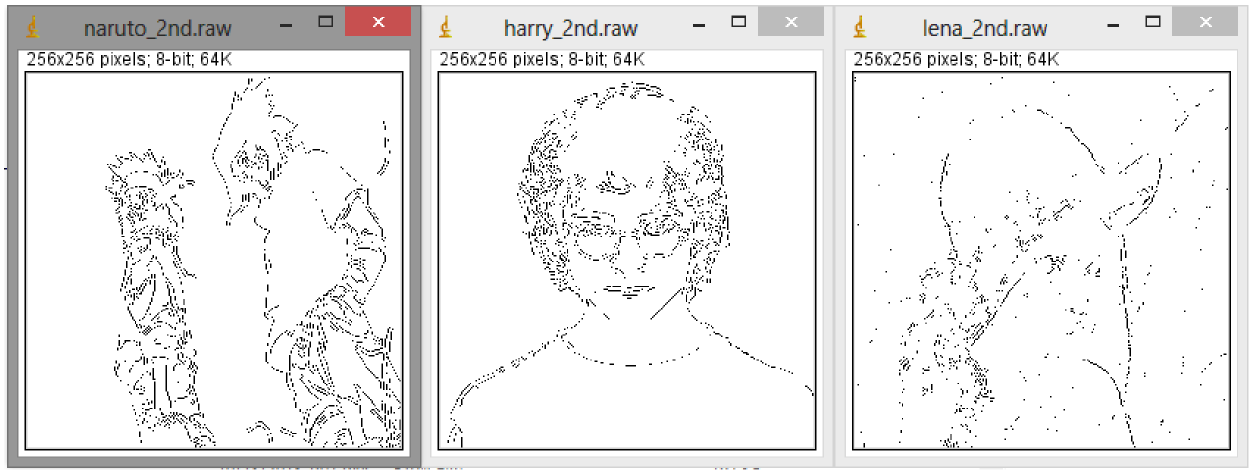
The gradient map is then threshold to reduce it into a 3 level image (-1,0,1) since 2nd order derivative will have negative terms as vales and these will be symmetric, hence I have flipped these negative values to positive side, added their count with positive side and then plotted the histogram to determine the knee point threshold.

Next this 3 level image is compared with 8 types of possible zero crossing matrices in order to determine that a zero crossing has taken place. This will then give the edge map, which can be used to generate the pencil sketch

Results:



These represent the histograms of LoG maps of three images on which the thresholdoing is decided


naruto_2nd.raw
256x256 pixels; 8-bit; 64K
harry_2nd.raw
256x256 pixels; 8-bit; 64K
lena_2nd.raw
256x256 pixels; 8-bit; 64K

Pencil Sketch of three images after LoG edge detection

P1.C Background special effect

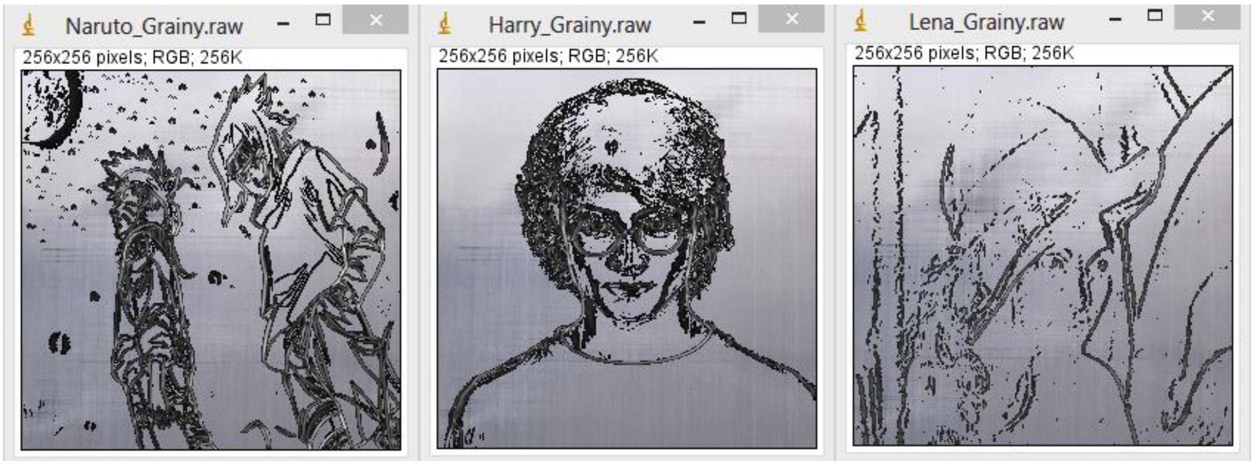
In this part we combined the pencil edge image with a grainy background image to get a background feel of the drawing canvas which is used by artists to do pencil sketch.

The image combination is implemented as follows

$$G(j,k) = I(j,k) + APLHA*Grainy(j,k) + BETA$$

Where APLHA and BETA control how the grainy background looks and I is the pencil sketch, and Grainy is the grainy background image and G is the output image. The following is obtained with APLHA = -0.80 and BETA = 20

Although we can vary these two values and see their effect as well.


Naruto_Grainy.raw
256x256 pixels; RGB; 256K
Harry_Grainy.raw
256x256 pixels; RGB; 256K
Lena_Grainy.raw
256x256 pixels; RGB; 256K

Grainy background

<u>P1.D Transition Effect</u>

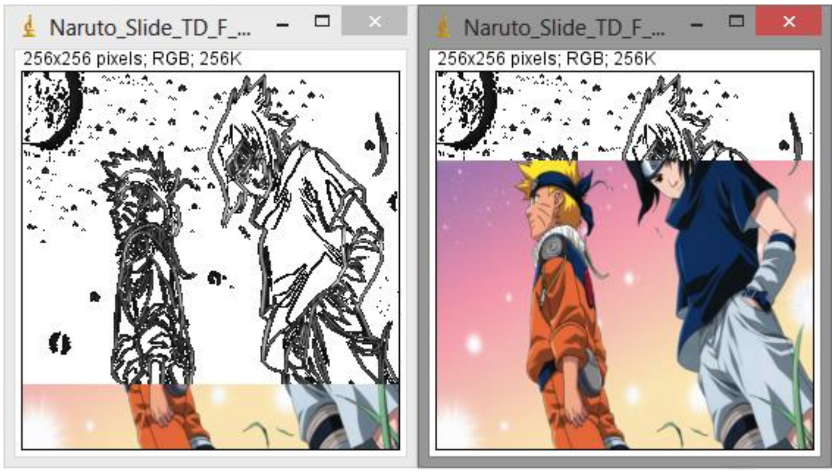
In this part we create transition effects using the original RGB image and the Pencil Sketch generated image. We have following types of transition effects.

1) Sliding Transition
2) Fade In Transition

The sliding transition implemented as follows, we slide through the RGB image into pencil sketch image in top to bottom direction as shown and also we slide through left to right

The following is done using combination of images and several such images (frames are generated) and stitched together to form a video sequence.

The numbers of frames for Sliding sequence are 64 frames for each of the given images, which are then processed in matlab to make a video of the effect.



Example screenshot of sliding effect for naruto.raw

In case of Fade in transition effect, there is a gradual transition from the color image into the pencil sketch image. This is implemented using the following formula

$$\text{Frame}(j,k) = (0.5 + \text{APLHA}) \times \text{ImageRGB}(j,k) + (0.5 - \text{APLHA}) \times \text{Pencil}(j,k)$$

This is repeated by varing ALPHA from -0.5 to + 0.5 in steps to give a total of 20 frames, these frames are then stiched in matlab to give the transition effect fade in.

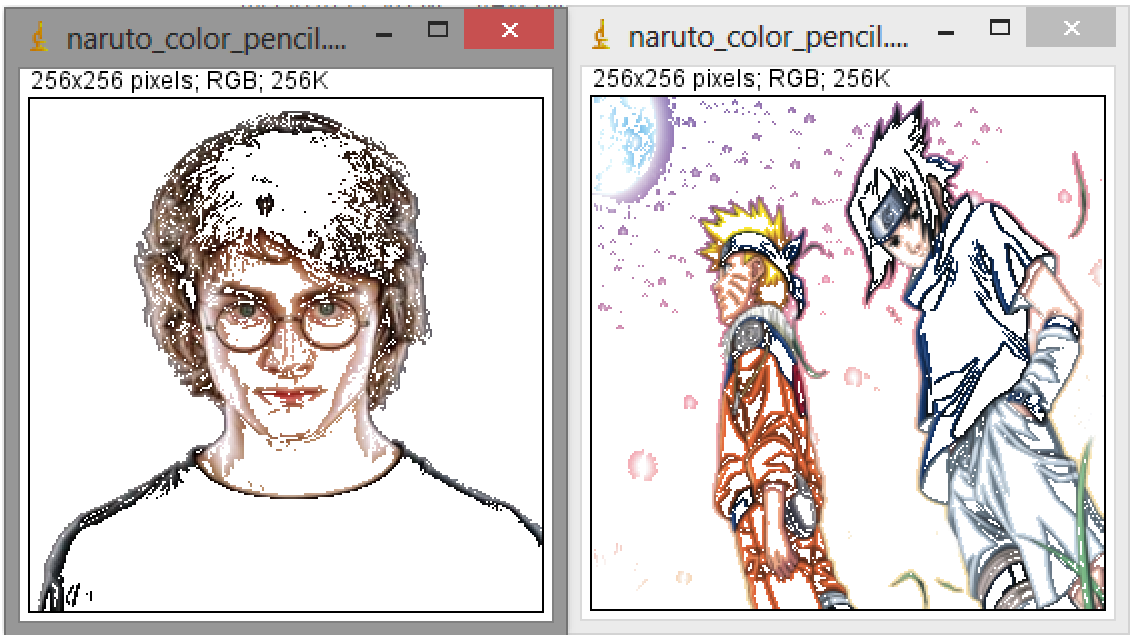
P1.E Color Sketch

The implementation of color pencil sketch is done as follows,

1) Obtaine luminance channel information.
2) Perform the edge detection on luminance channel
3) Generate edge map based on this information
4) Replace RGB pixels from original image into the places where edge is detected

This is the algorithm I designed, as performing edge detection on RGB channels separately will give color imbalance on the edge detected. This will result in the loss of perceived color difference as in case of original image.

Hence we know that eyes are most sensitive to luminance channel and hence determining the edge on this information of luma channel will preserve the color balance later when the RGB pixels are placed on thr edge detected. The following are the outputs I got,



Conclusion and Discussion:

As seen from the implementation of the edge detection, we see that deciding the threshold for the 1st order method is not easy since, edges which have a gradient value below threshold are missed. Hence this is taken care in the 2nd order method, where the edge locations will pinpoint to a zero crossing and deciding a threshold is easier since the 2nr order gradient gives a pattern where threshold is decided at the knee point. However the results show that in most case 1st order method gives a better edge map than the 2nd order method.

Many effects can be given to the image as seen above like the sliding and fade in effects which are aesthetically pleasing.

Color pencil sketch is obtained in a similar way, only the important thing to be taken care of is the color balance, and hence in case of color pencil sketch, the edge detection is done on the luma channel as luma channel is most sensitive to eye.

<u>Problem: 1 Morphological Operations</u>

<u>Motivation:</u>

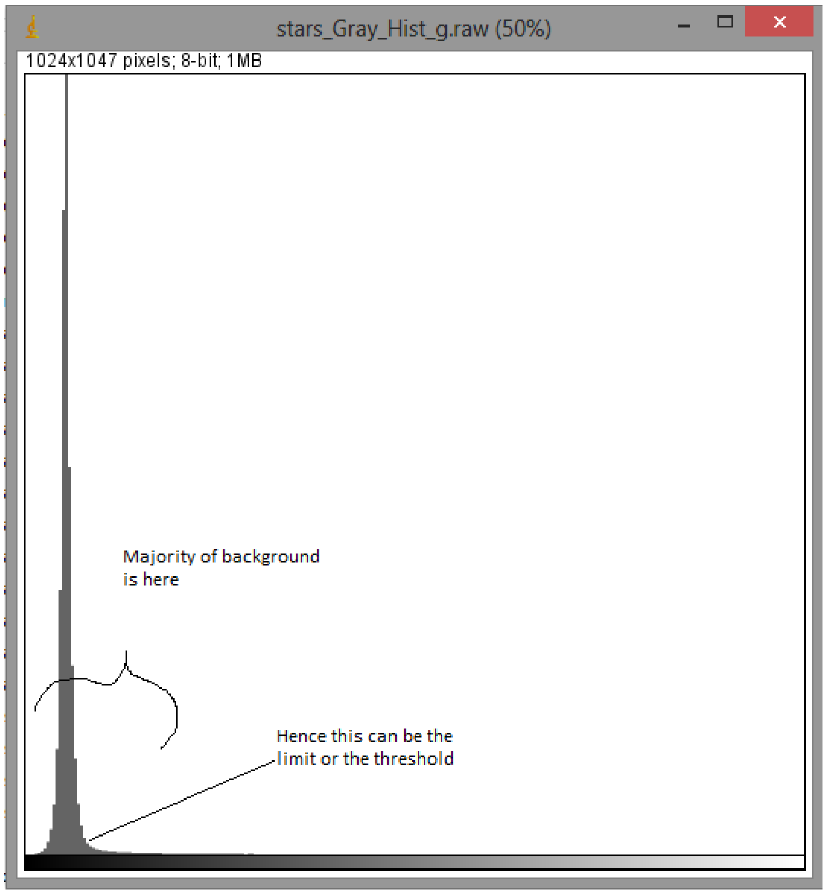
Morphology deals with the operations on the images by which we can study and analyze the structure of the image and the objects in the image. Different types of morphological operations include shrinking, thinning and skeletonization of images which have many applications in the real world. We look at these operations from an application point of view.

<u>Approaches and Procedure and results:</u>

<u>P2.A Shrinking</u>

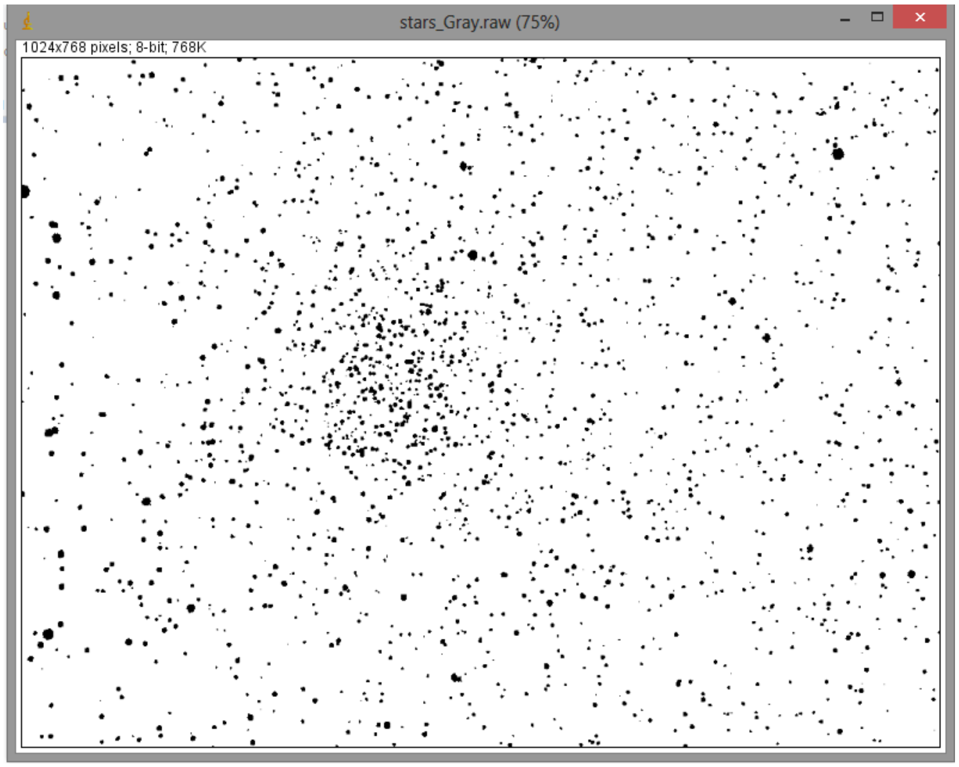
In this part of problem we implement shrinking operation to count the number of stars in the given image. For any morphological operation, we need to convert the original image into a binary image, where there are two levels(BLACK and WHITE) with BLACK mostly representing the foreground or the object and white representing the background.

In this case after plotting the histogram of stars.raw image, we decide the threshold to separate the stars from the background. One common way to separate the black background is to make all the pixels with intensity more than 0 to foreground. However as the background is not perfect black, we will get many pixels which are part of background as foreground.



Histogram of stars.raw in grayscale

Thus from the histogram we decide the threshold for separating background as 27, this will give us the following binary image

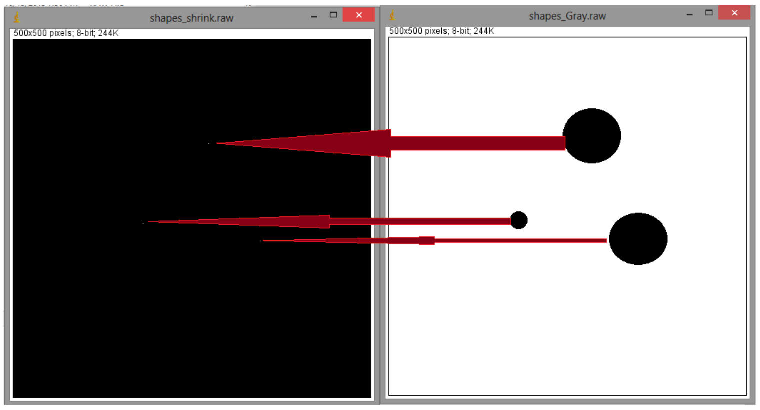


Binary image after thresholding by 27

Thus we see that a reasonable amount of stars are obtained in the foreground

Next we apply the shrinking masks on each pixel of the image, the shrinking masks given in the pattern table will give us the positions where we can eliminate the pixel in order to shrink the image.

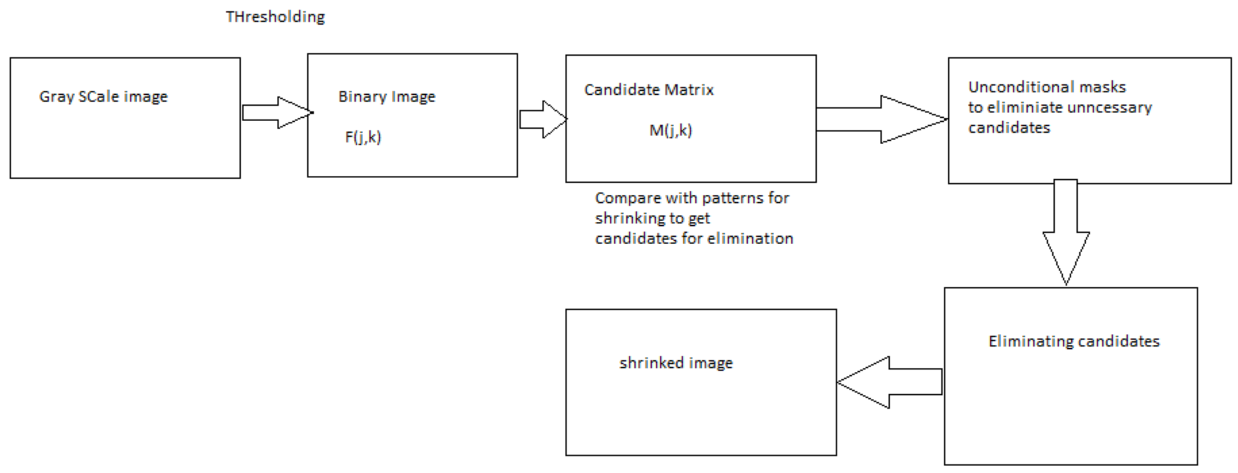
A full object will shrink to a dot , that is if the object has a continuous foreground distribution will reduce to a dot, and the object which has some background within will reduce to a ring like structure as seen from the below example.



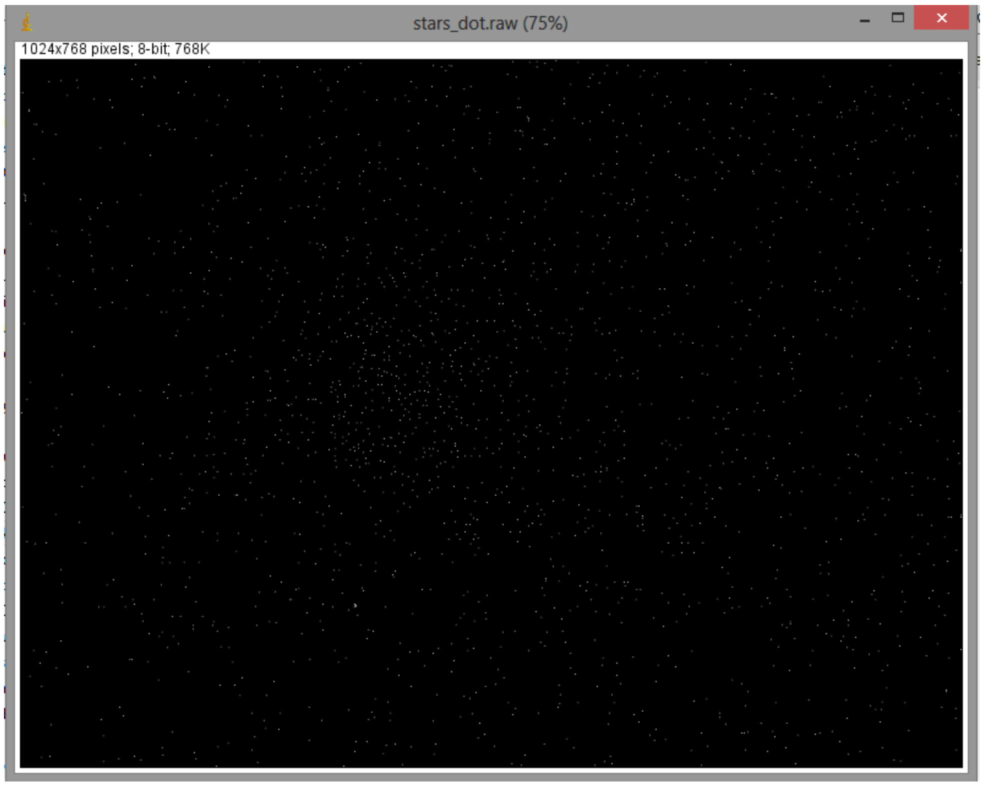
Shrinking on example of test image showing reduction into 3 dots.

Hence after the first stage of applyin filters, we check for the unconditional masks as if there are even number of pixels in the object, all of them might get eliminated and hence to avoid this, we use the unconditional masks which will eliminate the candidate to be removed and preserve the candidate which is to be removed without eliminating the object completely.

This can be sumurized as follows

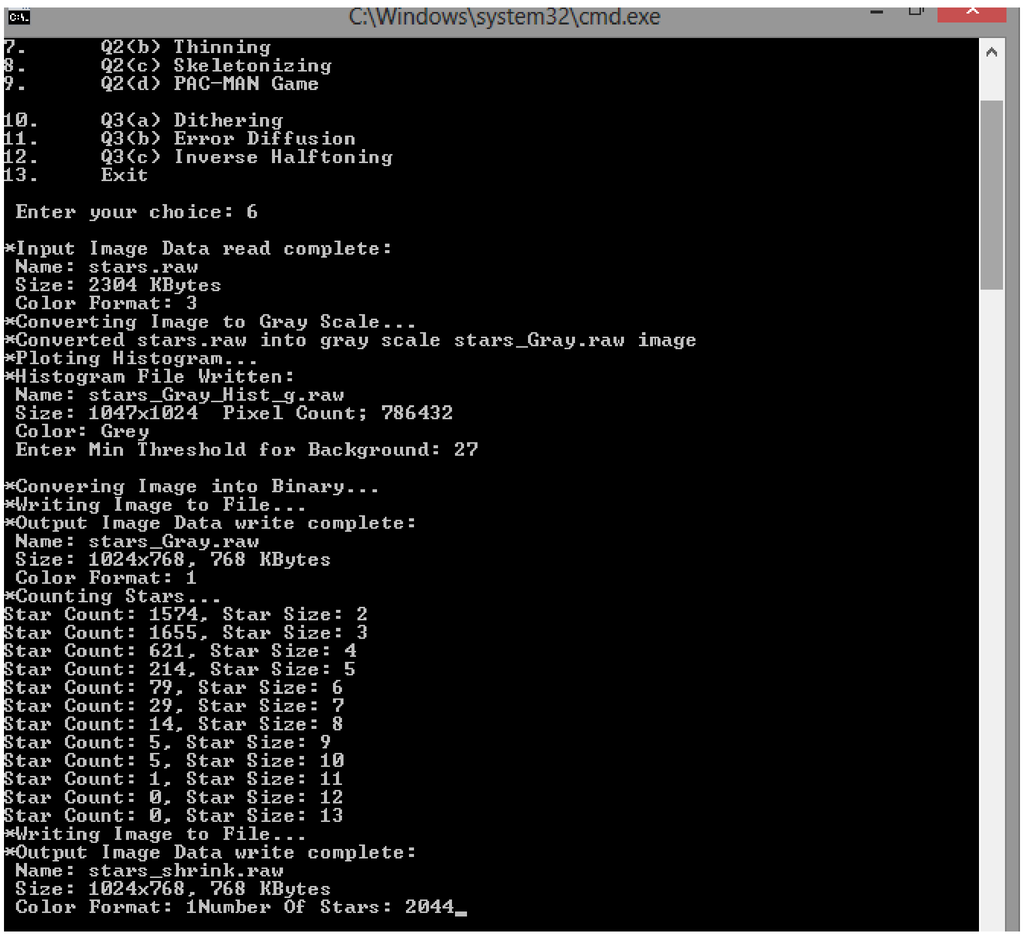

THresholding
Gray SCale image
Binary Image
F(j,k)
Candidate Matrix
M(j,k)
Compare with patterns for shrinking to get candidates for elimination
Unconditional masks to eliminiate unncessary candidates
Eliminating candidates
shrinked image

Thus for a thresholding of 27, we get the following shrinked image of stars.raw



Shrinked image of stars.raw

Thus the total number of stars can obtained by counting the number of stars and in this case it is 2044



```
C:\Windows\system32\cmd.exe
7.      Q2(b) Thinning
8.      Q2(c) Skeletonizing
9.      Q2(d) PAC-MAN Game

10.     Q3(a) Dithering
11.     Q3(b) Error Diffusion
12.     Q3(c) Inverse Halftoning
13.     Exit

 Enter your choice: 6

*Input Image Data read complete:
 Name: stars.raw
 Size: 2304 KBytes
 Color Format: 3
*Converting Image to Gray Scale...
*Converted stars.raw into gray scale stars_Gray.raw image
*Ploting Histogram...
*Histogram File Written:
 Name: stars_Gray_Hist_g.raw
 Size: 1047x1024  Pixel Count; 786432
 Color: Grey
 Enter Min Threshold for Background: 27

*Convering Image into Binary...
*Writing Image to File...
*Output Image Data write complete:
 Name: stars_Gray.raw
 Size: 1024x768, 768 KBytes
 Color Format: 1
*Counting Stars...
Star Count: 1574, Star Size: 2
Star Count: 1655, Star Size: 3
Star Count: 621, Star Size: 4
Star Count: 214, Star Size: 5
Star Count: 79, Star Size: 6
Star Count: 29, Star Size: 7
Star Count: 14, Star Size: 8
Star Count: 5, Star Size: 9
Star Count: 5, Star Size: 10
Star Count: 1, Star Size: 11
Star Count: 0, Star Size: 12
Star Count: 0, Star Size: 13
*Writing Image to File...
*Output Image Data write complete:
 Name: stars_shrink.raw
 Size: 1024x768, 768 KBytes
 Color Format: 1Number Of Stars: 2044_
```

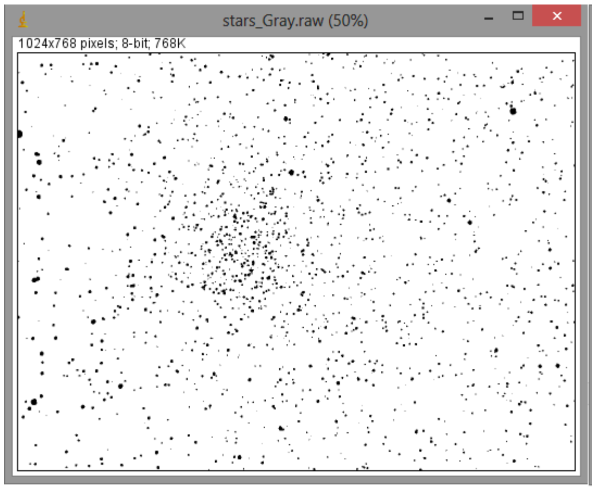
The star size can be obtained by the following method.

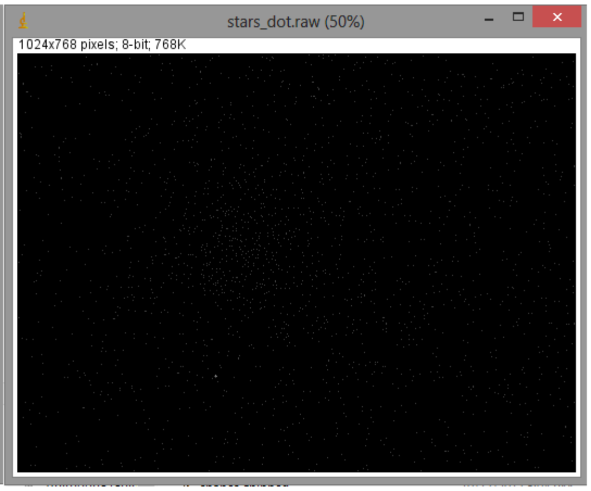
After the first stage of comparing the masks, we find out the number of candidates which are to be eliminated. Then we find the number again on the next iteration.

The difference between this will give the total number of stars at the sizes from 2 onwards. This can be seen in the result above.

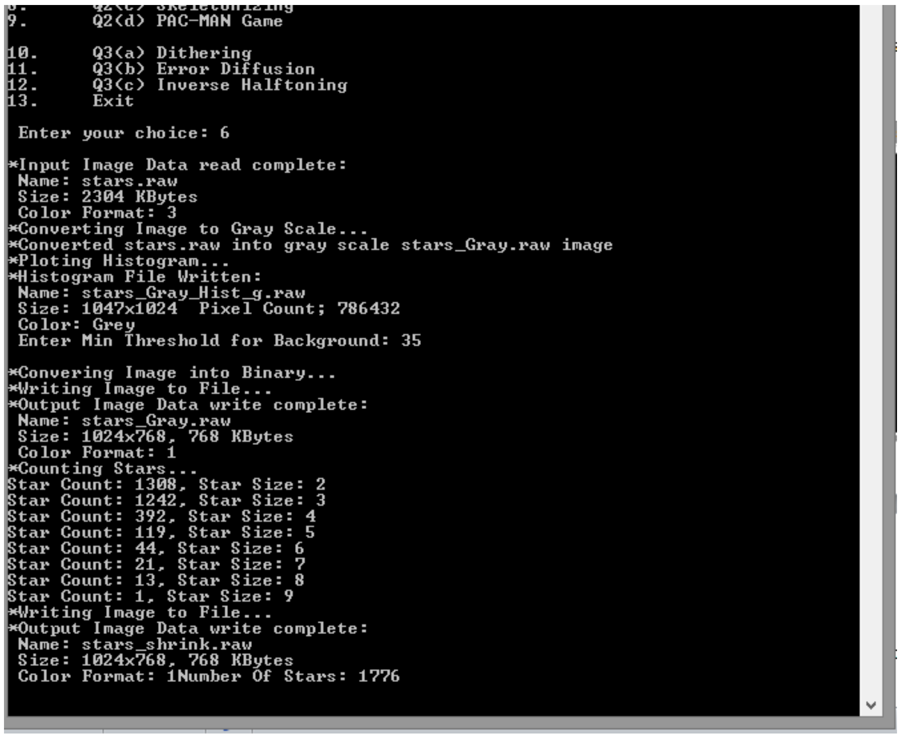
If we threshold at another value, the result changes as follows

Thresholding at 35 we get





The binary image of stars and the shrinked image of stars is as shown above and the total count is 1776



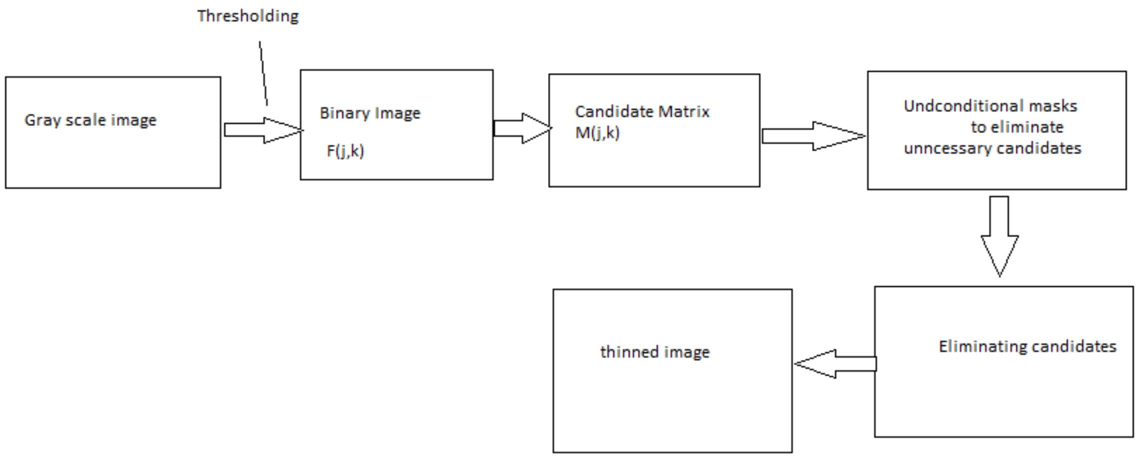
```
9.      Q2(d) PAC-MAN Game

10.     Q3(a) Dithering
11.     Q3(b) Error Diffusion
12.     Q3(c) Inverse Halftoning
13.     Exit

 Enter your choice: 6

*Input Image Data read complete:
 Name: stars.raw
 Size: 2304 KBytes
 Color Format: 3
*Converting Image to Gray Scale...
*Converted stars.raw into gray scale stars_Gray.raw image
*Ploting Histogram...
*Histogram File Written:
 Name: stars_Gray_Hist_g.raw
 Size: 1047x1024  Pixel Count; 786432
 Color: Grey
 Enter Min Threshold for Background: 35

*Convering Image into Binary...
*Writing Image to File...
*Output Image Data write complete:
 Name: stars_Gray.raw
 Size: 1024x768, 768 KBytes
 Color Format: 1
*Counting Stars...
Star Count: 1308, Star Size: 2
Star Count: 1242, Star Size: 3
Star Count: 392, Star Size: 4
Star Count: 119, Star Size: 5
Star Count: 44, Star Size: 6
Star Count: 21, Star Size: 7
Star Count: 13, Star Size: 8
Star Count: 1, Star Size: 9
*Writing Image to File...
*Output Image Data write complete:
 Name: stars_shrink.raw
 Size: 1024x768, 768 KBytes
 Color Format: 1Number Of Stars: 1776
```

Result of thresholding at 35

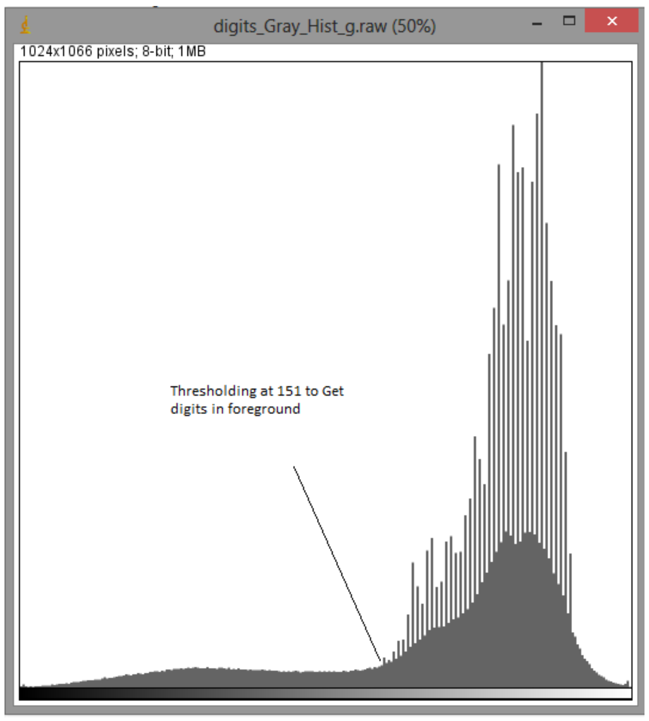
P2.B Thinning

This part of the problem we implement the thinning operation to get the total number of digits in the given image digits.raw

Similar to shrinking, thinning is implemented using thinning filters, and again a check is performed by unconditional masks to avoid elimination of even sized objects. Thus we can summarize thinning as follows

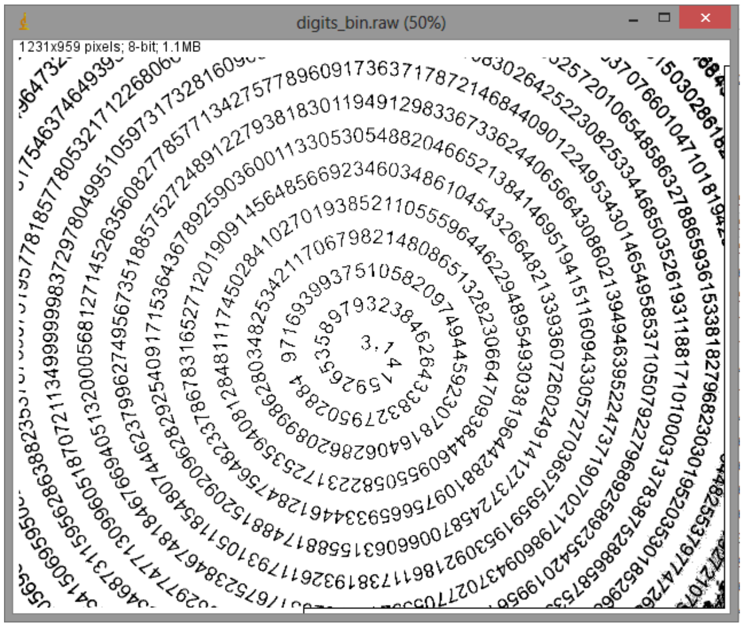


Thus we have digits.raw which gives us the following histogram



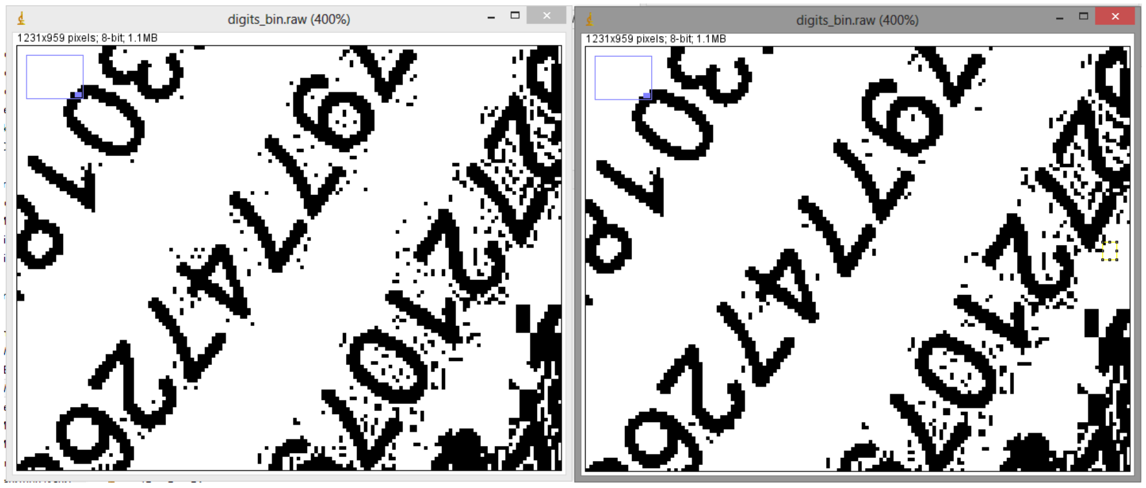
Histogram of digits.raw

On thresholding we get



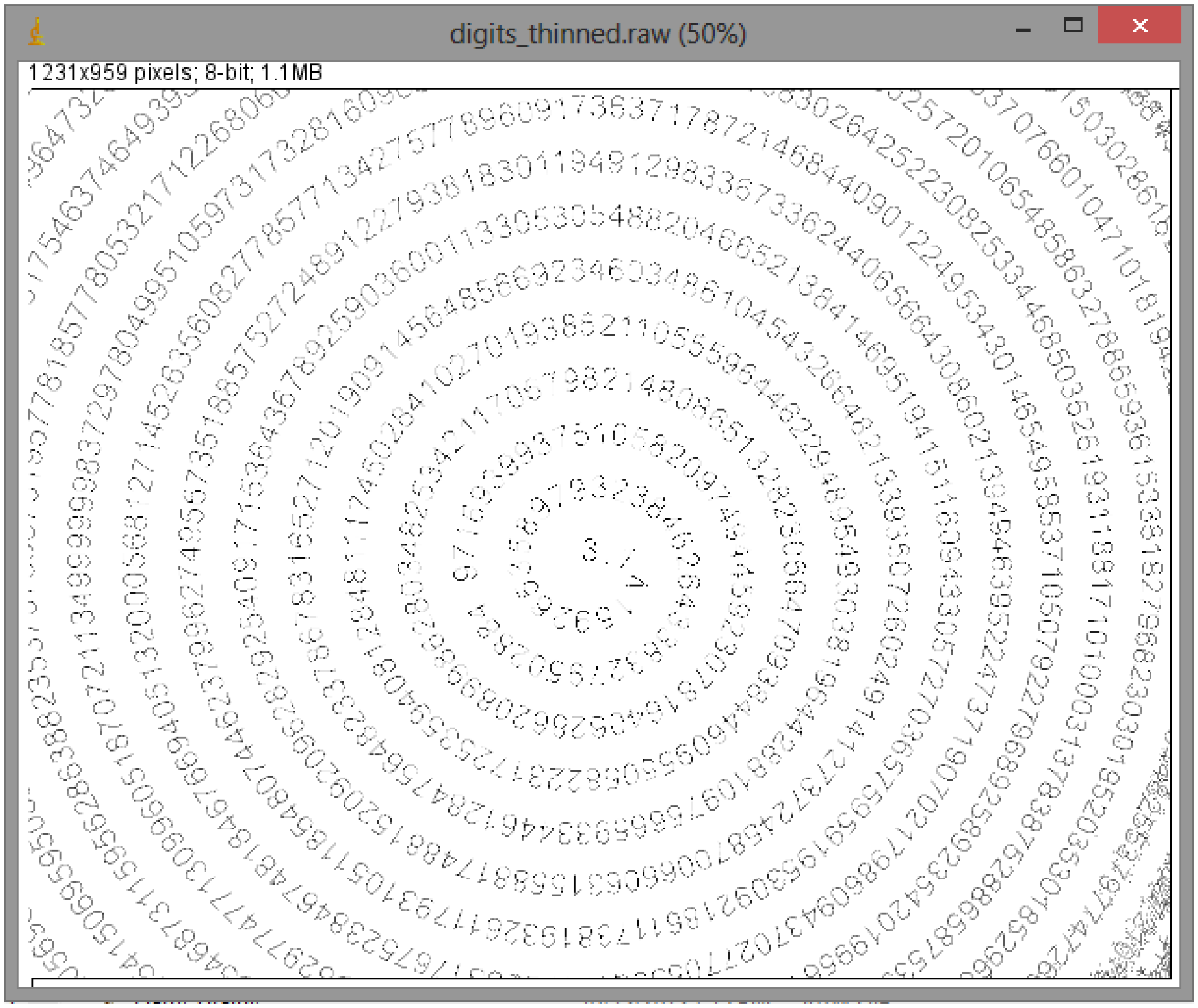
Binary after thresholding

However we see that there is an impulse noise at the bottom right corner, this noise can be reduced by removing the pixels which are surrounded by 0 on all sides.



Removing outlier pixels. Image with reduced outlier pixels is at right

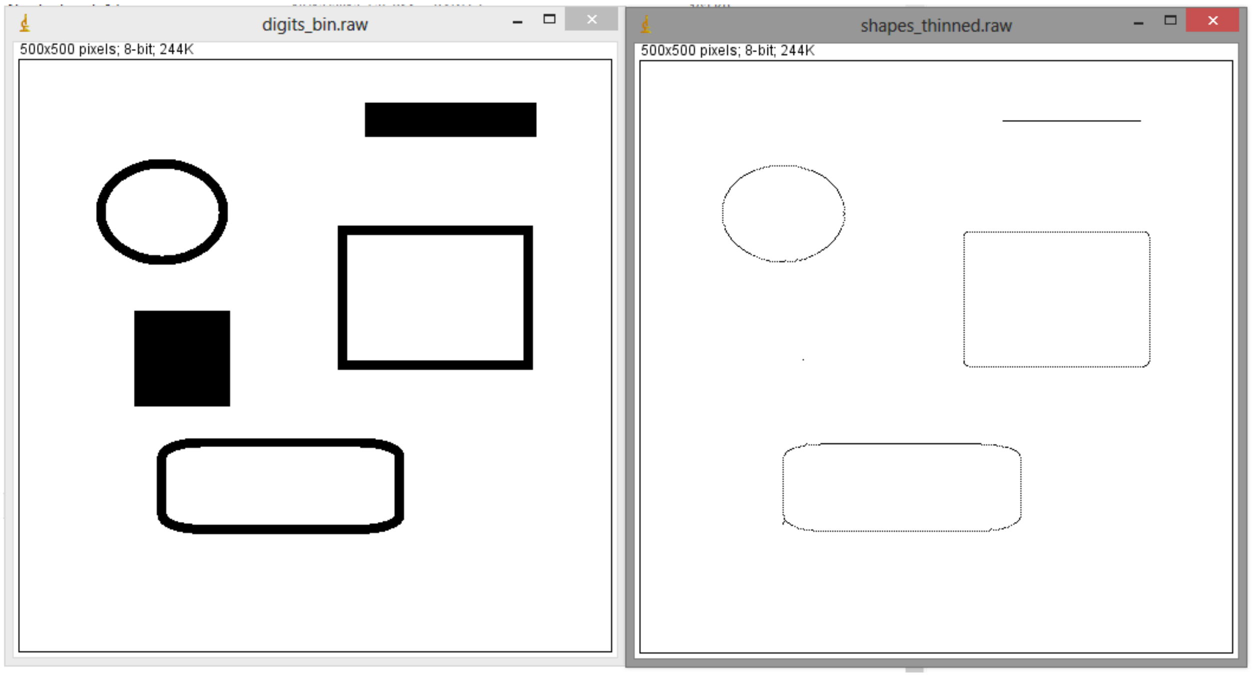
Next we apply the thinning filters one by one in each pixels to get a thinned image as follows



Thinned image of digits.raw

However I could not formulate the algorithm for counting the total number of digits in the image.

But the thinning filter work fine with the following test image as well

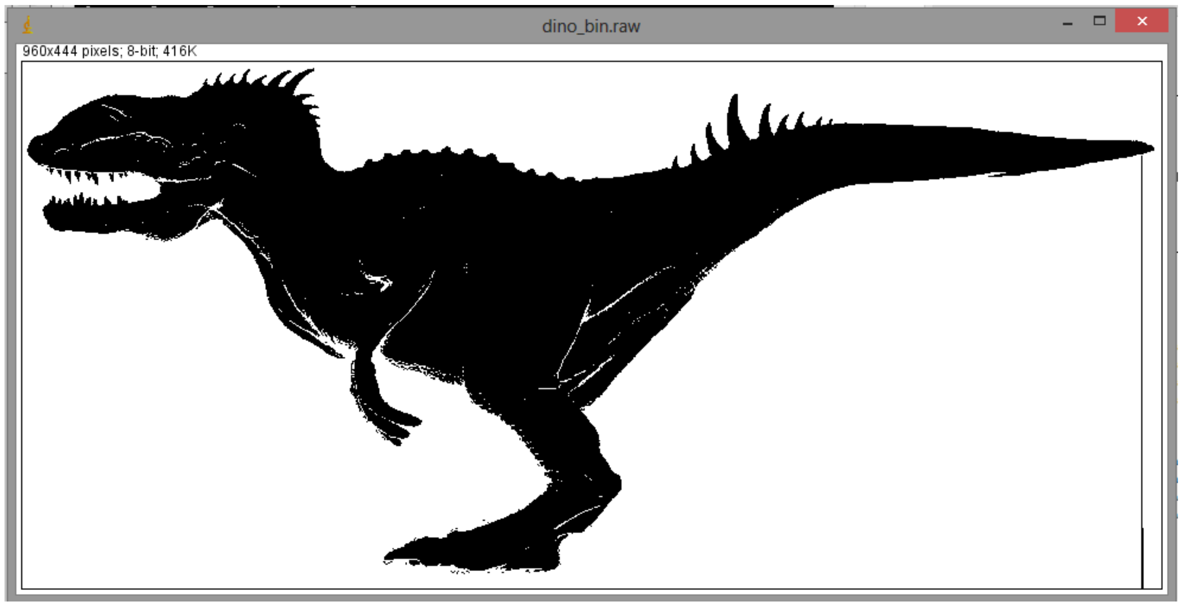


Thinning on test image test.raw

<u>P2.C Skeletonizing</u>

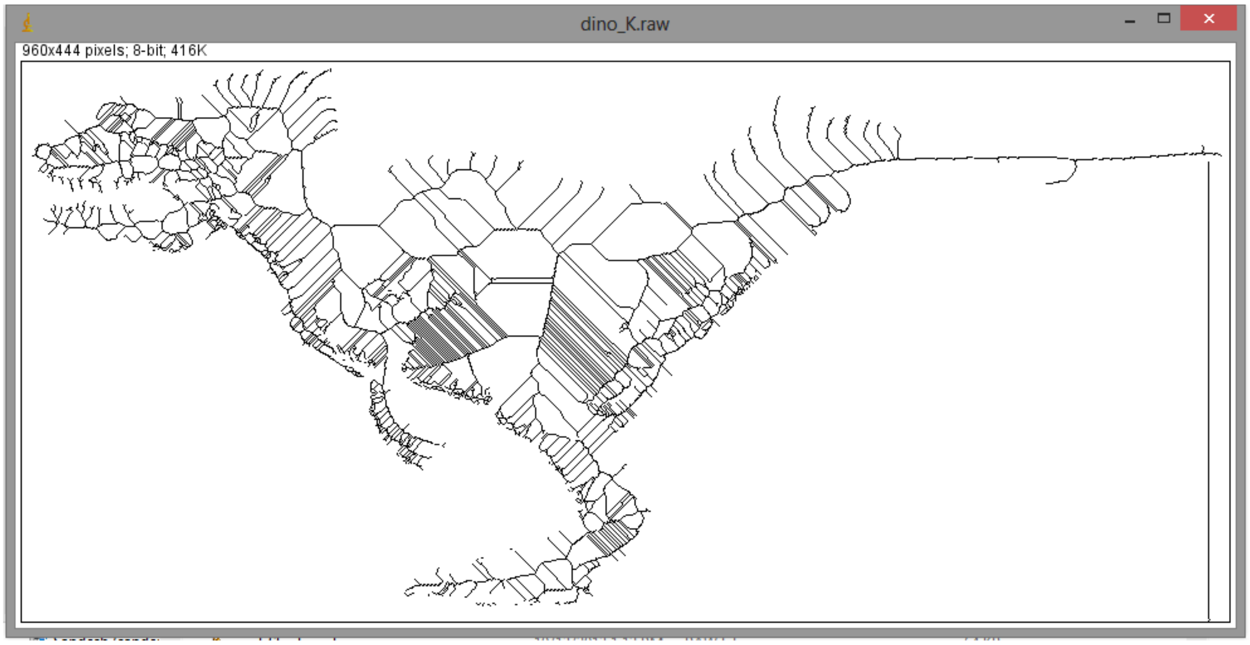
In this part of the problem, we skeletonize a given image of a dinosaur. The given image is in RGB gray level format. Again we follow similar procedures like in the previous two filter implementations and use the skeletonizing masks to skeletonize the image.

However after thresholding, the image was a binary image with many holes in between as follows.



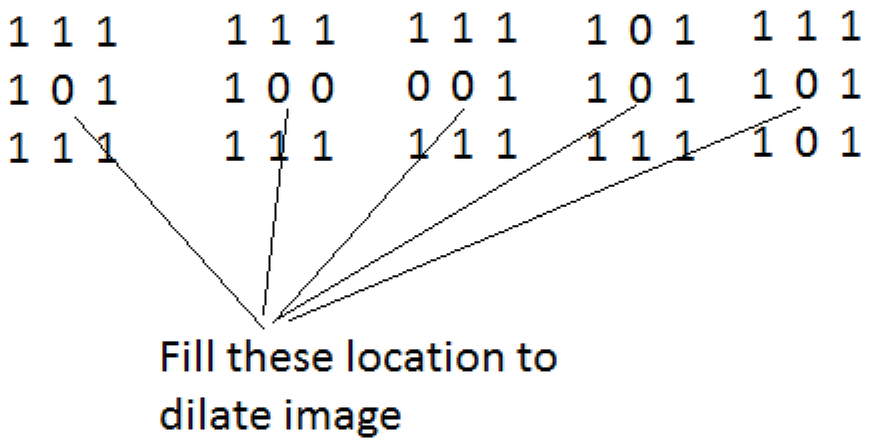
Thresholding at 65 shows many holes in the image

And the resulting skeleton is as follows



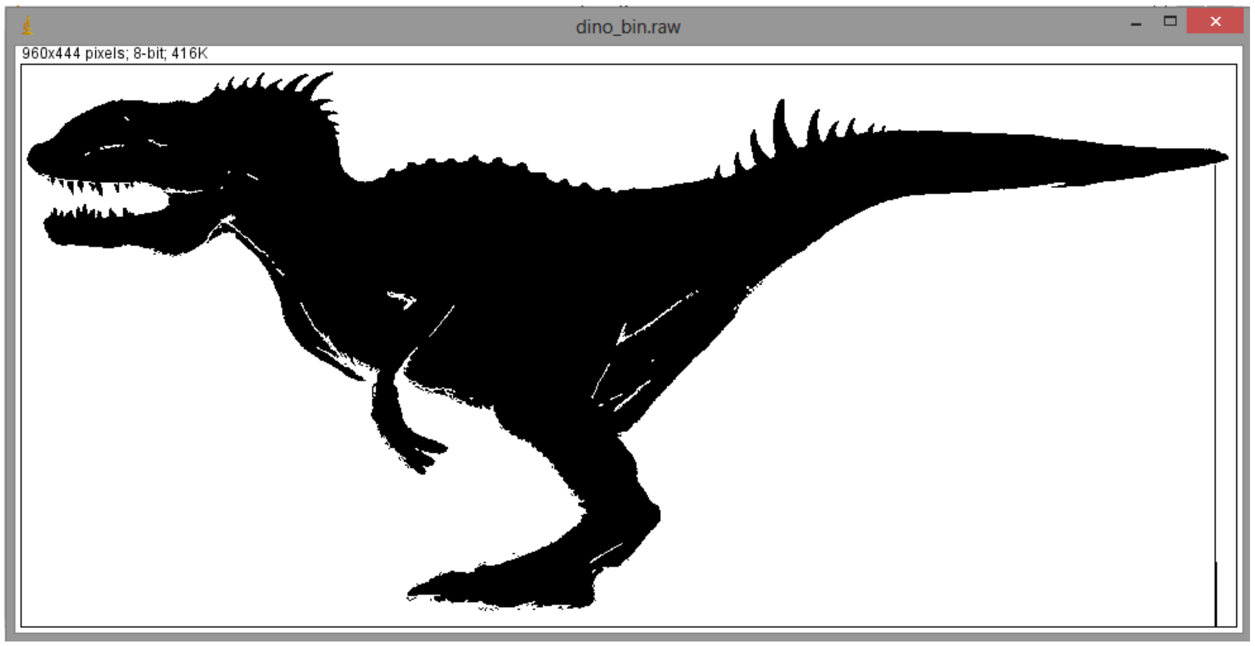
Resulting skeleton image

Hence we perform dialation to fill in the holes and after performing dilation as follows



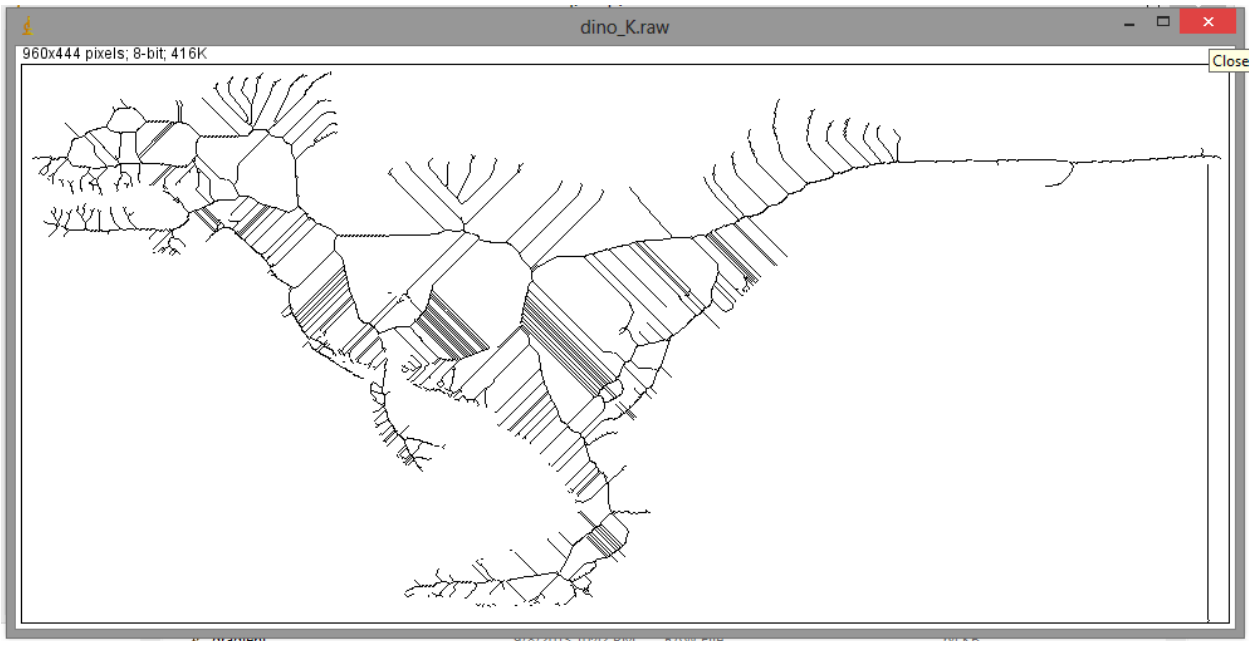
We use these masks to dilate the Image and remove the holes

However even after dialting 4 times we get



Binary after dilation of image 4 times

And the resulting skelton



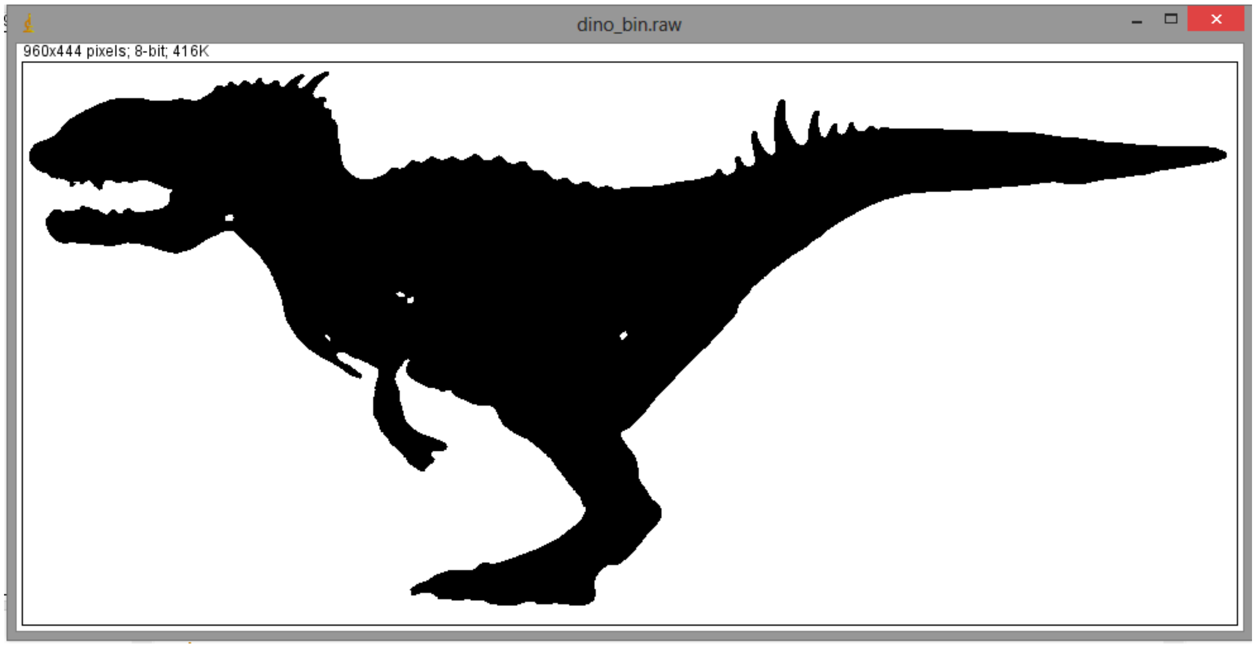
Skeletone image

We notice that as more holes are filled, the bones in the skeleton appear more realistic

Hence to completely fill the image, we use a median filter with window 3 to fill in the holes completely

And we get the following result

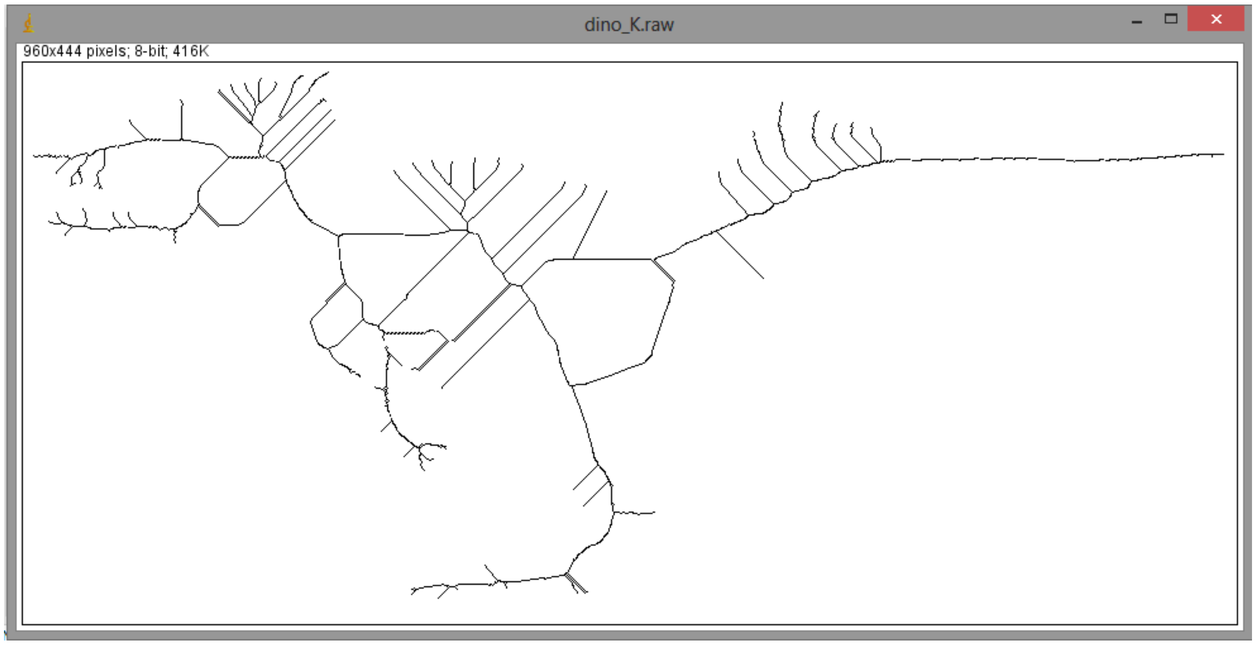
However the operation of median filter is very time expensive and takes some amount of time to complete even on a core i7.



Binary after median filter

We note that as we increase the window of filter, the hor ns and the protruding bone begin to disappear hence I have set a trade off between the smoothness and the number of holes in the image body to get an optimum skeleton

As follows



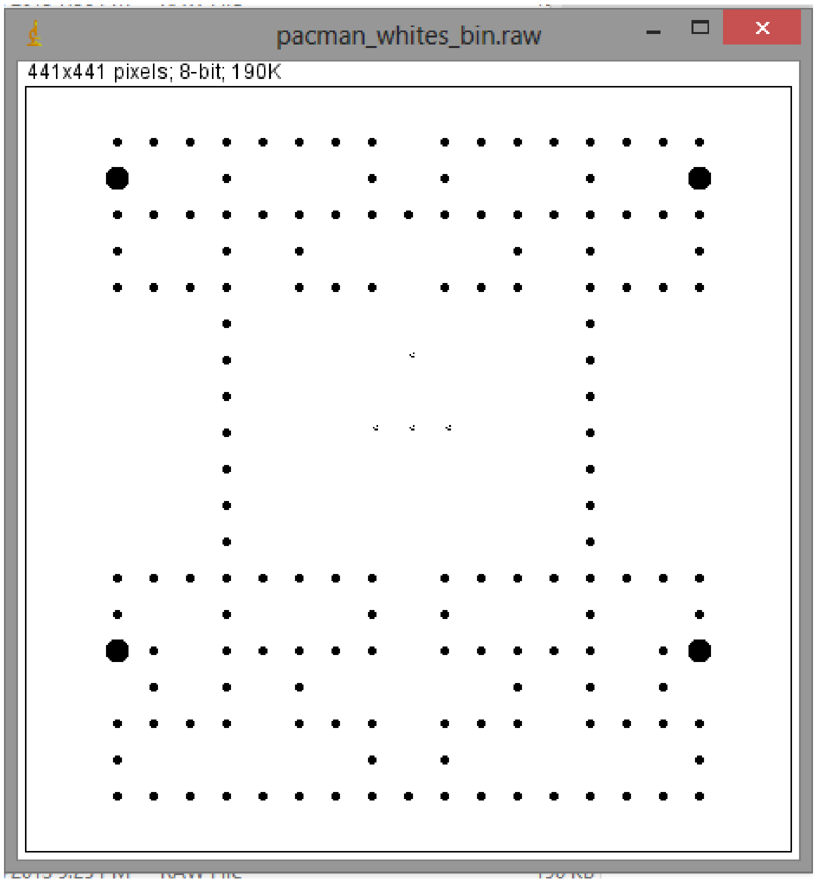
Skeleton after median filtering and 4 times dialation

<u>P2.C Pacman Game</u>

In this part of problem we have to determine the number of point balls in the packman,raw image, number of walls and number of turns

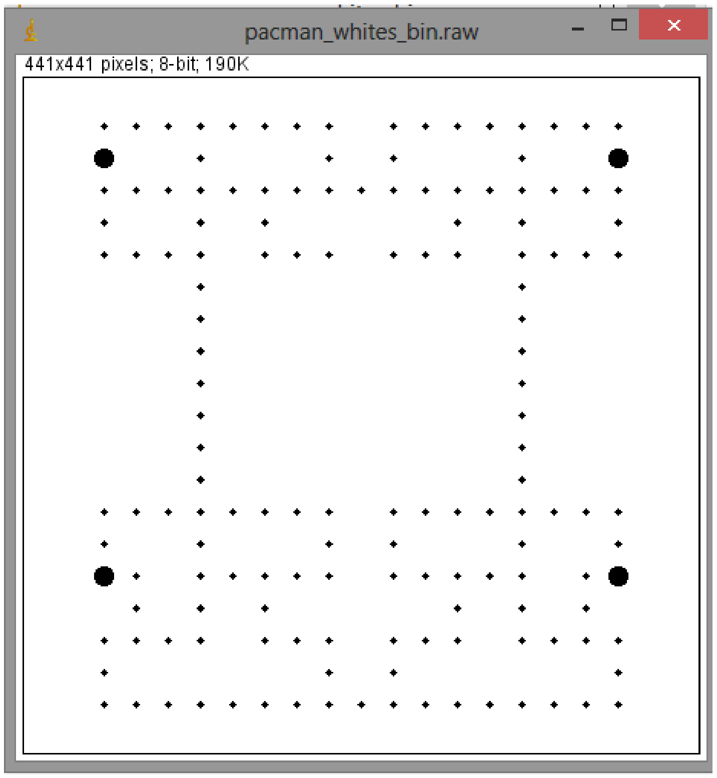
We begin this by converting into grayscale image.

Now for the pointballs we observe that all the balls including the bonus balls, have a majority of white color, hence we threshold at 255 for white balls and we obtain the following binary for white objects



Binary after threshoding white objects

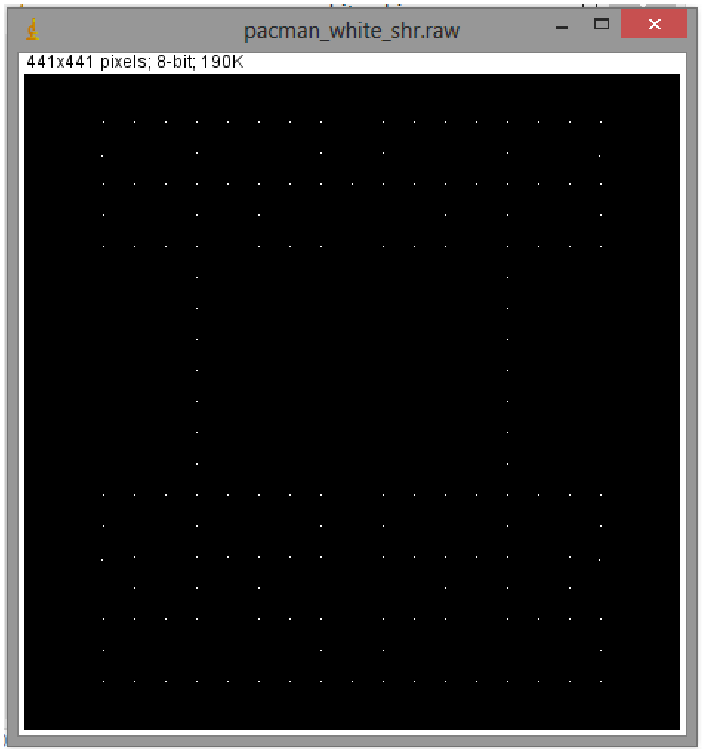
However we notice that the eyes of the enemy have also been captures and wee need to remove them this is done using the median filter again of window 2 to eliminate them. And we obtain the following binary after median filtering



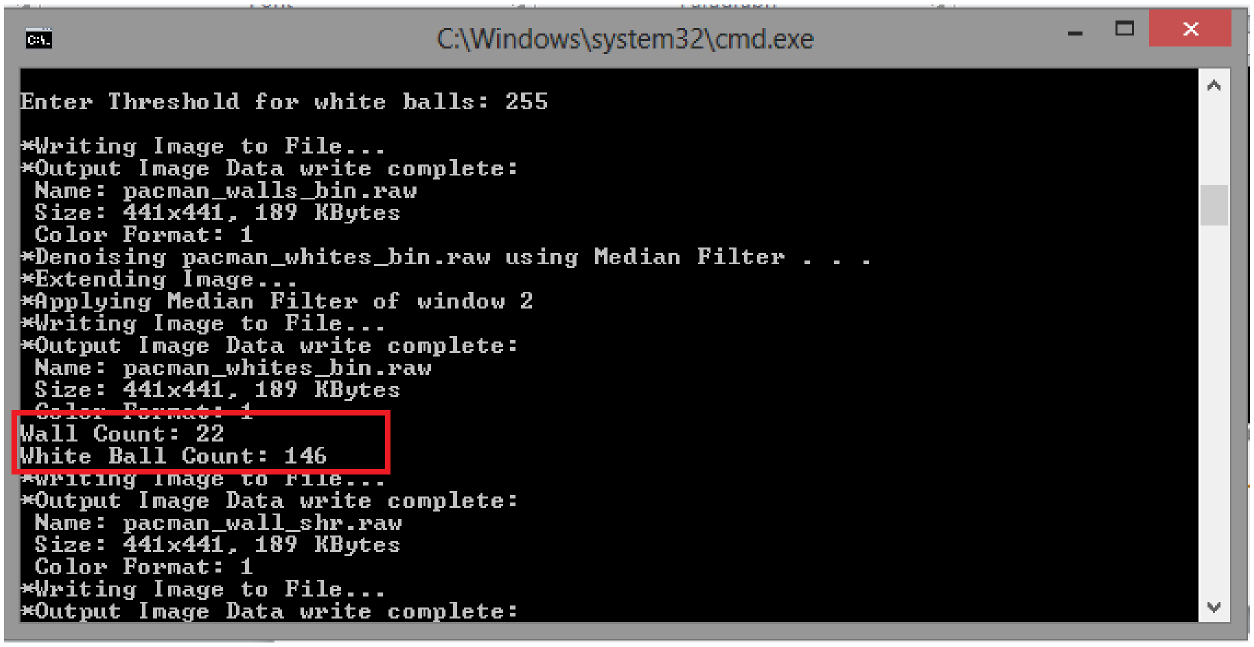
Binary after median filtering

Thus we see that the eyes and the unwanted white objects have been removed andwe just have the balls

Next we shrink this image to get a point size balls as follows

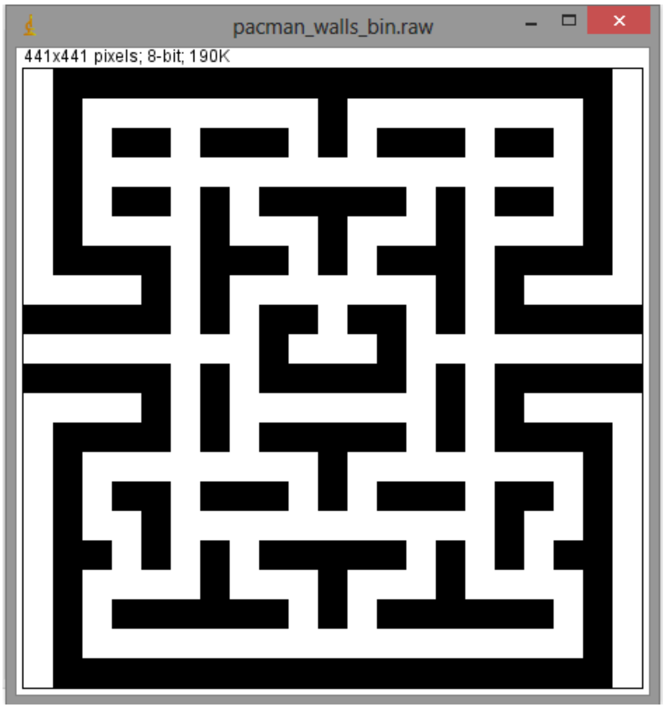


Then we count the total number of pixels and we minus the 4 as we have 4 large bonus balls as well thus we get the total number of ball count to be 146



```
Enter Threshold for white balls: 255

*Writing Image to File...
*Output Image Data write complete:
 Name: pacman_walls_bin.raw
 Size: 441x441, 189 KBytes
 Color Format: 1
*Denoising pacman_whites_bin.raw using Median Filter . . .
*Extending Image...
*Applying Median Filter of window 2
*Writing Image to File...
*Output Image Data write complete:
 Name: pacman_whites_bin.raw
 Size: 441x441, 189 KBytes
 Color Format: 1
Wall Count: 22
White Ball Count: 146
*Writing Image to File...
*Output Image Data write complete:
 Name: pacman_wall_shr.raw
 Size: 441x441, 189 KBytes
 Color Format: 1
*Writing Image to File...
*Output Image Data write complete:
```

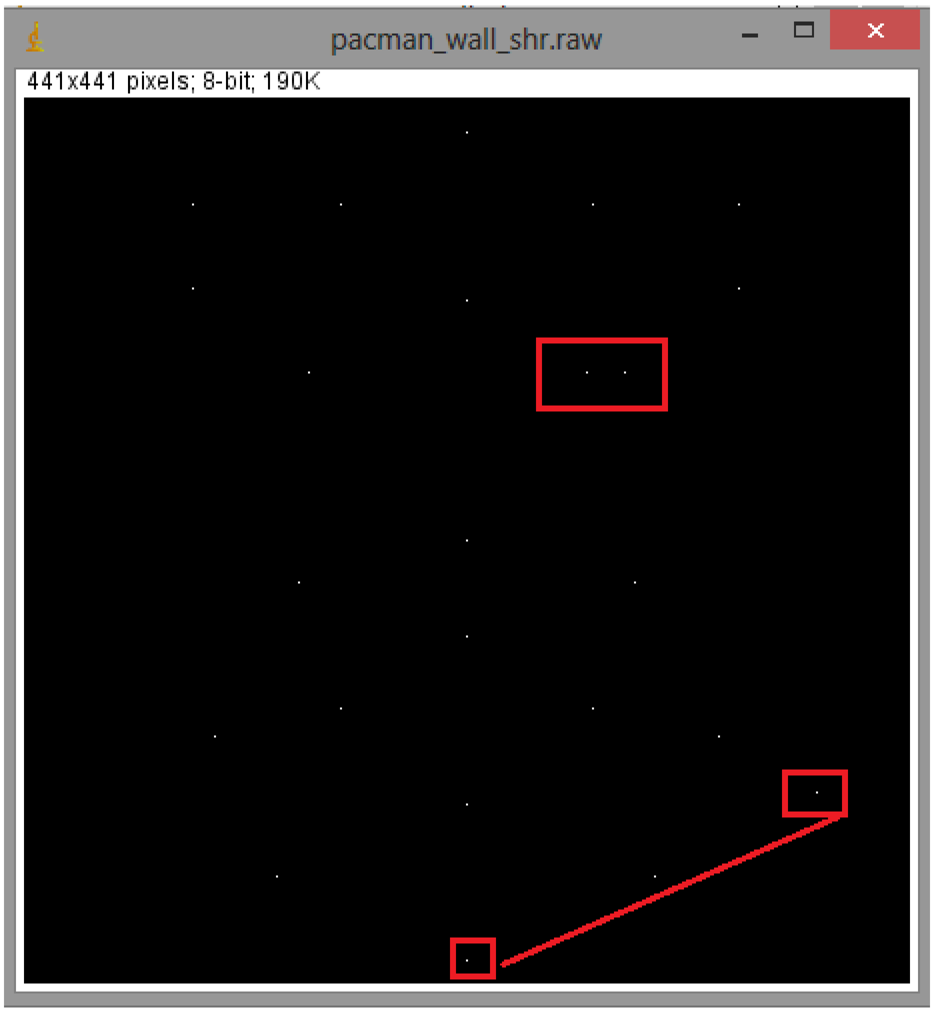
Now for walls we can follow similar approach and shrink all the walls into one pixel. For this we need to threshold just the wall. After observing the luminance histogram we find that at intensity 89 we have the maximum peak which must the walls. Thus thresholding walls at 89 we get



Binary for wall threshold at 89

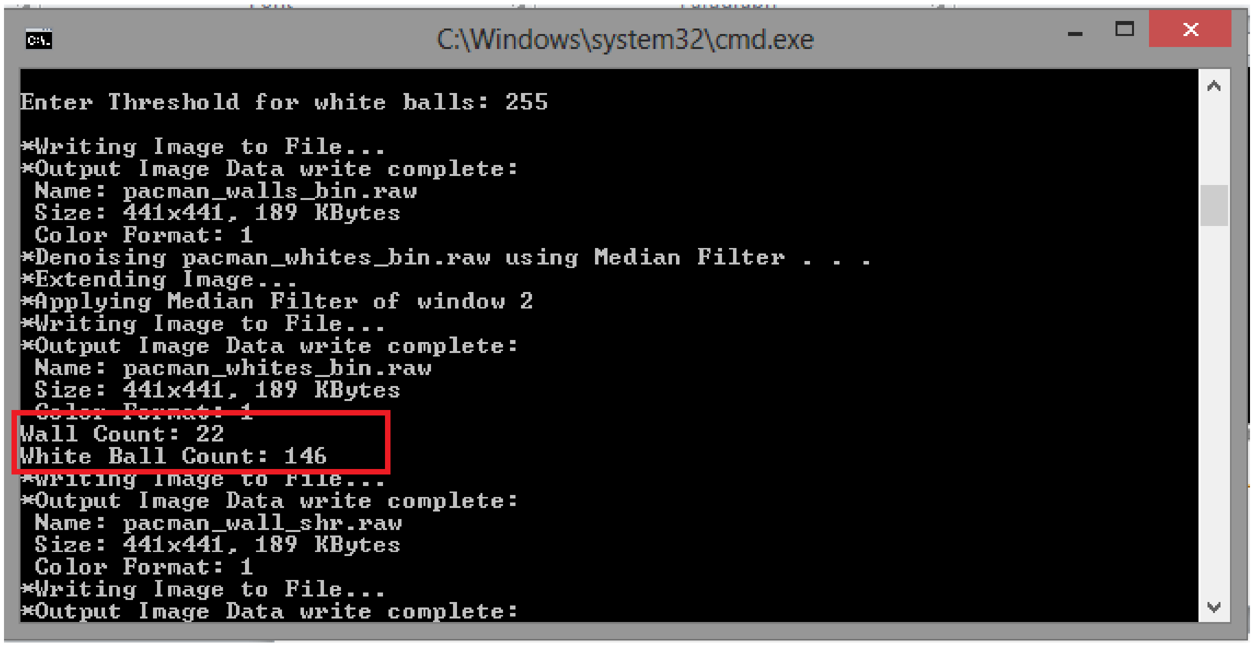
Thus now applying same logic we shrink these wall into one pixel and count the total number of pixels and get the wall count as 22.

However in the result of shrinking two walls gave me two dots as shown



Shrinking of walls gives this

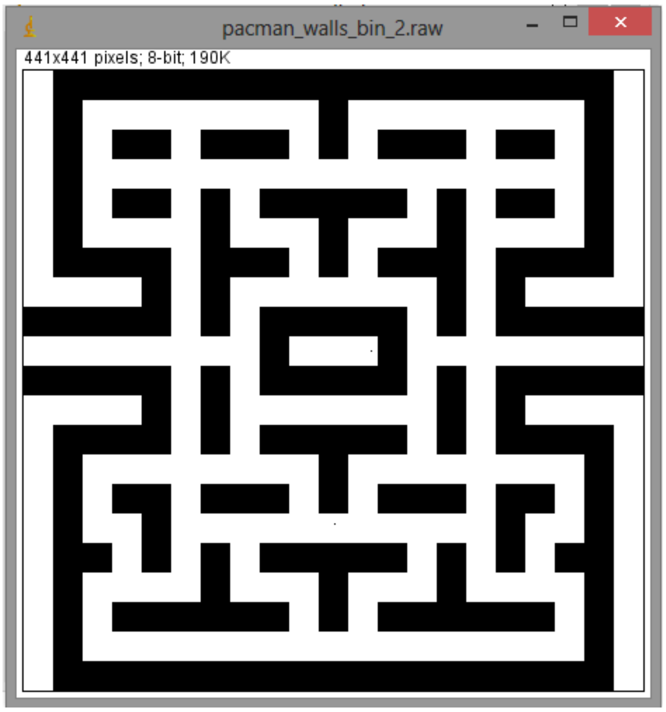
Hence from the total count of 24 I subtracted 2 to get the count of 22 walls



```
Enter Threshold for white balls: 255

*Writing Image to File...
*Output Image Data write complete:
 Name: pacman_walls_bin.raw
 Size: 441x441, 189 KBytes
 Color Format: 1
*Denoising pacman_whites_bin.raw using Median Filter . . .
*Extending Image...
*Applying Median Filter of window 2
*Writing Image to File...
*Output Image Data write complete:
 Name: pacman_whites_bin.raw
 Size: 441x441, 189 KBytes
 Color Format: 1
Wall Count: 22
White Ball Count: 146
*Writing Image to File...
*Output Image Data write complete:
 Name: pacman_wall_shr.raw
 Size: 441x441, 189 KBytes
 Color Format: 1
*Writing Image to File...
*Output Image Data write complete:
```

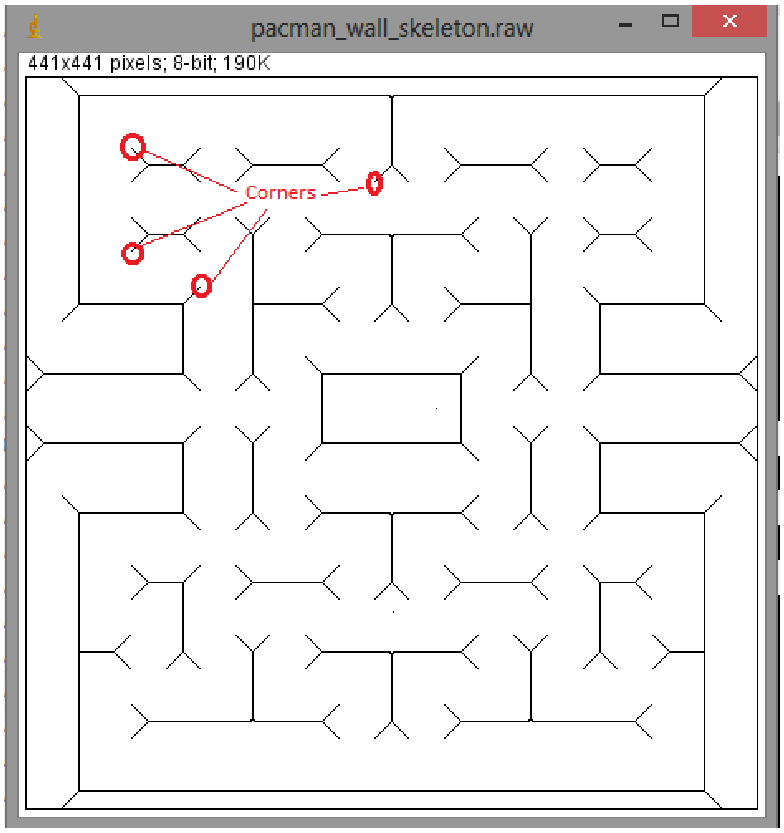
Result of wall count and ball count

Now for the turns we have the new threshold for walls as the pacman does not enter the center wall it must be sealed and by observing the value of the entrance we find that it is 141 and thresholding the walls at 89 and 141 will give us the walls where the pacman can actually go around as follows



Binary for walls thresholded at 141 and 89

Now If we skeletonize this image we get the following image of the skeleton of the walls



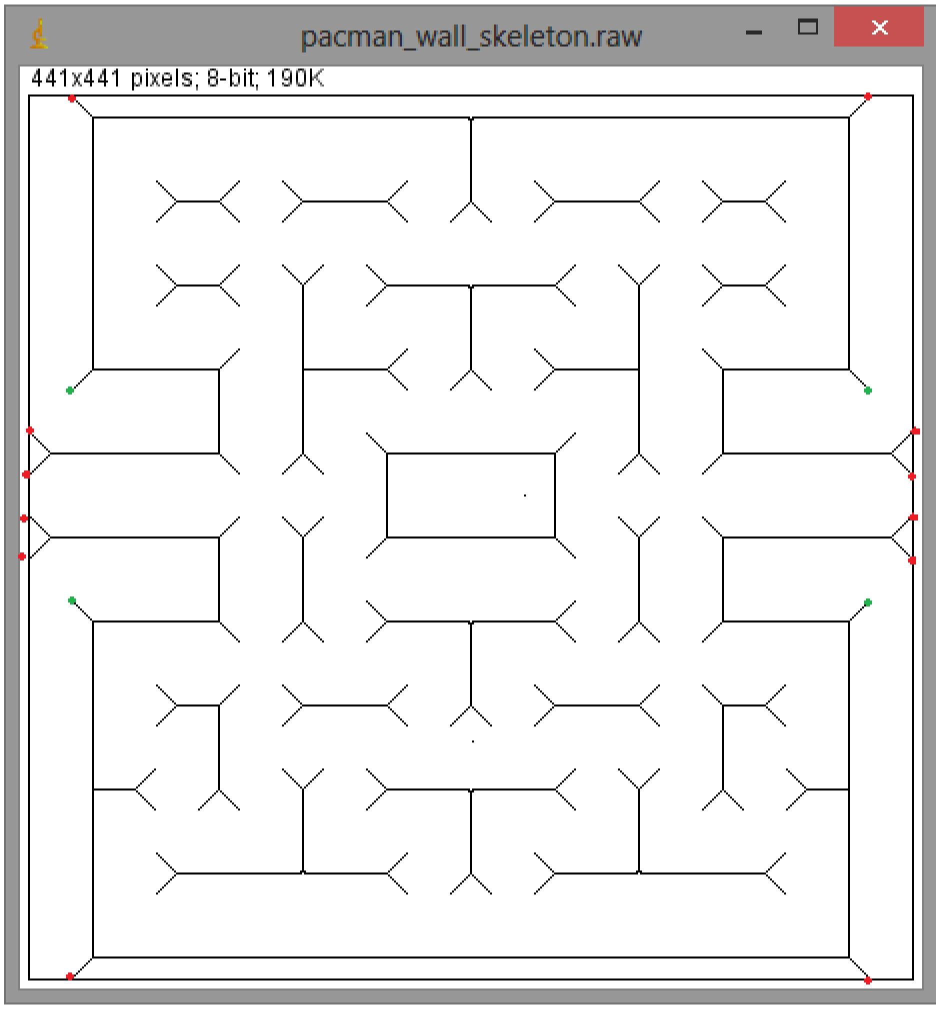
Skeleton of the walls showing pattern at the corner

Now we simply compare each pixel by a 5x5 window of the following patterns to mark it as a turn

```
10000   00001   00000   00000
01000   00010   00000   00000
00100   00100   00100   00100
00000   00000   01000   00010
00000   00000   10000   00001
```

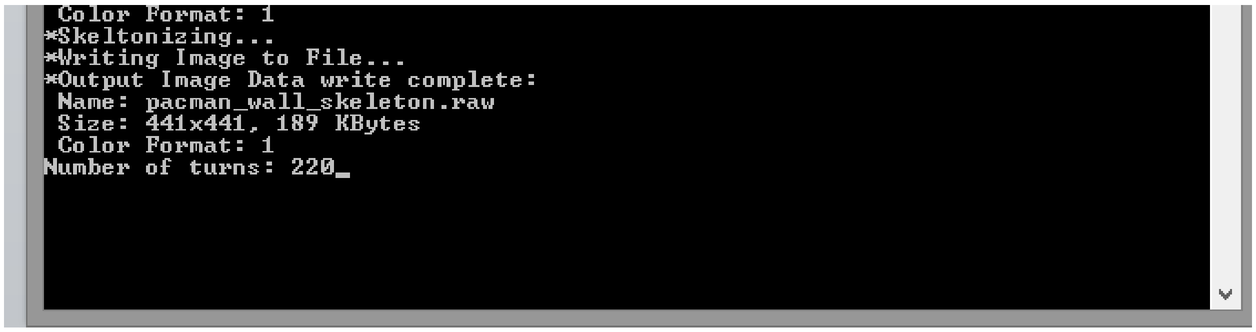
Turn patterns to identify

Each hit with this pattern will detect 2 turns, as we are starting from index location (2,2) we will not be dectecting the following corners which are indicated by red dots.



However we will be detecting the corners marked by the green dots, hence we need to subtract 4x2 = 8 corners from our final count to give the exact number of turns the pacman can make

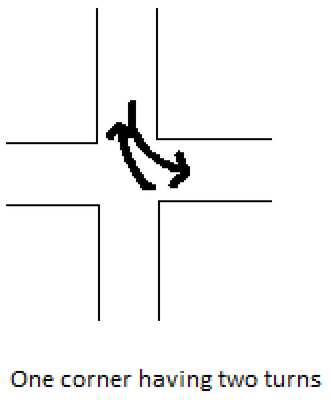
Thus we get 220 as the total number of turns



```
 Color Format: 1
*Skeltonizing...
*Writing Image to File...
*Output Image Data write complete:
 Name: pacman_wall_skeleton.raw
 Size: 441x441, 189 KBytes
 Color Format: 1
Number of turns: 220_
```

Result for total number of turns 220

This counting is based on the following logic that any corner will have two turns



Two turns for each corner

<u>Conclusion and Discussion:</u>

Thus in this problem we have seen differet morphological operations and their applications in given sample cases, to count the stars r digits or a complex application of the pacman.

In these operations we see that each operation yields a structure of the object which is consistant with the object, for example shrinking of a circle will be a dot. Thinning of a bar will be a line and skeletonization will give us a skeleton structure of the object in consideration.

While elimination plays an importan role in these methods, we must also make sure not to completely eliminate the object and hence we use the unconditional masks to make a choice of which candidate to consider and which to not for an even sized object.

Shrining reduces objects at around their Geometric Center. And this is seen in counting of stars.

Thinning of objects with symmetry from all side will be similar to shrinking as seen from test images examples

Problem: 3 Digital Halftoning

Motivation:

Digital halftining is a method by which we represent a given image by only two tones or colors so that when viewed from a large distance, it looks similar to the original image. Digital halftoning in extensivel used in offset printing where many colors are represented using very limited colors. We look at many different ways of halftoning digitally using dithering and error diffusion. We also look at the inverse process of halftoning on how to generate original image representation from its digital halftone image.

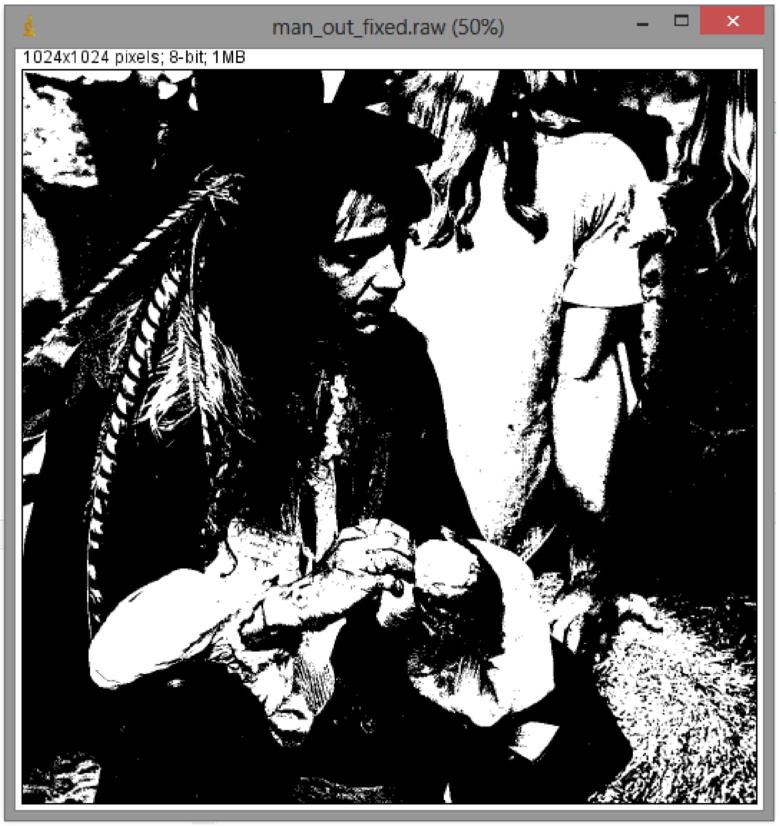
Approaches and Procedure and results:

P3.A Dithering

In this part of problem we are given with a grayscale image of man.raw and we need to implement three different ways to halftone this image into representation using only black and white pixels. We have the following three methods of halftoning.

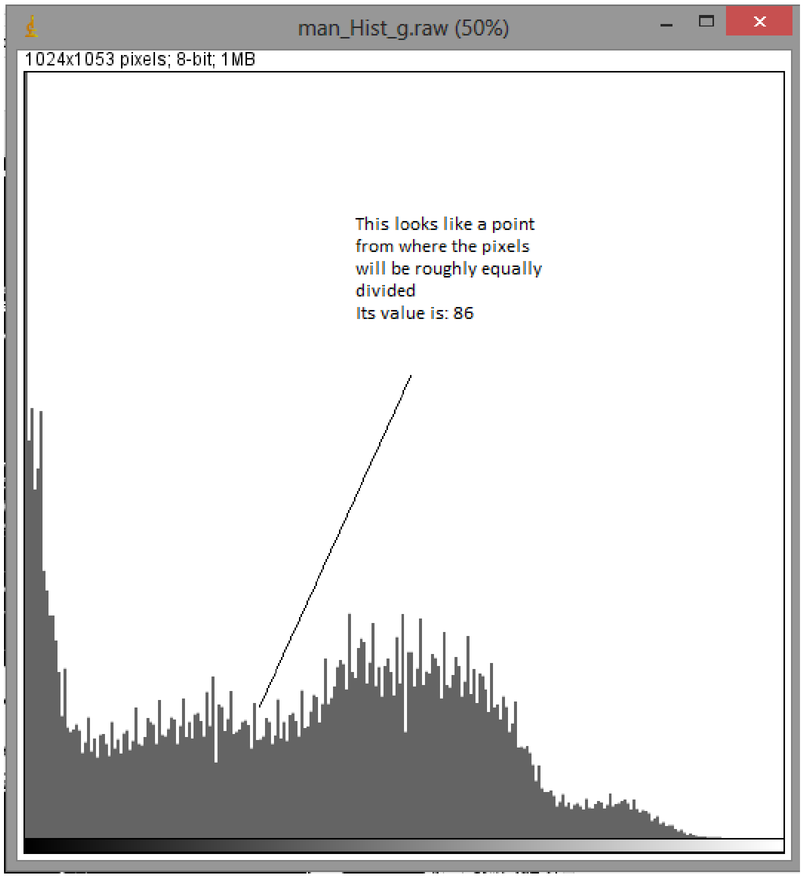
1) Fixed Threshold Selection

In this method we select a particular threshold for diving the image into black and white pixels. A most obvious choice of threshold for the scale of 0 to 255 would be 127. Which gives the following result



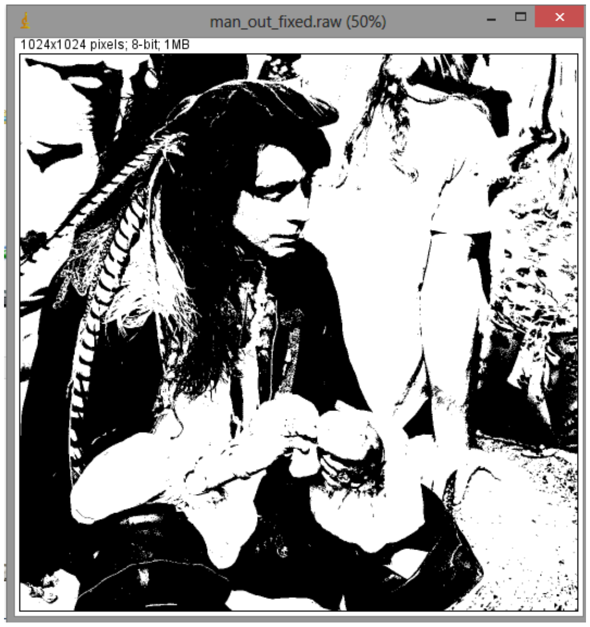
Halftone image of man.raw at 127 thresholding

However if we look at the histogram and see that we want to divide the pixels into black and white equally, then the threshold selected will be different for each image and it will be how the pixels are distributed in the histogram. Observing the histogram we see the following



Histogram of man.raw

On selecting 86 as threshold to get a even distributon of pixels we get the following halftine image



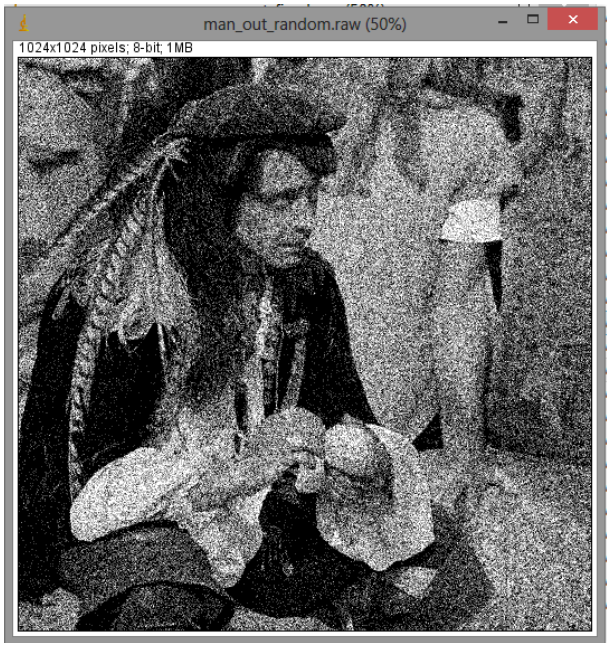
Halftone image on a fixed threshold of 86 to get even distribution

We see that this halftine image is much more better than the previous halftone image

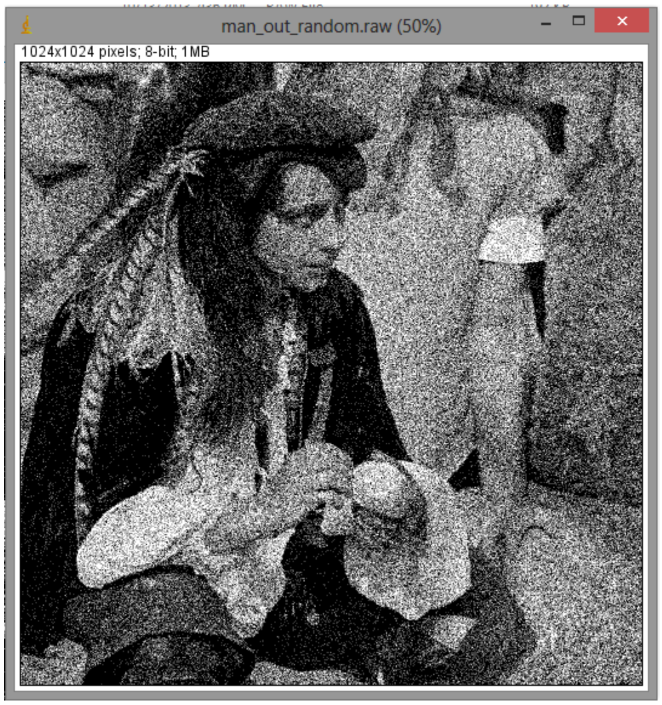
2) Random Thresholding

Random thresholding involves a random selection of threshold in between 0 to 255 in a uniform way so that we get a white noise induced image after thresholding as follows

This is done using the rand() in C with two different seed



Random threshold selection for seed 255



Random threshold selection for seed 127

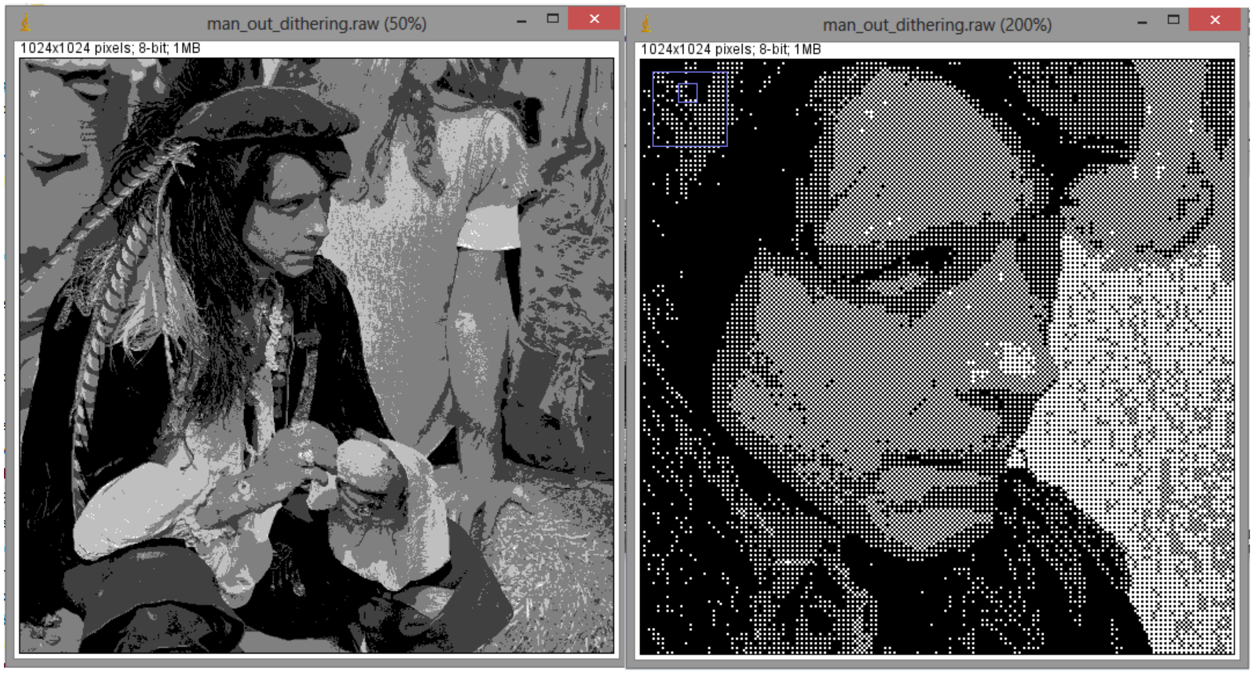
We see that despite using different seeds for random number generation we still get similar noisy image which is halftoned.

3) Dithering

Next we use the dithering matrix to decide the threshold and halftone the image on the basis of that threshold.

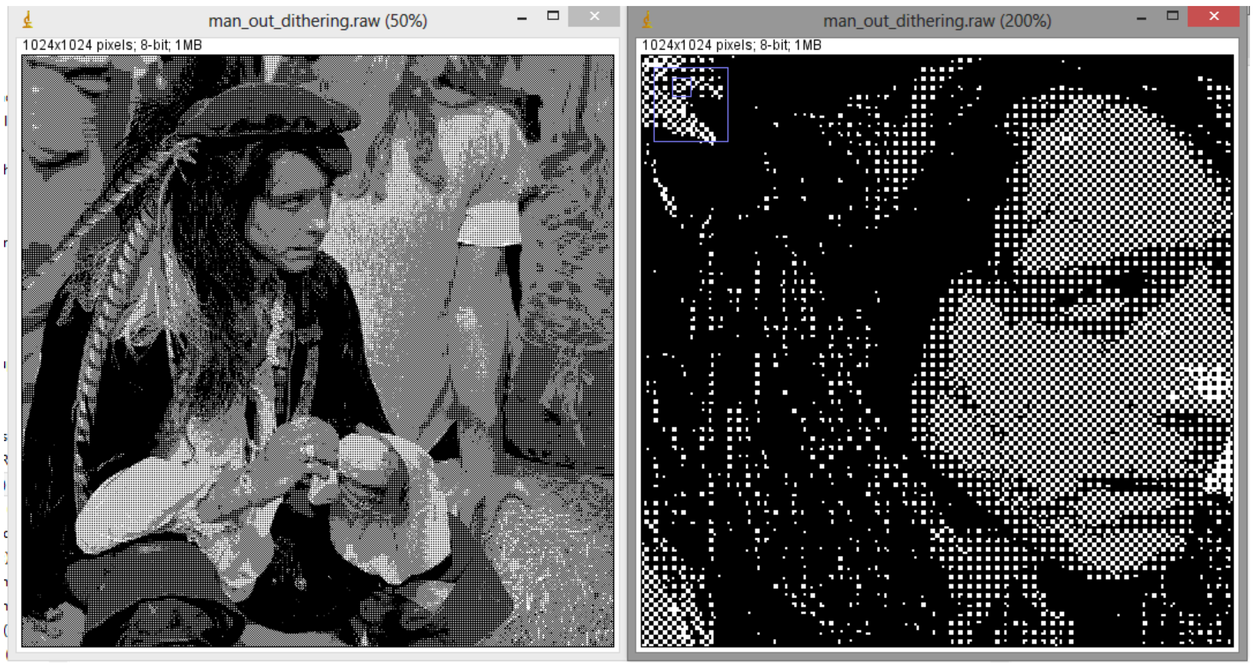
The following is the 2x2 THresholding matrix

And we get the following as the halftone output from 2x2 matrix


man_out_dithering.raw (50%)
1024x1024 pixels; 8-bit; 1MB
man_out_dithering.raw (200%)
1024x1024 pixels; 8-bit; 1MB

Image(left) Imaged zoomed to show the halftone(right)

Similarly we can generate a 4x4 dithering matrix recursively from the 2x2 matrix according to the given formula in the homework sheet to get a better result as shown below

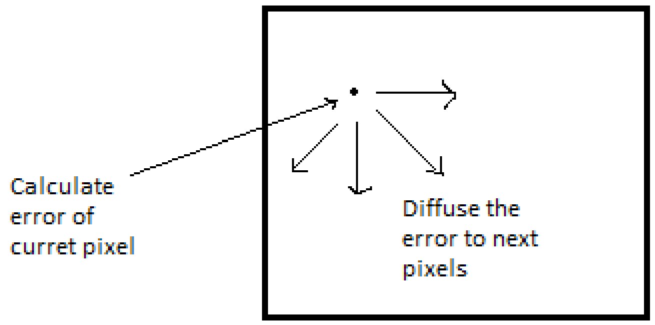

man_out_dithering.raw (50%)
1024x1024 pixels; 8-bit; 1MB
man_out_dithering.raw (200%)
1024x1024 pixels; 8-bit; 1MB

Image(left) Image zoomed in to show halftones(right)

Thus we see that the dithering is used principally to convert a gray level image into a halftone image. Using the dithering matrix improves image quality over random thresholding and fixed thresholding methods of halftoning.

P3.B Error Diffusion

In this method of halftoning we calculate the error of a pixel value from its halftone value and diffuse this error to the future neighbors of the pixel in order to get a smooth halftone effect in the output image. There are following different ways to diffuse or distribute the error to the next pixels of the image.


Calculate error of curret pixel
Diffuse the error to next pixels

Error diffusion method.

The 3 mentioned ways to diffuse the error around the next pixels are as follows

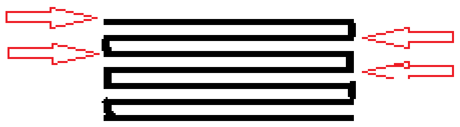
1) Floyd-Steinberg error diffusion with serpentine scanning

This method uses a 3x3 diffusion matrix which looks as follows

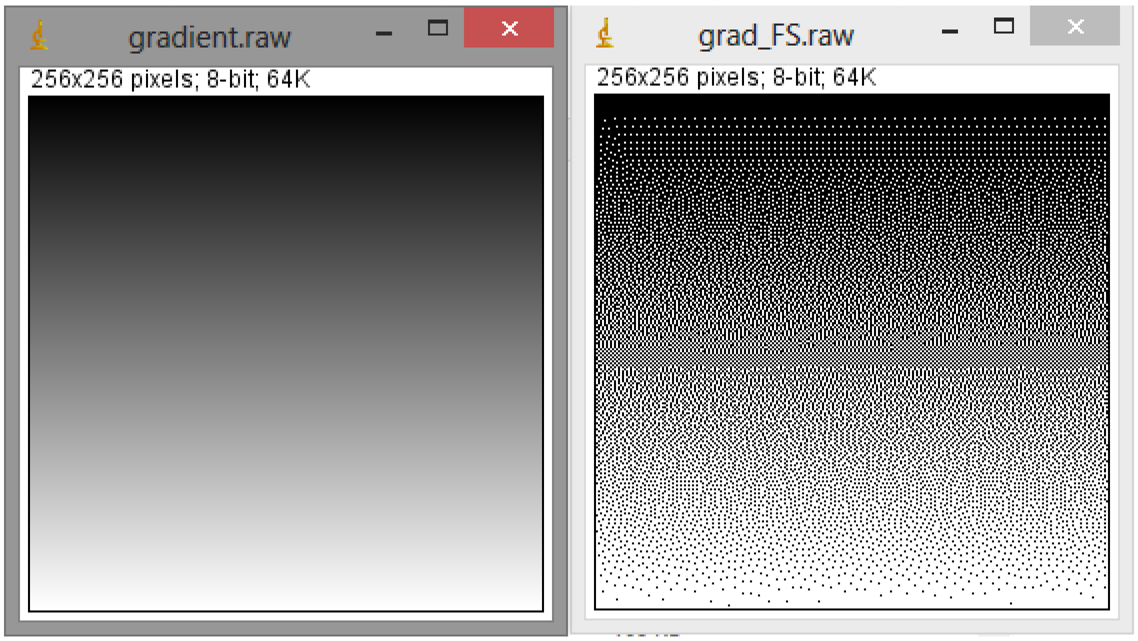
$$\frac{1}{16}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 7 \\ 3 & 5 & 1 \end{bmatrix}$$

Diffusion Matrix for Floyed Steinber method

Serpentine scanning avoid accumulation of error on the sides of image as the error is being diffused it will be diffused equally between all the pixels by this method of scanning


Serpentine scanning method

After implementing this method over the gradient.raw image we get the following output



Input(left) and output(right) using Floyed Steinberg matrix for diffusion

2) Jarvis, Judice and Ninke Error diffusion (JJN)
This method gave another way to diffuse error more efficiently over the neighbor pixels(future) by another diffusion matrix given by

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

JJN Error Diffusion Matrix

Again implementing this using the serpentine scanning method we get the following results

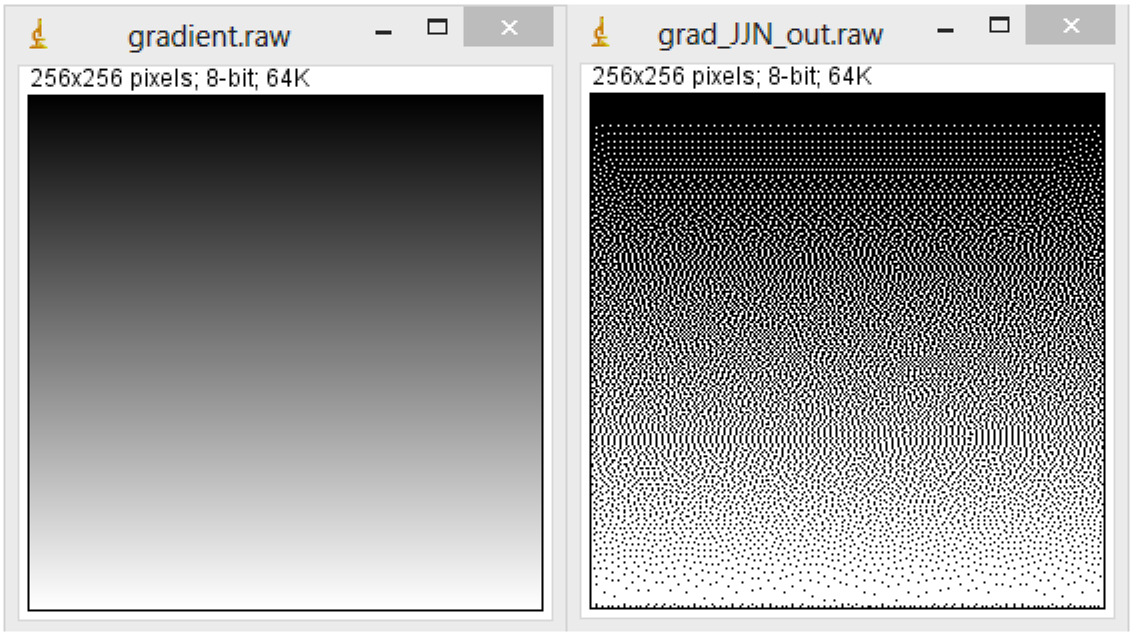


Image input(left) output of JJN diffusion(right)

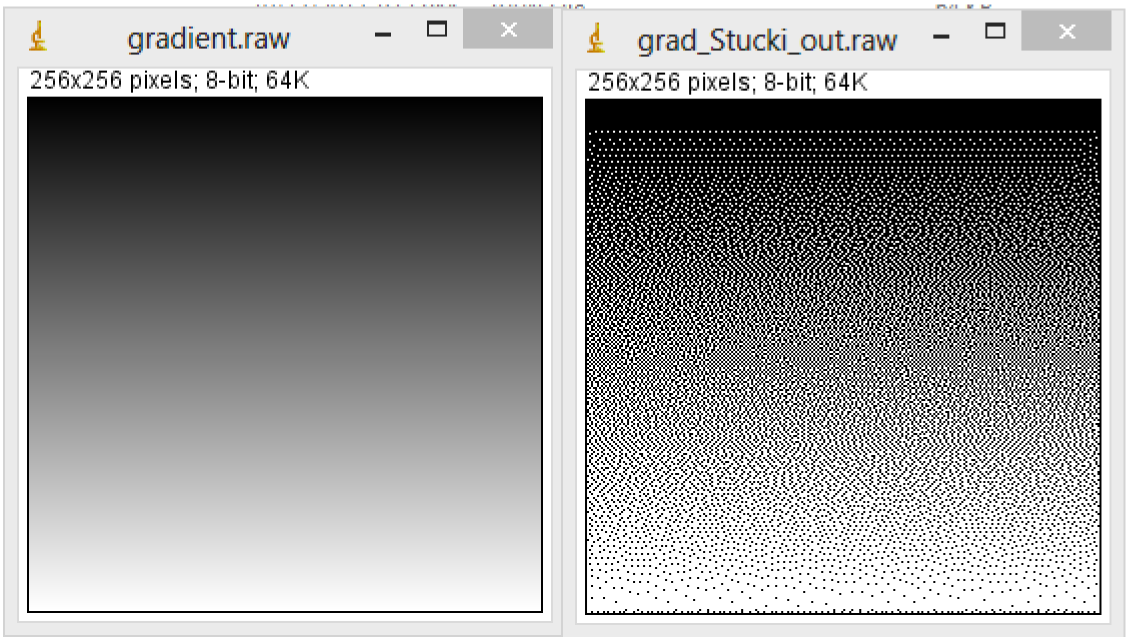
3) Stucki Error Diffusion

This involves another diffusion matrix as follows

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

Stucki error diffusion matrix

Implementing which we get the following result in the serpentine scanning fashion



Input image(left) Stucki error diffusion output(right)

In all the cases we observe a halftoning of gradient distribution performed. We will discuss the comparisons further

<u>P3.C Inverse Halftoning</u>

Inverse halftoning is the process by which we generate the original monochrome image from the halftone image. According to the paper[1], the inverse halftoning can be done in the following two ways

1) Using just the binary image

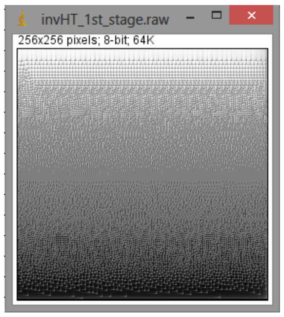
In this method, the Adaptive Binary Runlength is used to find runs of zeros and ones in the halftone image and then represent them in their gray scale distribution by the given formula on page 145 eq 5 and 6.

These runs are calculated for rows and coloumns individually and then they are averaged together.

This method only uses the runs of ones and zeros and adaptively distributes the gray level within these locations.

This forms the first stage of the cascaded algorithm mentioned in figure 6 of [1]

After implementing this on the output of JJN Error Diffusion we have the following as output of ABRL method



It can be however noticed that though the image manages to appear according to the original image, it has many "impulses" which are due to the fact that run lengths of the consecutive ones or zeros distribute the gray level within that length only. And hence we get impules in between the region where transition takes place.

2) Cascade algorithm
In the next part of the paper, as mentioned in the block diagram, we can remove these impulses by using a 5x5 window to have an adaptive low pass filter which will be dependent on the contents of the window itself. And this makes it sensitive to the local variance within the window.

This stage will calculate the variance and mean of the local window and depending on it, it will filter out spatially the values using the following formula

G''(j,k) = MU + (SIGMA^2)/((SIGMA^2 + K)*(G'(j,k) – MU)

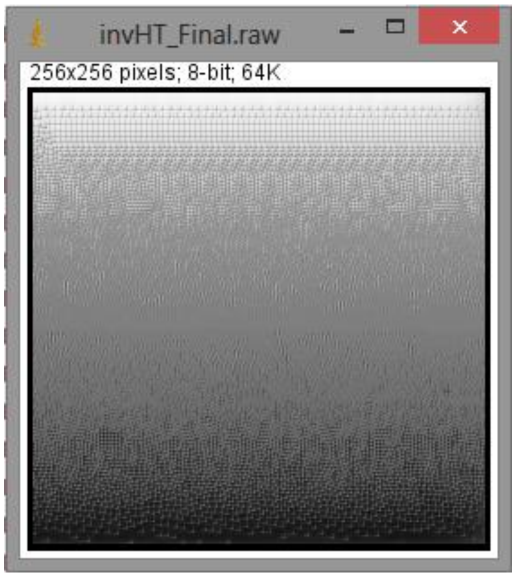
Where G''(j,k) is the filtered output of this stage,
MU is mean and SIGMA is standard deviation of the local window being considered, K = 800 given
And G'(j,k) is the output of the first stage.

After this stage next block gives us removal of remaining impulses using MAXVAR which is the maximum variance or it is considered as adjustable threshold on the variance calculated every time with the 5x5 window beneath which it will adjust the pixels given by the equation on page 148 which involve BETA another parameter with valued given by 30,40,50

Finally dynamic range mapping is required, but in our case this has been handled after the first case itself where G'(j,k)the ouput of first stage is mapped to 0 to 255.

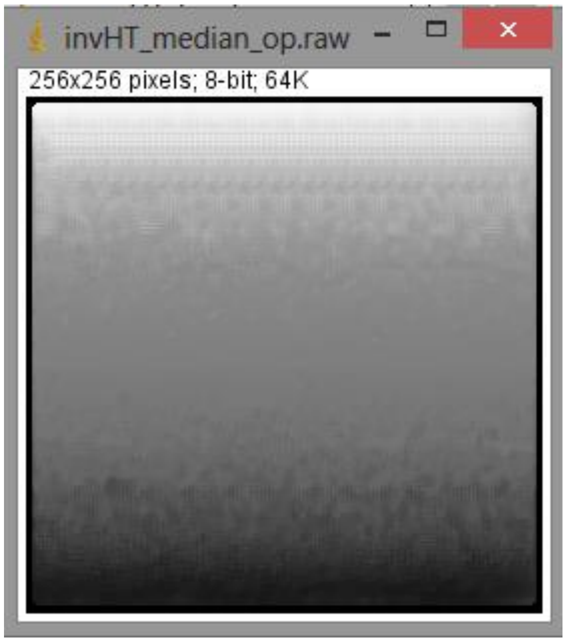
Also since we are taking a 5x5 window, we are not extending the image by 2 pixels, hence we might get a black boundary of 2 pixel width around thr output image.

Thus
We have the ouput of the final stage as follows.



Output of final stage.

We can further improve this by performing median filter of window 5x5 over this image which yields the following



Median filter output

Conclusion and Discussion:

In this problem we looked at different methods of halftoning and the inverse as well. Digital halftoning done using a fixed threshold will very rarely give a desired output, hence we must go for approaches which are much more efficient. Among them is the noise addition which improves the perceived tonal distribution but it increases the noise content in the image.

Dithering is a good solution to random thresholding, where in we calculate which pixel is mostl likely to be changed to 0 or 255 depending upont the parameters of the dithering matrix.
Other method to convert to halftone image is error diffusion where we calculate how much error does a pixel have over its threshold and then we distribute this error to the future neighbor pixels in order to have a smooth looking halftone distribution, we saw 3 different type of diffusion matrices which allowed us to diffuse the errorto next pixels. The Floyed Steinberg, JJN and Stucki matrices. Out of which the JJN matrix gave the most efficient performance as seen from the result, the FS and Stucki matrices gave a band of perceived gray tone in between the image which formed a contour like feature, however thr JJN matrix efficiently halftoned the image and did not gave any such feature and its out put was the most resembeled one.

Finally we also looked at the inverse halftone methods to obtain the original image from a halftone image using the algorithm mentioned and presented in [1]. This algorithm is divided into two parts, the first part generates a rough gray estimate of the original image from the runs of zeros and ones in rows and coloumns and the next part uses a local adaptive filter which smoothens the output of first stage to give a better result, also we saw at the output being passed through a median filter to give a further better resemblance to the original image.

References

[1] C. M. Miceli and K. J. Parker, \Inverse halftoning," Journal of Electronic Imaging, vol. 1, no. 2, pp. 143{151, 1992.