Homework 2 Report

// Date : 16 February 2018

// Name : Tamoghna Chattopadhyay

// ID : 8541324935

// email : tchattop@usc.edu

Problem 1.(a)

I. Abstract and Motivation

In this problem we were asked to apply geometrical modifications to the image. We had to use a spatial warping technique to transform an input square image into an output disk image. The wrapped image should satisfy three conditions:

- Pixels that lie on boundaries of the square should still lie on the boundaries of the circle.
- The center of original images should be mapped to the center of warped images.
- The mapping should be reversible, i.e. it is a one-to-one mapping.

Next, we had to apply a reverse spatial warping to each warped image to recover back the original square image and check for discrepancies.

In Geometrical Modifications, we generally manipulate the pixel position in cartesian coordinate and keep the intensities unchanged using a transformation matrix. Operations like translation, scaling and rotation come under these manipulations.

II. Approach and Procedures

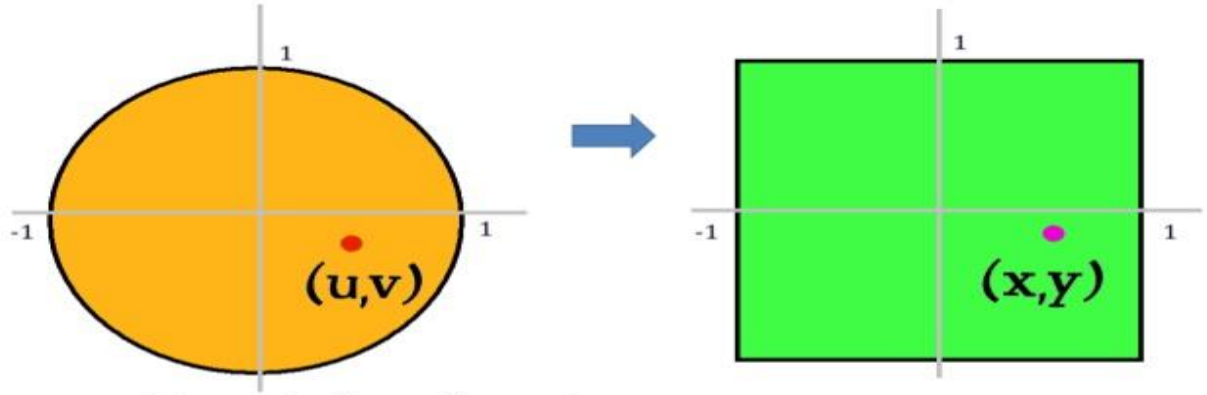
I found a research paper online by Mr. Chamberlain Fong titled “Analytical methods for squaring the disc” in which he gives various methods and the formulas to convert a square image to a disc image and vice versa. I decided to use the C++ language to write the code for the problem statement.
The starting point of the operation was to convert the square image from 512*512 size to a unit square with x axis ranging from -1 to 1 and y axis ranging from +1 to -1. This is because the formula presented by Mr Fong works on those ranges. Then I applied the formula for conversion and then again changed the size of the image to 512*512.

I used the Elliptical Grid Mapping method described in the Paper.

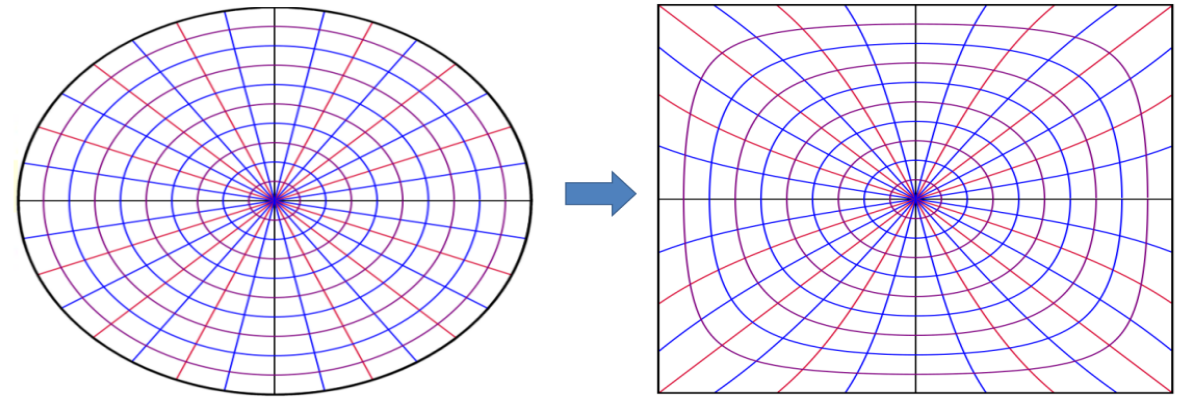
How to map every point in the unit disc to a square region?

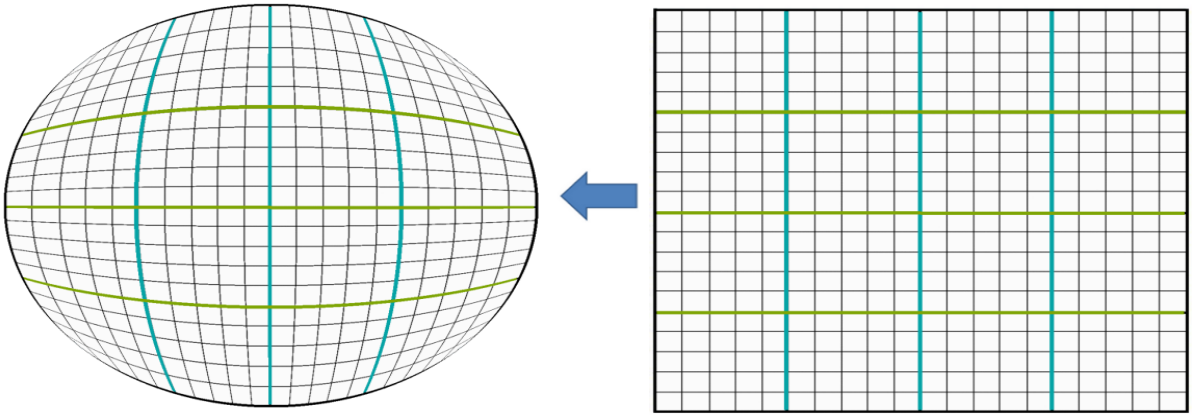
$(x, y) = f(u, v)$

mapping $f: \mathbb{R}^2 \to \mathbb{R}^2$ $\quad (u, v) = f^{-1}(x, y)$



domain: $\{(u, v) \mid u^2 + v^2 \leq 1\}$ range: [-1, 1] × [-1, 1]





Disc to square mapping:

$$x = \frac{1}{2}\sqrt{2 + u^2 - v^2 + 2\sqrt{2}\,u} - \frac{1}{2}\sqrt{2 + u^2 - v^2 - 2\sqrt{2}\,u}$$

$$y = \frac{1}{2}\sqrt{2 - u^2 + v^2 + 2\sqrt{2}\,v} - \frac{1}{2}\sqrt{2 - u^2 + v^2 - 2\sqrt{2}\,v}$$

Square to disc mapping:

$$u = x\sqrt{1 - \frac{y^2}{2}} \qquad v = y\sqrt{1 - \frac{x^2}{2}}$$

III. Result and Discussion

Original Image - Panda

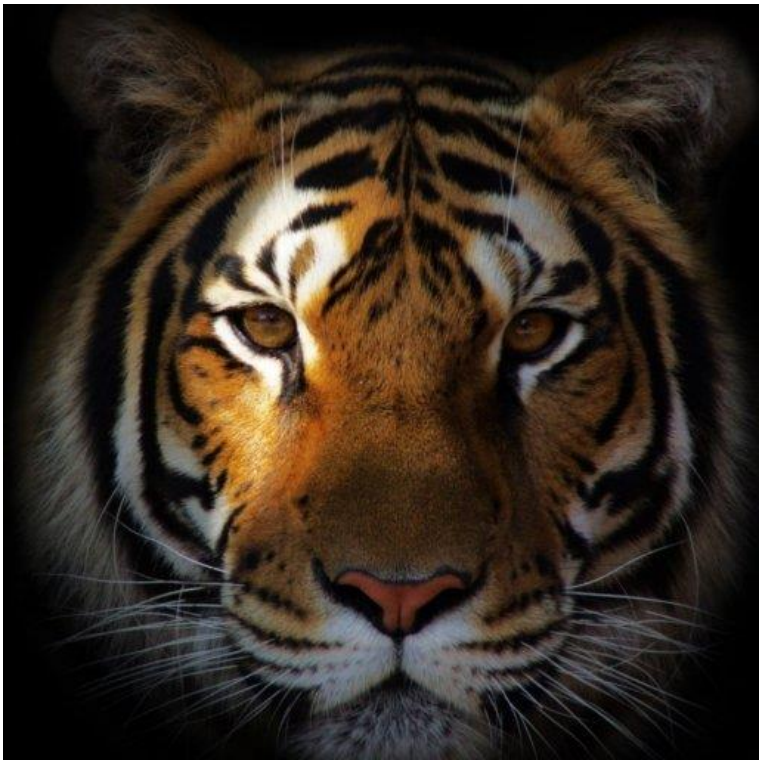
Image converted to disc

Disc Image converted back to a square image.

Original Image - Puppy

Image converted to disc

Disc image converted back to a square image.



Original Image - Tiger

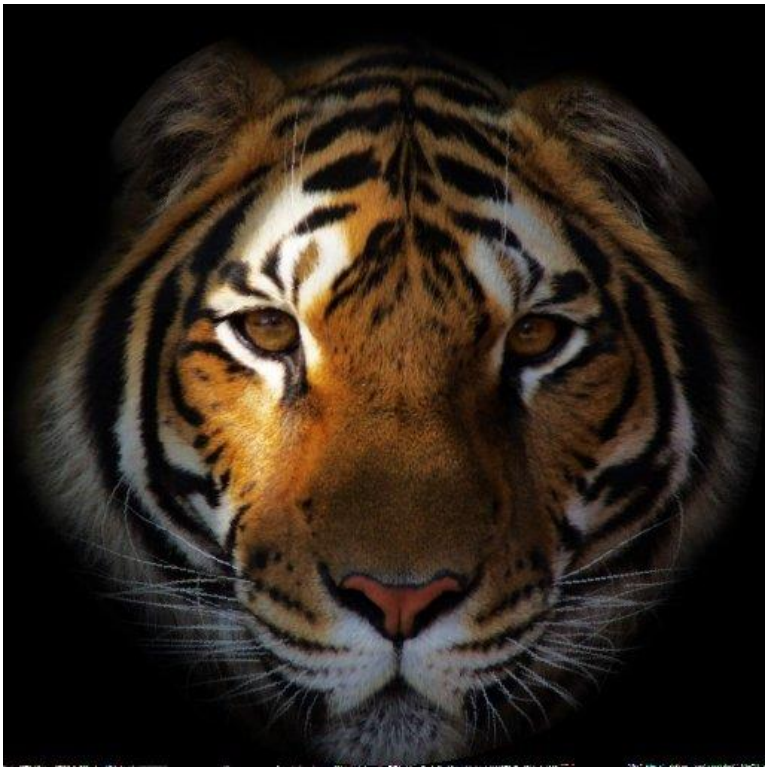
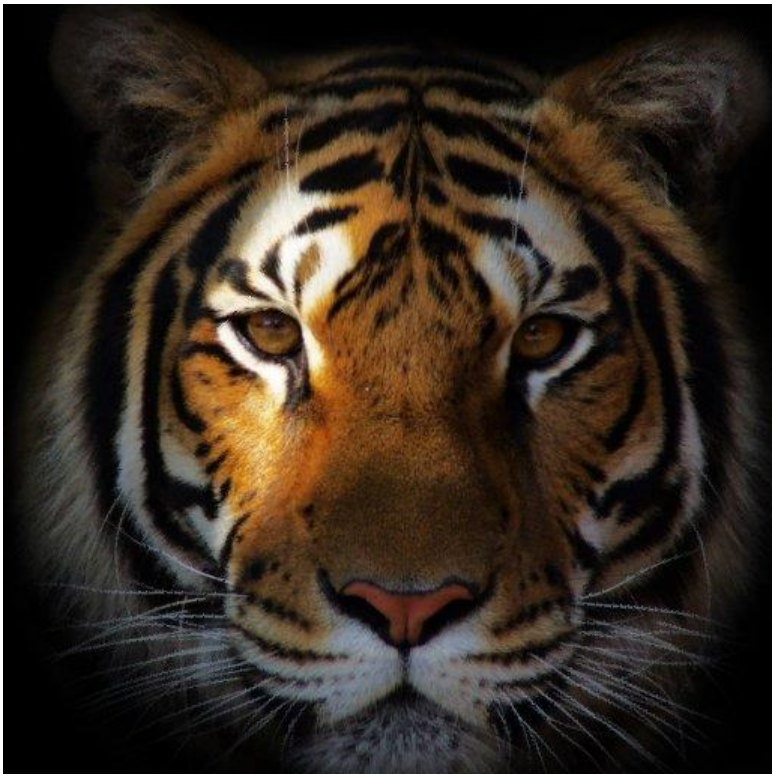


Image converted to disc



Disc Image converted back to square image.

In this case backward mapping was used when converting from a disc to a square image.

The recovered image is almost similar to the original image given. There are some minute differences around the edges which are caused by the loss of information when converting from the square to the

disc image. When the conversion tom circle takes place, the image is shrunk across the edges to convert into a disc. This leads to the loss of information as some of the pixel intensities are lost.

Problem 1.(b)

I. Abstract and Motivation

In this problem , we have to use homographic transformation and image stitching techniques to create a panorama that consists of multiple images. We were given three images: left, middle and right and had to combine them into a single image.

II. Approach and Procedures

I used C++ to write the code. I had to use dynamic memory allocation in this code as the image sizes were too big to store the image data in the stack memory. So image data was stored in the heap and dynamically worked with.

The first step in creating the panorama is to select multiple control points which are same in two images. So here, we first pasted the middle position in a big black image at desired position. Then we compared this image with both left and right images, and found out four common points for both the images to calculate the homography matrix.

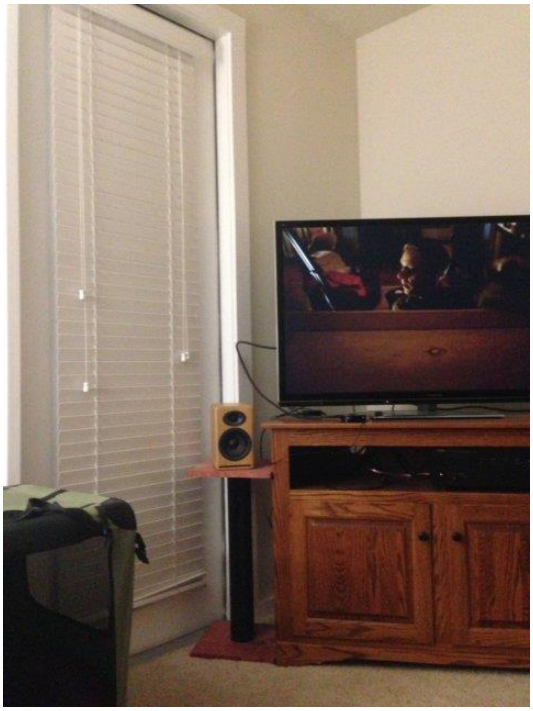
```
P = [       x(1,1) x(1,2) 1 0 0 0 (-1*(x(1,1)*x1(1,1))) (-1*(x(1,2)*x1(1,1)));
            x(2,1) x(2,2) 1 0 0 0 (-1*(x(2,1)*x1(2,1))) (-1*(x(2,2)*x1(2,1)));
            x(3,1) x(3,2) 1 0 0 0 (-1*(x(3,1)*x1(3,1))) (-1*(x(3,2)*x1(3,1)));
            x(4,1) x(4,2) 1 0 0 0 (-1*(x(4,1)*x1(4,1))) (-1*(x(4,2)*x1(4,1)));
            0 0 0 x(1,1) x(1,2) 1 (-1*(x(1,1)*x1(1,2))) (-1*(x(1,2)*x1(1,2)));
            0 0 0 x(2,1) x(2,2) 1 (-1*(x(2,1)*x1(2,2))) (-1*(x(2,2)*x1(2,2)));
            0 0 0 x(3,1) x(3,2) 1 (-1*(x(3,1)*x1(3,2))) (-1*(x(3,2)*x1(3,2)));
            0 0 0 x(4,1) x(4,2) 1 (-1*(x(4,1)*x1(4,2))) (-1*(x(4,2)*x1(4,2)))      ];

b=[x1(1,1);x1(2,1);x1(3,1);x1(4,1);x1(1,2);x1(2,2);x1(3,2);x1(4,2)];

H=P\b;
```

Here, the coordinates from the left and right image are denoted by x and the ones from middle are denoted by x1. H gives the matrix value. We select four points from each images to build linear equations and solve them. We normally set H33 = 1 so that there are only 8 parameters to be calculated. Once this matrix is got, we can project all points from one image to the middle by the formulae:

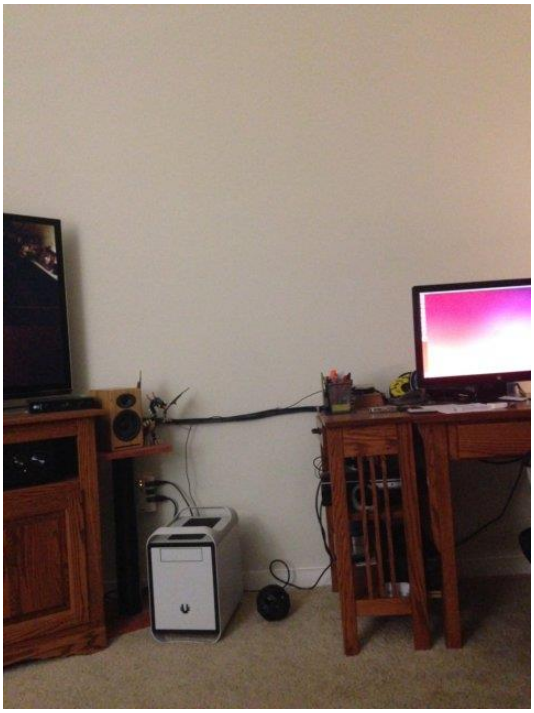
```
x = (H[0][0]*i + H[0][1]*j +  H[0][2]*1) / (H[3][0]*i +  H[3][1]*j +  H[3][2] * 1) ;
y = (H[1][0]*i + H[1][1]*j +  H[1][2]*1) / (H[3][0]*i +  H[3][1]*j +  H[3][2] * 1) ;
```

Here, normalization is being done for the third coordinate and H is the matrix to be applied. I and J are the coordinates from the image. This is done for both the left and right images and they are pasted on the big middle image we created.

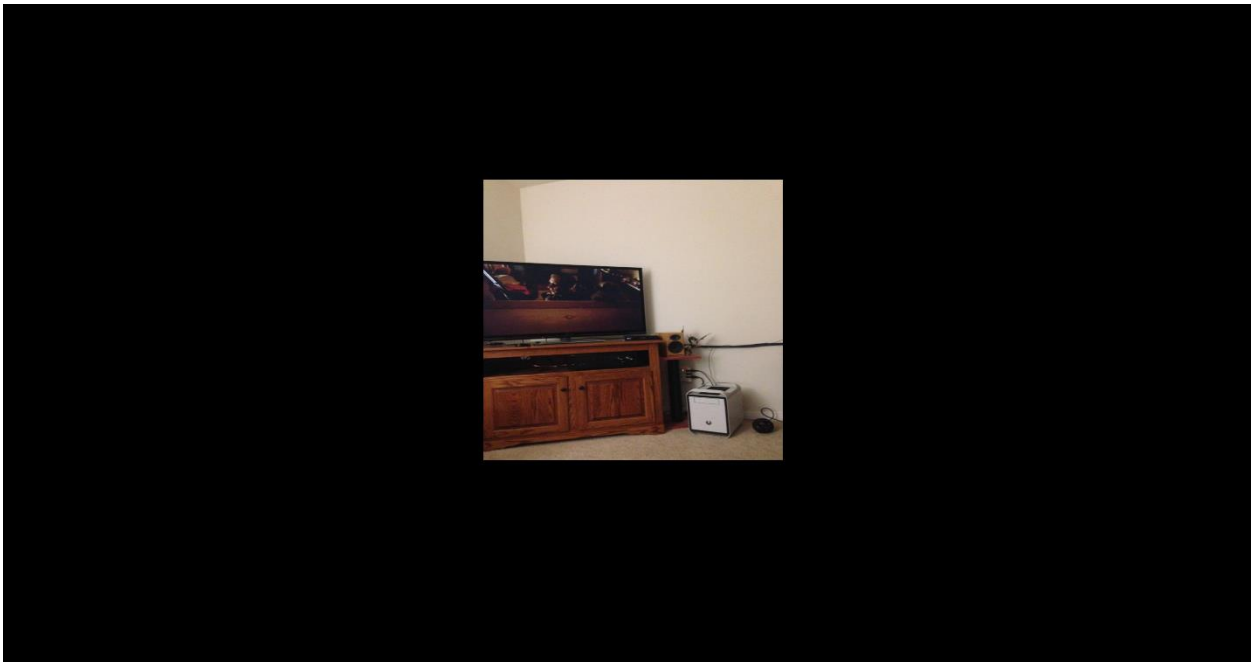
III. Result and Discussion



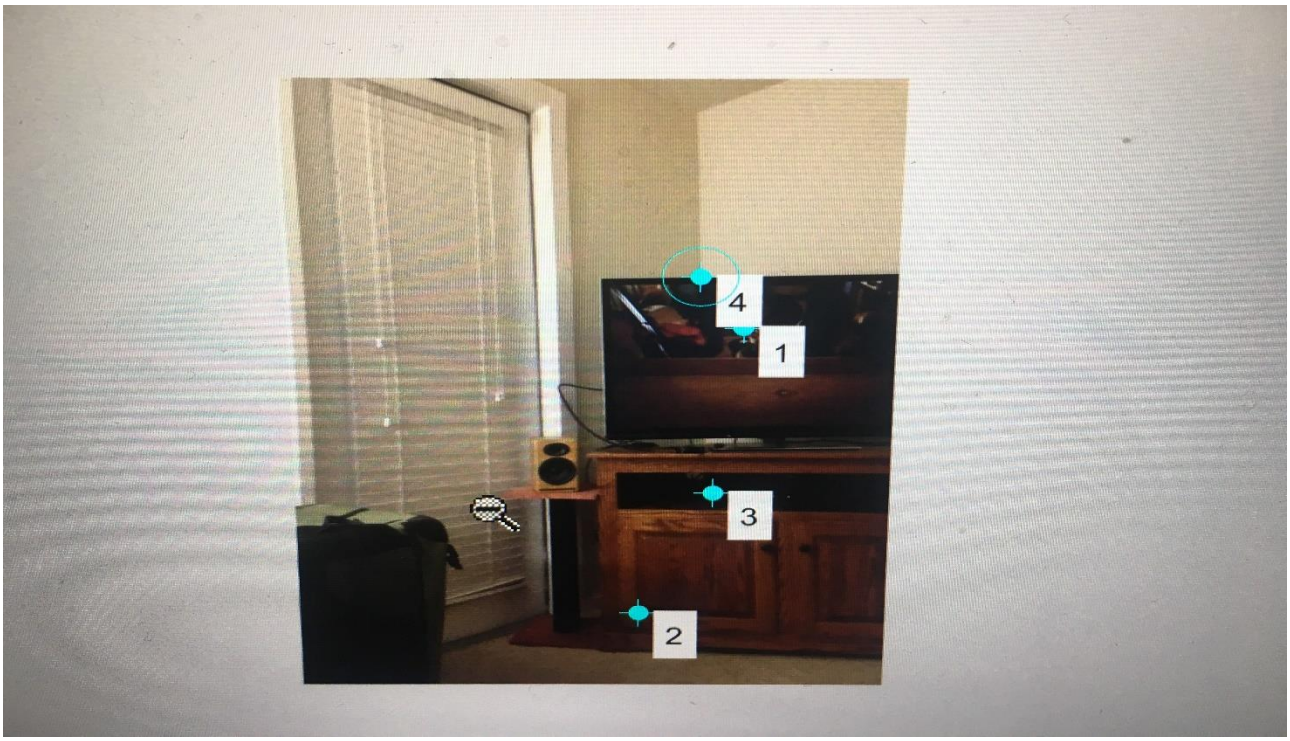
Left Image



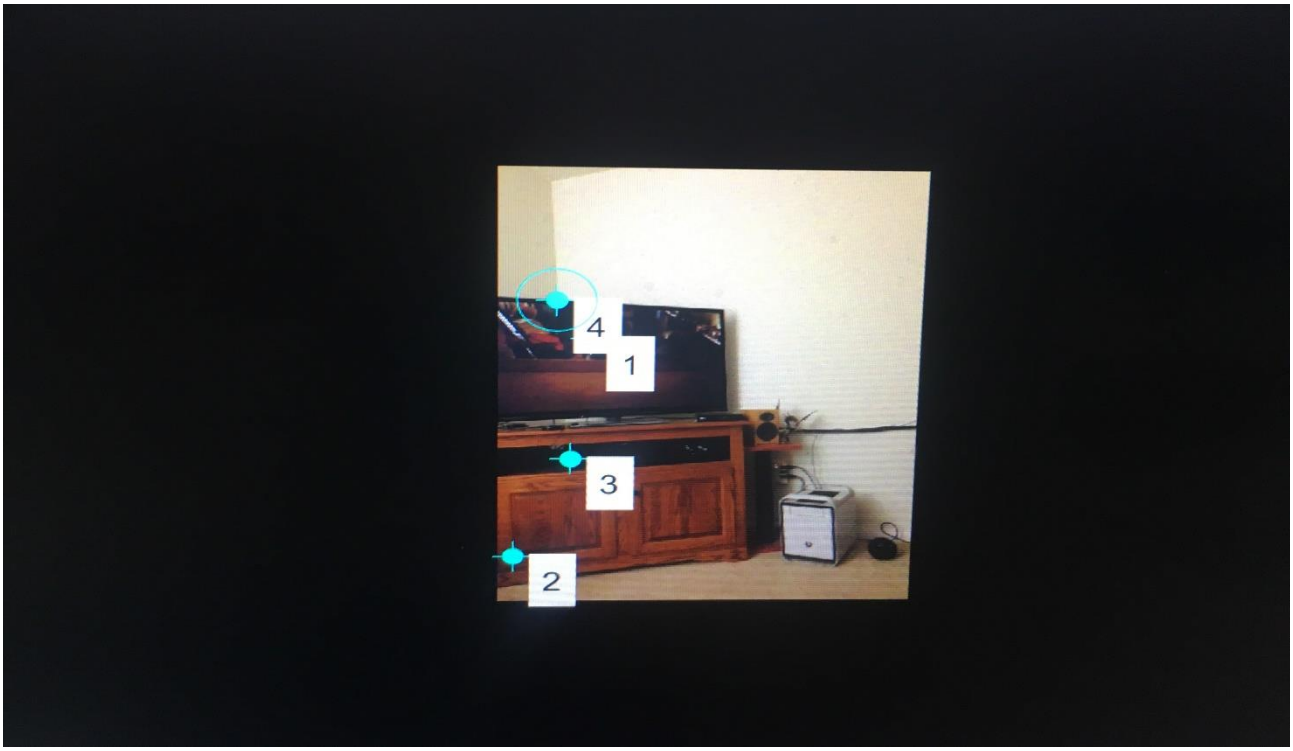
Right Image



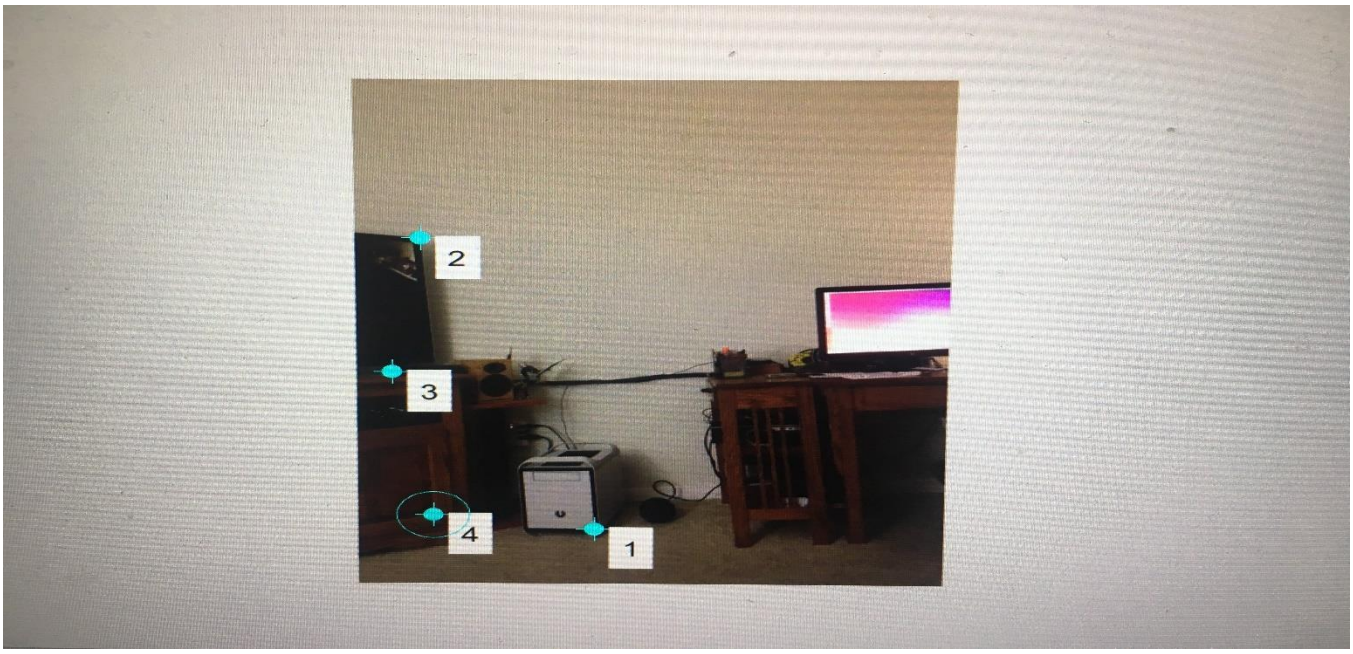
The above image is the middle image pasted on desired location on the bigger image. From comparing them we can select the control points.



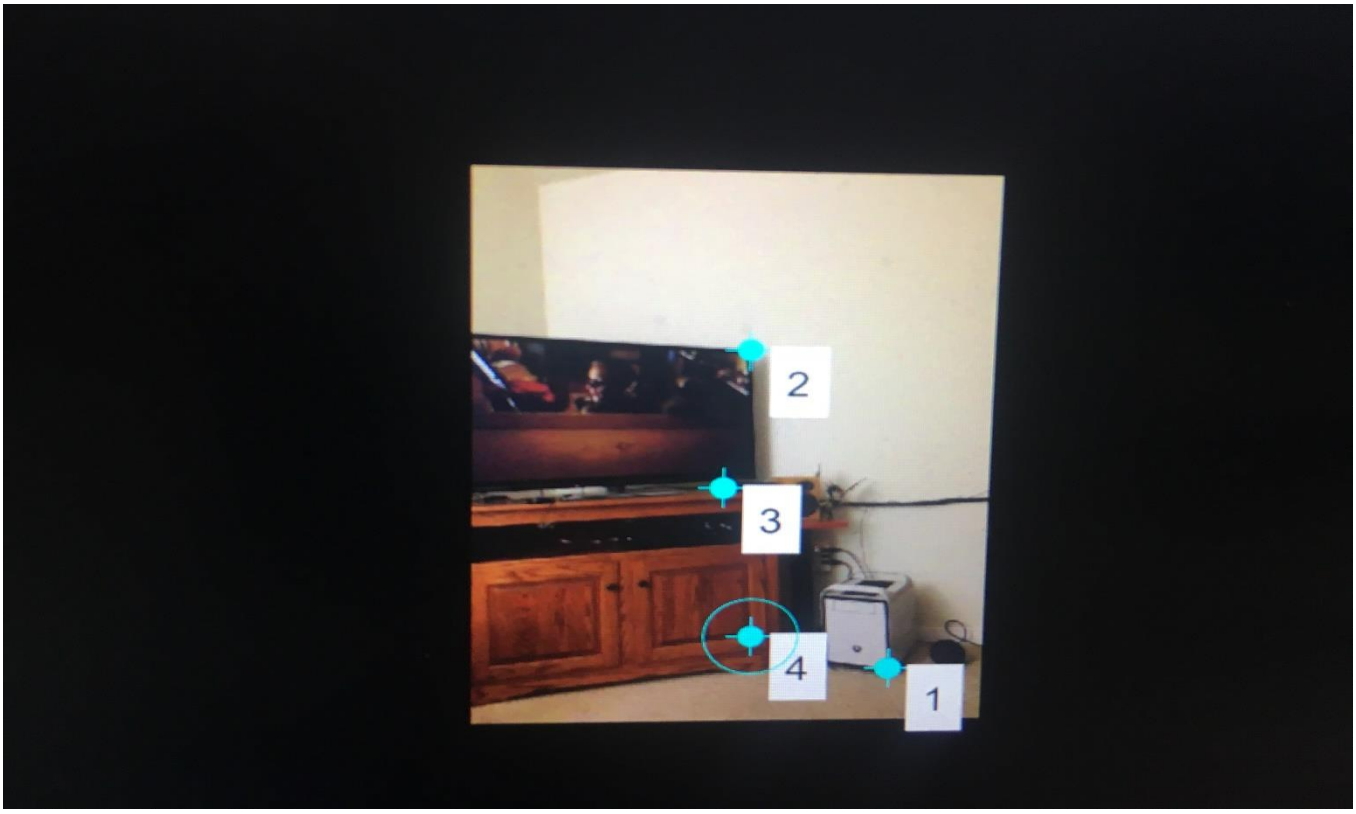
Control Points on left image



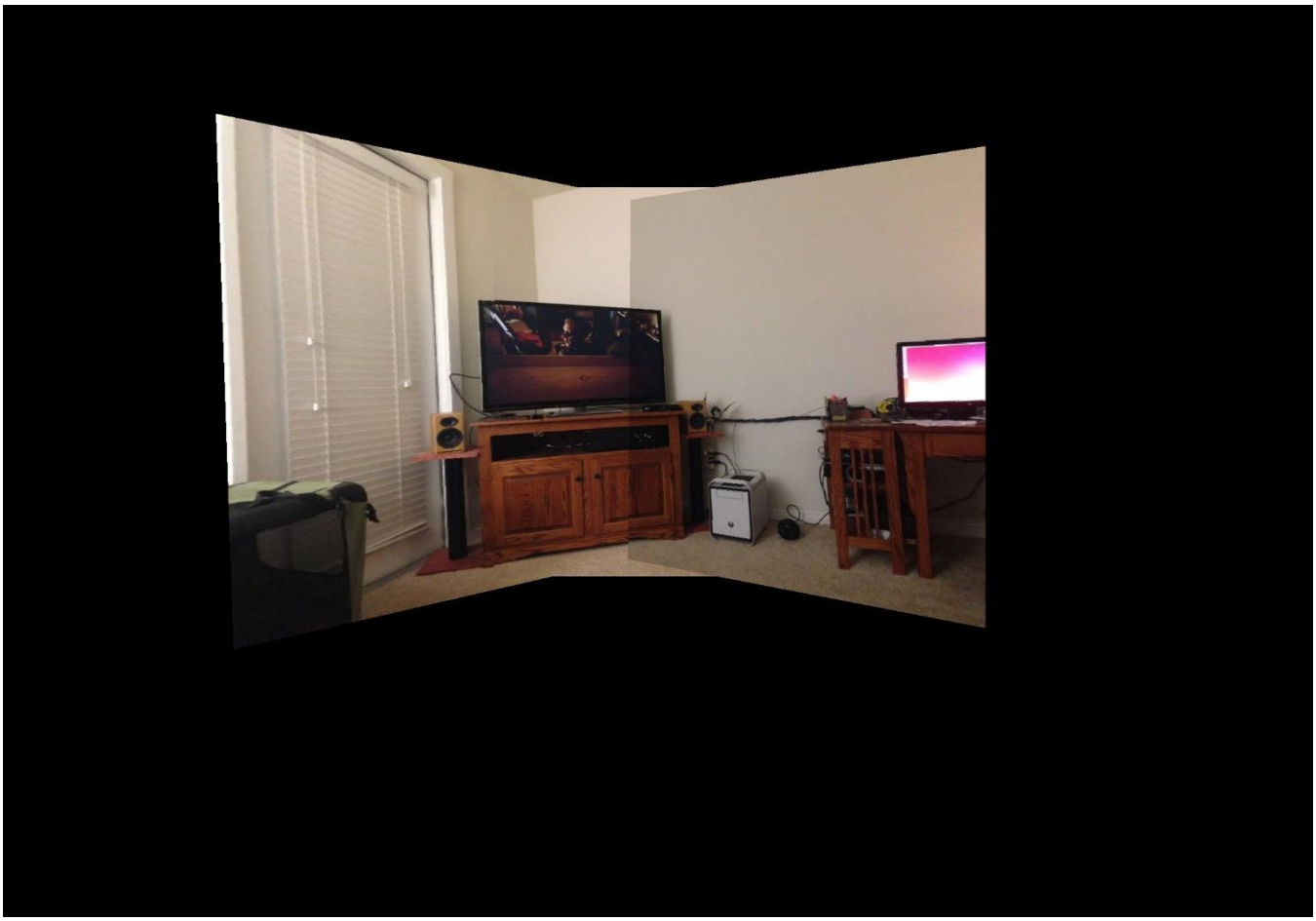
Corresponding Control Points on Middle Image



Control points on Right Image



Corresponding Control Points on Middle Image



This is the final Panorama Image created as a result.

Here we used four control points. If more were used, the placement and orientation of the left and right images will be better as the number of points to give information about the image in it's new position is better. Thus the panorama effect will be better implemented if more number of control points are chosen.

There's a command cpselect in MATLAB which helped in point selection. It opens a command window from which we can zoom into both the images and select the required control points from the images. Then there's an option to export these coordinates to the workplace in MATLAB. One of the strategies while selecting the points were to select those which show a large variation in intensity value from the neighboring pixels.

Problem 2.(a)

I. Abstract and Motivation

Half- toning is used in the print industry. Normally we have one type of ink – black to form dots on the paper. White color is represented by not printing a dot. When viewed from afar, the dots blend

together, to create the illusion of shapes and continuous lines. The density of dots present in a region decides the darkness of the image.

In this problem we are asked to convert input grayscale image to halftoned image using fixed thresholding, random thresholding and dithering matrix method.

II. Approach and Procedures

The fixed thresholding method is the easiest half toning method where we set a threshold of 127 to divide the 256 grayscale levels into two ranges. For each pixel in the image, we map it to 0 if it is smaller than the threshold, otherwise we map it to 255. But this results in monotones in the output image. The resulting image is very flat with minimum variation.

The random thresholding method is used to remove the monotones caused by the previous method. In this case, we use a random threshold by generating a random number in the range of 0-255 for each pixel. Then we compare the pixel intensity with the random threshold, and set it to 0 if it's less than the threshold and 255 if otherwise. But still the image is not as good as expected.

The Dithering Matrix method is an elegant method of halftoning. It uses multiple quantization thresholds for different pixels in a region to do quantization procedure. The quantization pattern used is also called as the Bayer Array and is described as:

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

Where,

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

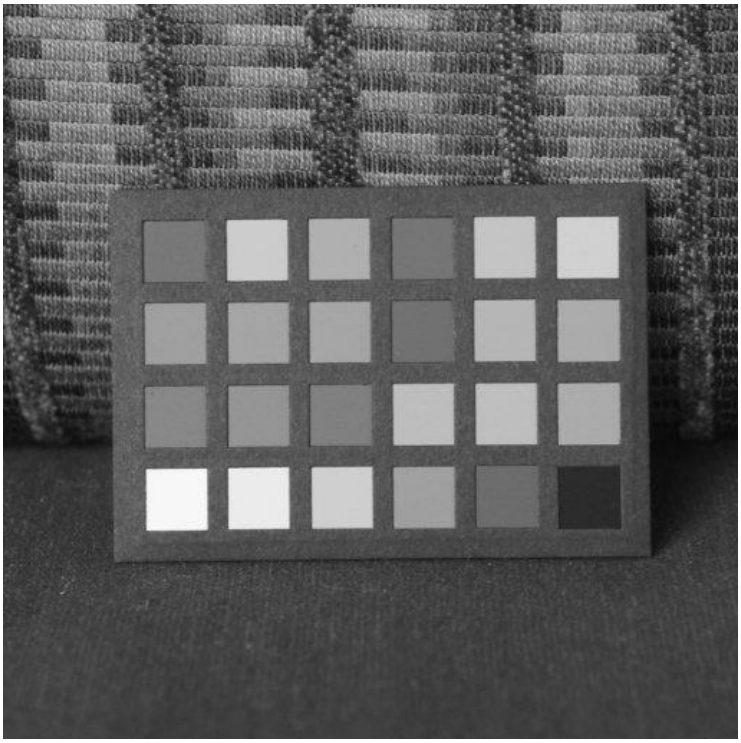
The index matrix can then be transformed into a threshold matrix T for an input gray-level image with
normalized pixel values (*i.e.* with its dynamic range between 0 and 1) by the following formula:
$T(x, y) = (I(x, y) + 0.5)/(\text{N*N})$
where (N*N) denotes the number of pixels in the matrix. Since the image is usually much larger than the threshold matrix, the matrix is repeated periodically across the full image. This is done by using the following formula:
$G(i, j) = 1$ if $F(i, j) > T(i\ mod N, j\ mod N)$; 0 $otherwise$
where *G(i, j)* is the normalized output binary image.

To apply the dithering approach when four gray levels were possible, I applied the following conditins after applying the dithering matrix:

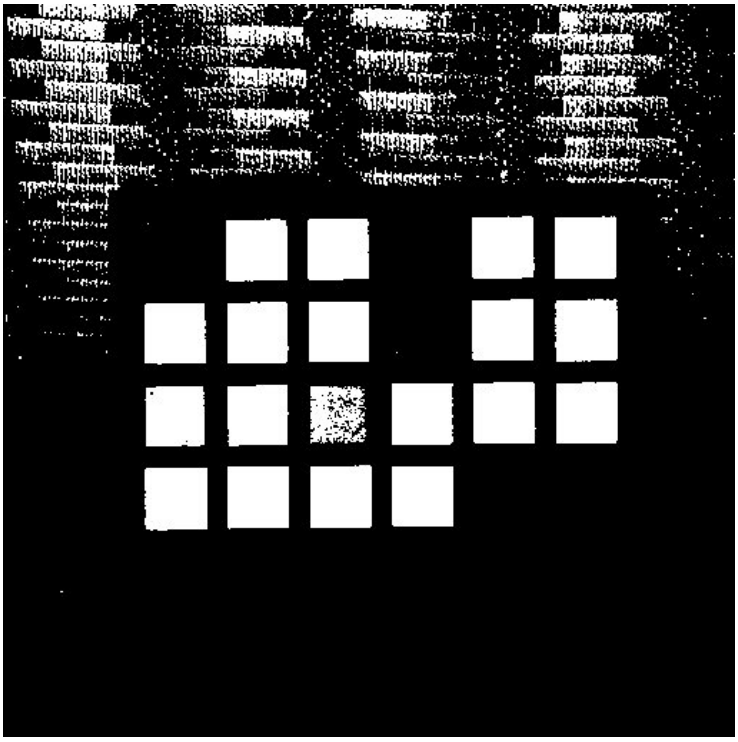
$$G(i,j) = \begin{cases} 0, & if\ 0 \le F(i,j) \le \dfrac{T(i\%N, j\%N)}{2} \\ 85, & if\ \dfrac{T(i\%N, j\%N)}{2} < F(i,j) \le T(i\%N, j\%N) \\ 170, & if\ T(i\%N, j\%N) < F(i,j) \le 127 + \dfrac{T(i\%N, j\%N)}{2} \\ 255, & if\ 127 + \dfrac{T(i\%N, j\%N)}{2} < F(i,j) \le 255 \end{cases}$$

I used C++ to write all my codes.

III. Result and Discussion

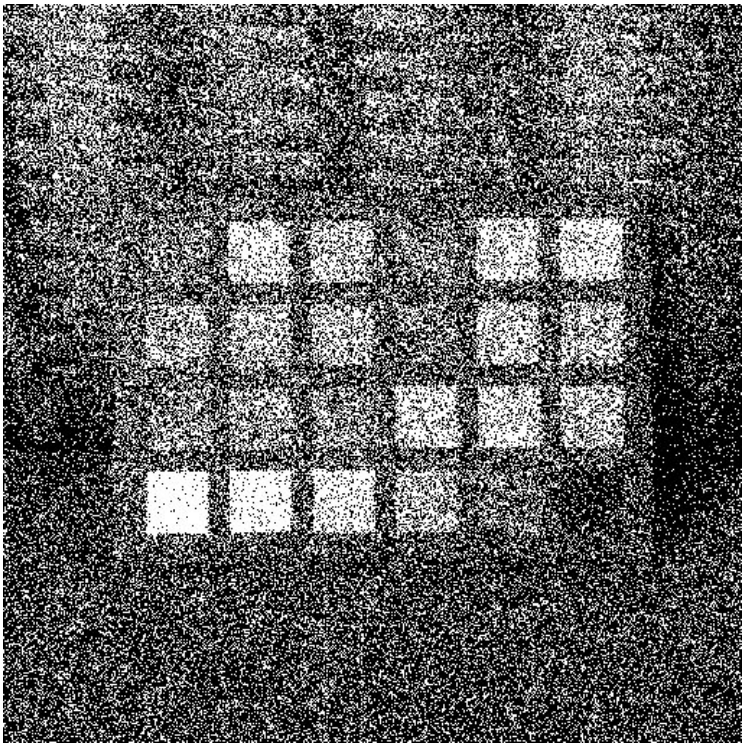


Original Image



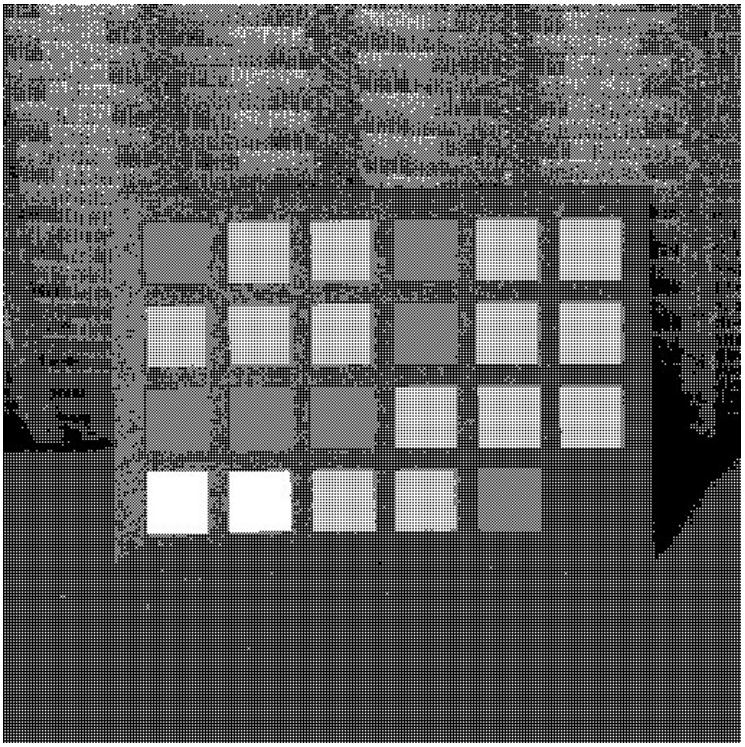
Fixed Thresholding Half toning.

As we can see, in this technique, the output looks no where close to the input and there are monotones in the image.



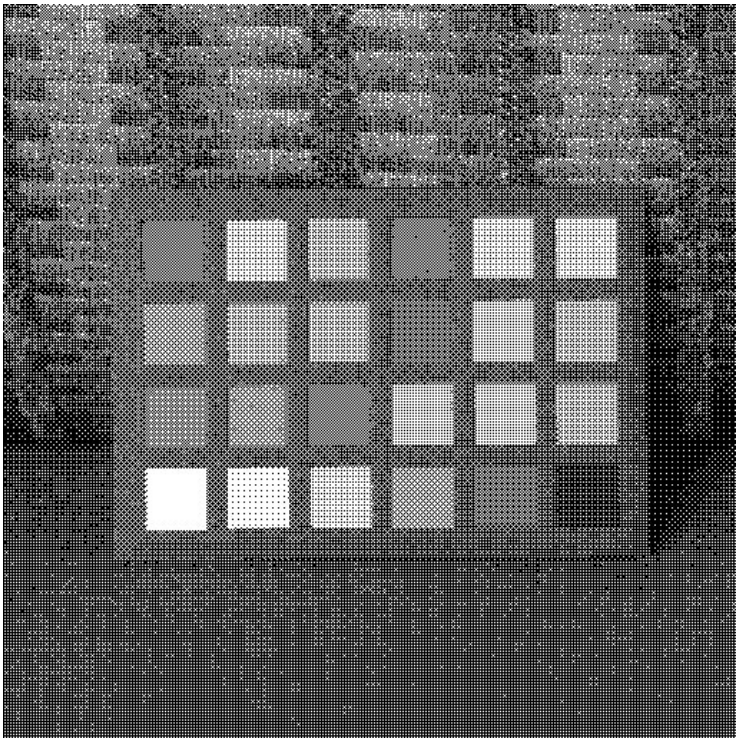
Random Thresholding Half Toning

As we can see from the output, it's better than fixed thresholding but the dots are too random and the image is not clearly seen.



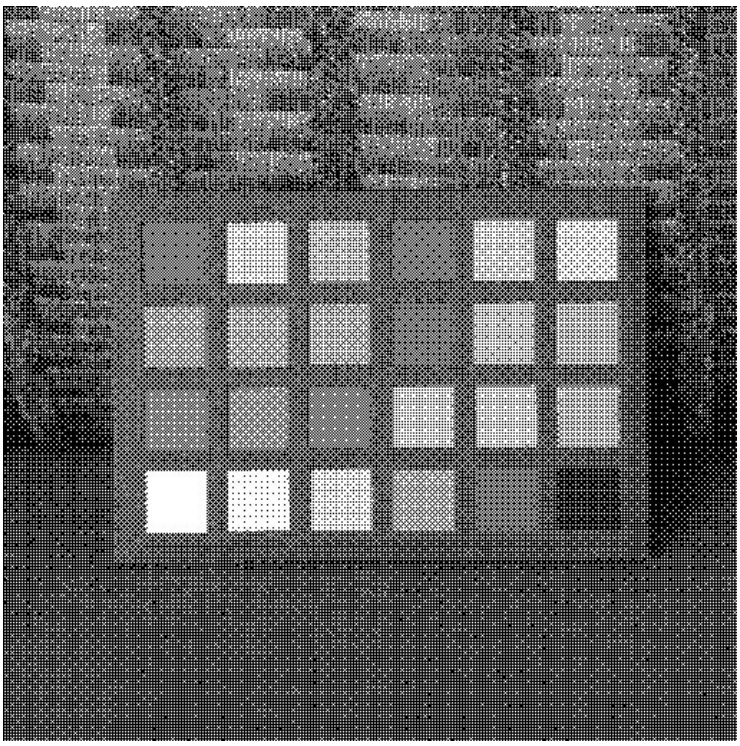
Dithering Halftoning technique with Index matrix of size 2

Dithering gives a much better result than the previous two methods but as the order of Bayer matrix is least, the density of dots is least due to which even from distance we can still see some of the dots in the image.



Dithering Halftoning technique with Index matrix of size 4

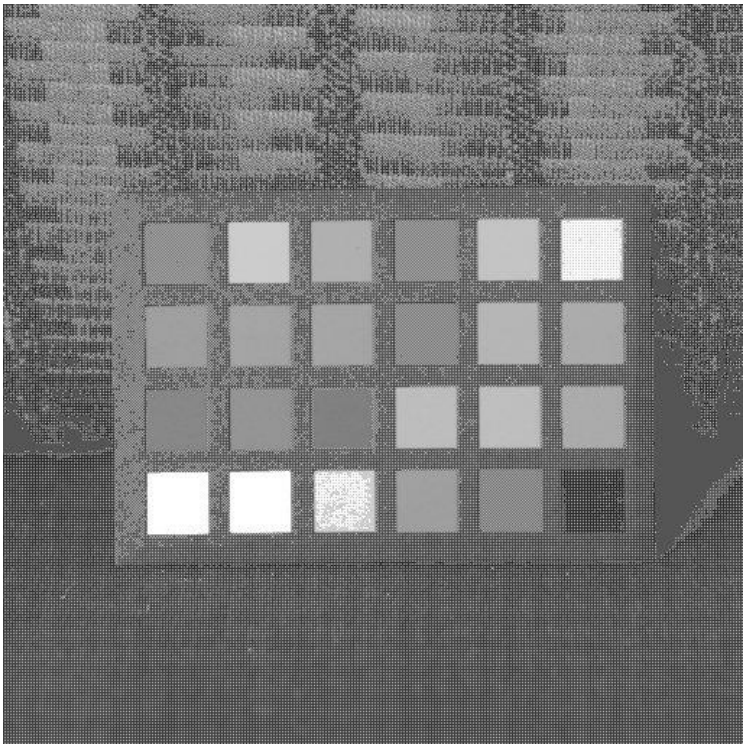
As the order of Bayer matrix is more than the previous case, the density of dots is more which leads to better image but still from distance we can see some of the dots in the image.



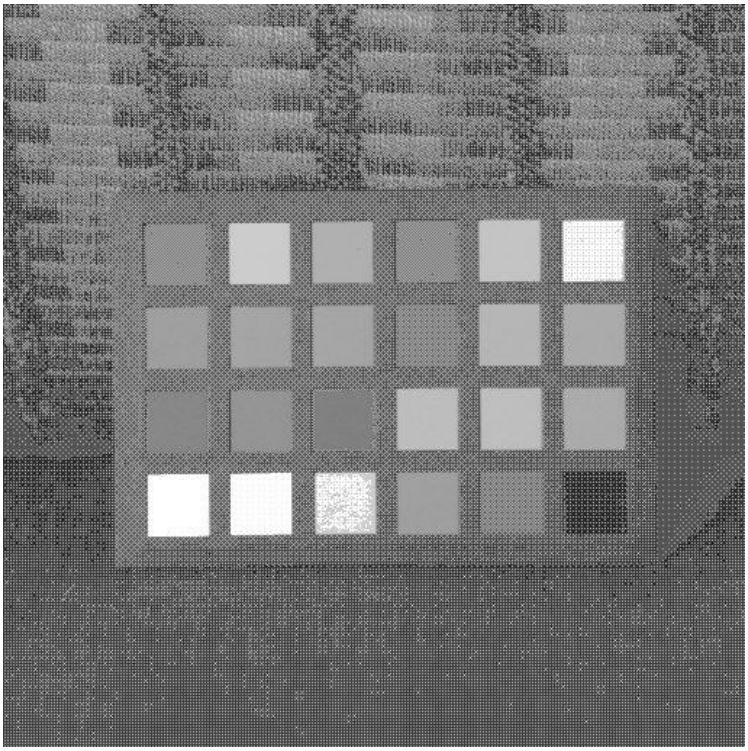
Dithering Halftoning technique with Index matrix of size 8

As the order of Bayer matrix is more than the previous case, the density of dots is most which leads to the closest to the original image.

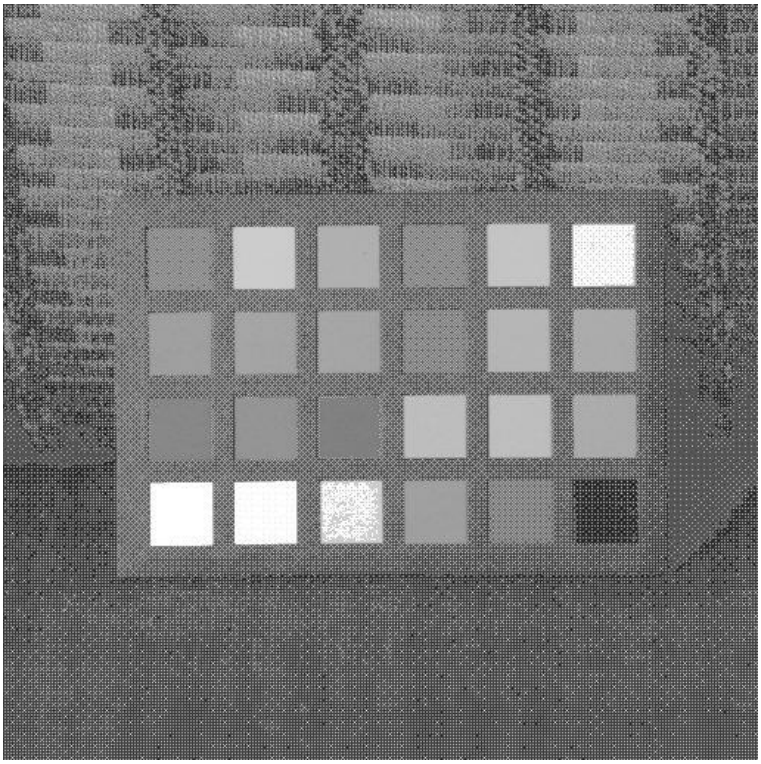
Thus we can infer that as the order of Bayer matrix increases, the density of dots increases and we get output closest to the input grayscale image.



Dithering Halftoning technique with Index matrix of size 2 and four gray levels.



Dithering Halftoning technique with Index matrix of size 4 and four gray levels.



Dithering Halftoning technique with Index matrix of size 8 and four gray levels.

With the introduction of additional gray levels in this technique, the details of the image are much more clearer and visible than the previous counterparts. Also, as the order of Bayer matrix increases, the density of dots increases and we get output closest to the input grayscale image.

Problem 2.(b)

I. Abstract and Motivation

We are expected to perform Error Diffusion Halftoning method. Here in this method the residual obtained from the quantization of one pixel is then distributed to the neighboring pixels that have yet to be processed. There are three major error diffusion matrices which we had to use: Floyd-Steinberg, JJN and Stucki matrix.

II. Approach and Procedures

I used C++ to write the code for the question. Going through the image from left to right, for each pixel, the error is calculated when quantizing, and then this error is diffused in the neighboring pixels in the various diffusion matrix:

Error Diffusion

Distribute the error to the neighboring pixels

P1	P2
P3	P4

B1=P1>128 1 else 0

E1=B1-P1;

Floyd-Steinberg

	(i,j)	7
3	5	1

(a) Floyd-Steinberg

		(i,j)	7	5
3	5	7	5	3
1	3	5	3	1

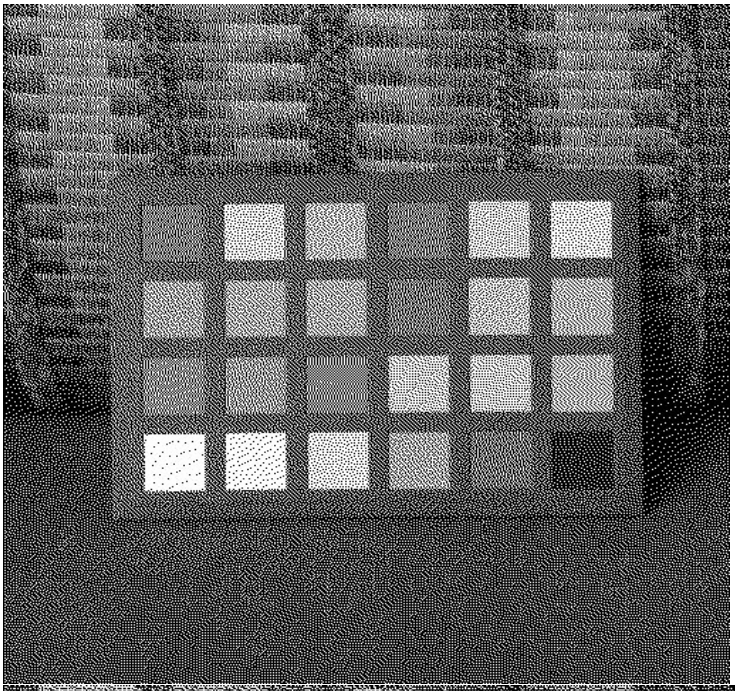
(b) Jarvis

		(i,j)	8	4
2	4	8	4	2
1	2	4	2	1

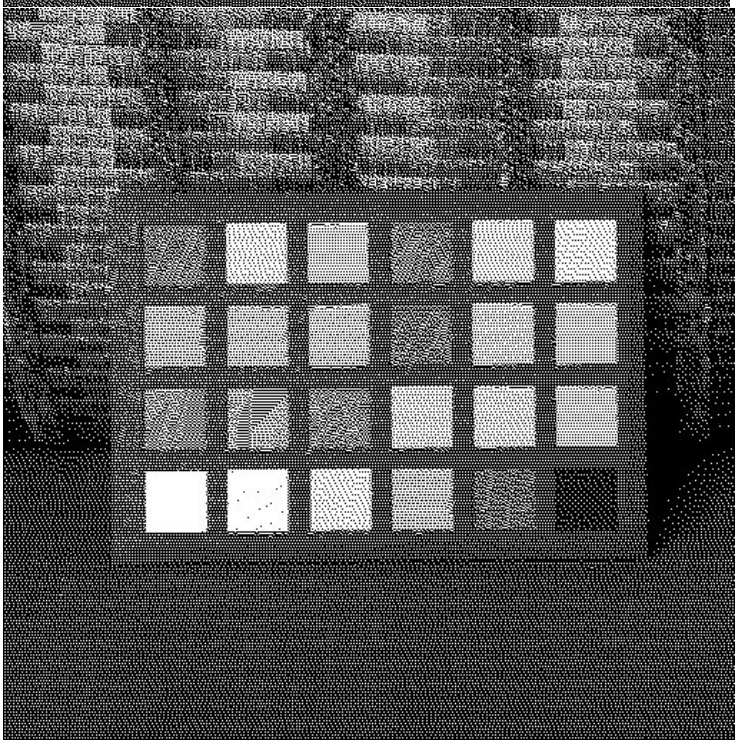
(c) Stucki

For the next pixel, thus pixel value plus error goes for the quantization process. This is done repeatedly for the whole image.

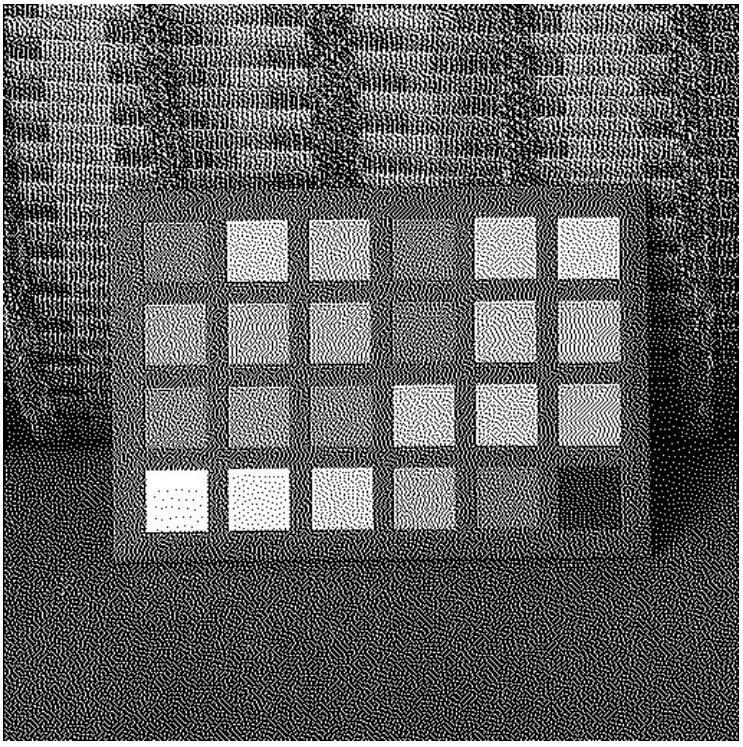
III. Result and Discussion



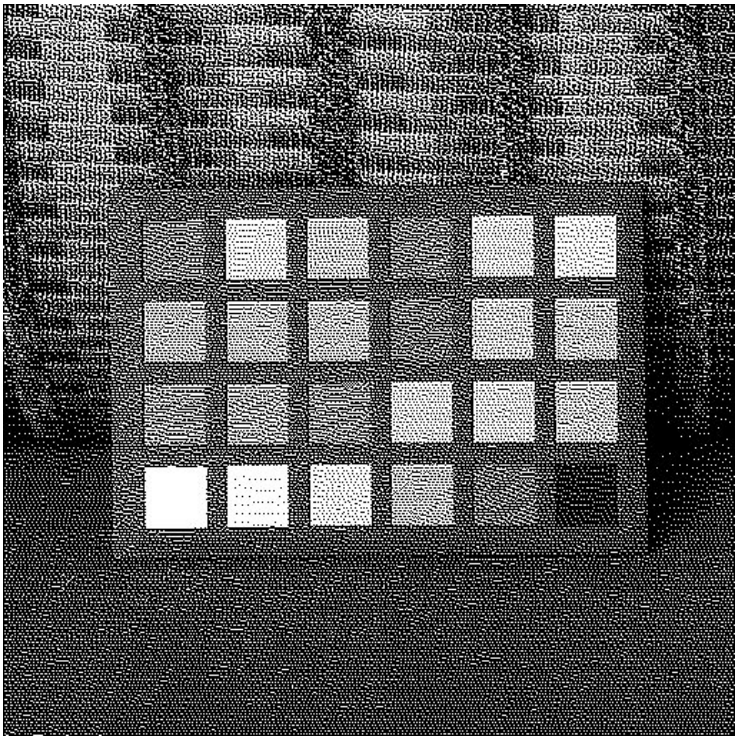
Error Diffusion by Floyd-Steinberg's error diffusion matrix



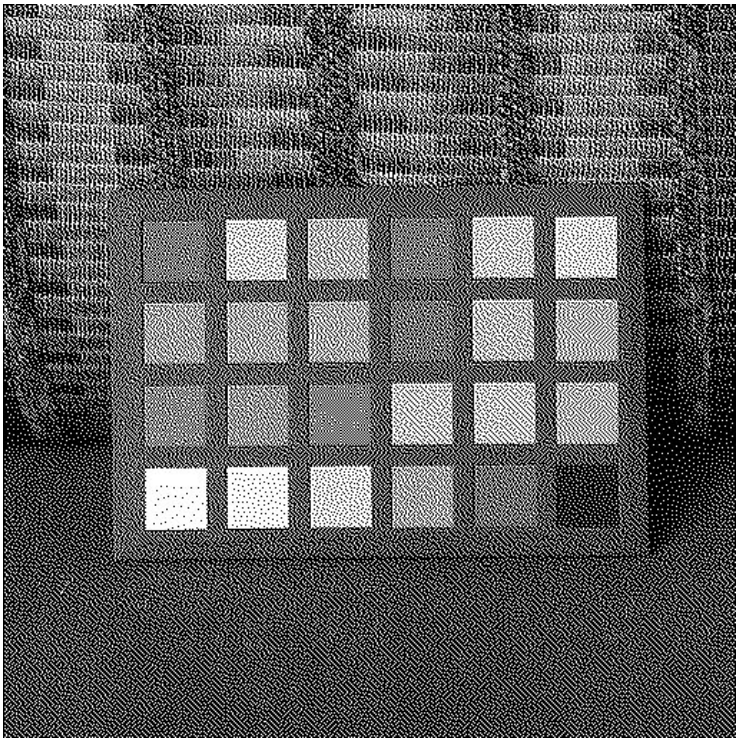
Error Diffusion by Floyd-Steinberg's error diffusion matrix in serpentine method.



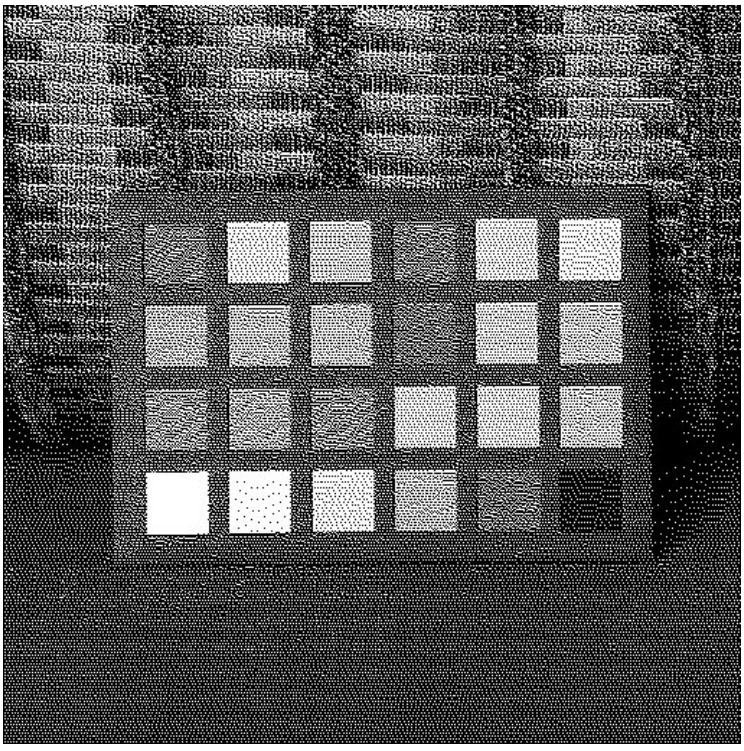
Error Diffusion by JJN error diffusion matrix



Error Diffusion by JJN error diffusion matrix in serpentine method



Error Diffusion by Stucki error diffusion matrix



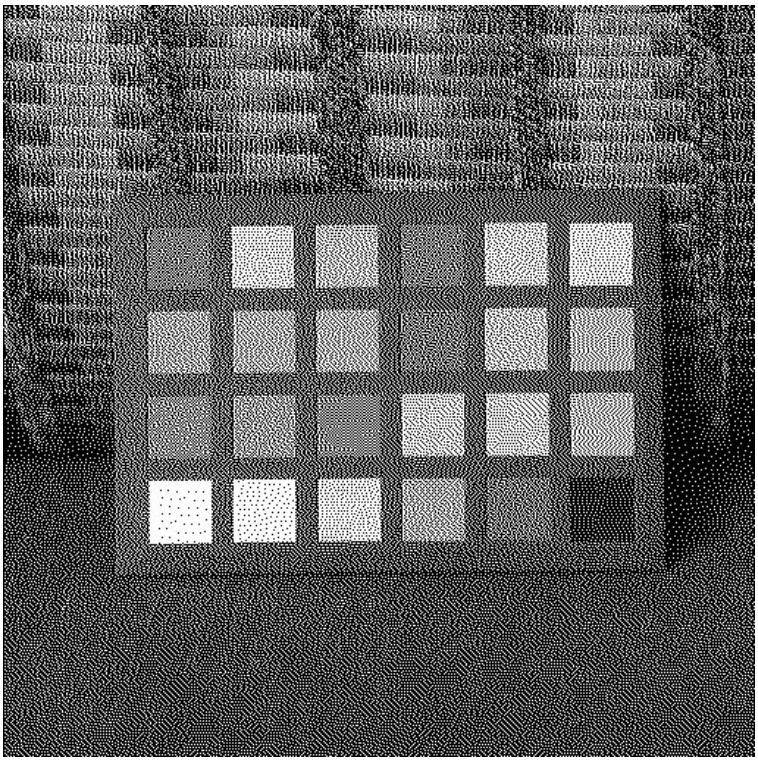
Error Diffusion by Stucki error diffusion matrix by serpentine method

I prefer the error diffusion method better than dithering method. This is because if there is text present in the image, error diffusion method is the best method to enhance the edges. The

output image gives the closest resemblance to the input by the error diffusion method. The finer details of the image are best preserved by the Stucki error diffusion method.
Daniel Burkes suggested a further improvement to the Stucki error diffusion matrix by removing the bottom row of the matrix. This removed the need for two forward arrays and resulted in a divisor which was a multiple of 2. This ensured that simple bit shifting can accomplish all error calculations. It works in similar matter to all other error diffusion matrix by adding residual quantization error of pixel to neighboring pixels. In this case on average, the quantization error is closer to 0. The mask is given below:

0	0	x	$\frac{8}{32}$	$\frac{4}{32}$
$\frac{2}{32}$	$\frac{4}{32}$	$\frac{8}{32}$	$\frac{4}{32}$	$\frac{2}{32}$

The result of applying Burkes Error diffusion matrix is that details in the Image are much more distinguishable now.



Error Diffusion by Burkes error diffusion matrix

Thus Burkes Error Diffusion Matrix gives the best result.

Problem 2.(c).1

I. Abstract and Motivation

We are expected to do color half toning in this problem. The easiest way to do this is to separate an image into RGB channels and then convert them to CMY channels. After this the Floyd-Steinberg error diffusion algorithm is applied to each channel to quantize them. This is the same process as used in problem 2.(b)

II. Approach and Procedures

I used C++ to write the code for the question. Going through the image from left to right, for each pixel, the error is calculated when quantizing, and then this error is diffused in the neighboring pixels using the Floyd-Steinberg diffusion matrix.

III. Result and Discussion

Original Image

Half Toned image using Floyd Steinberg Error Diffusion Matrix Algorithm

The half toned image is very granular in nature and has a lot of noise in it. Also, comparing the original and half toned image, we can say that there is a difference in the luminance of them both. The half toned image is a little duller than the original.

In this method, we apply the error diffusion matrix for each channel separately, which is a major shortcoming. For the final image, we integrate all the intensity values from the three channels to display the color. So even though we compensate the error in each channel and improve the picture quality, the result after the integration of the three channels can give a very large error. This is the main shortcoming of this approach.

Problem 2.(c).2

I. Abstract and Motivation

We are expected to do color half toning in this problem using the MBVQ based error diffusion. In this method, the CMY color space is partitioned into six Minimum Brightness Variation Quadrants(MBVQ) as shown:

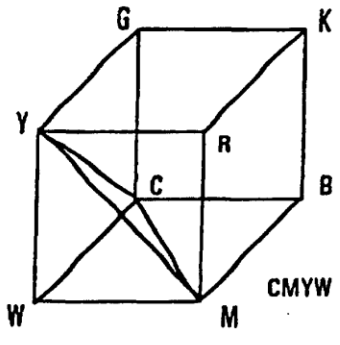


Figure 4A

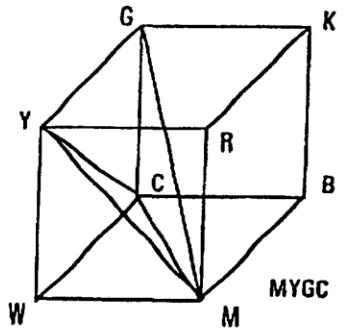


Figure 4B

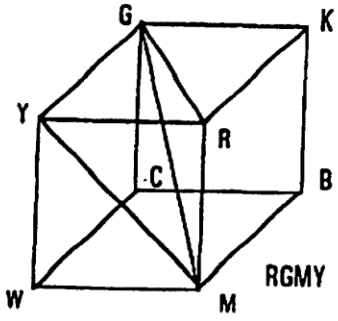


Figure 4C

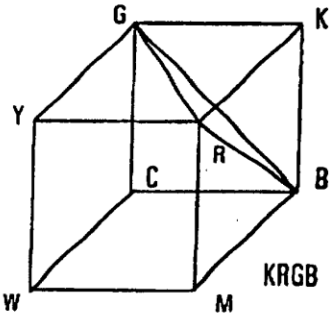


Figure 4D

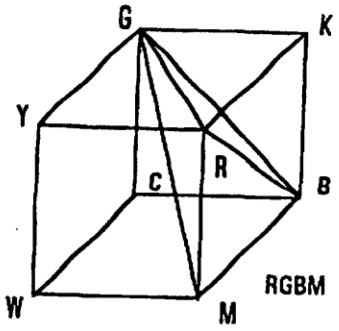


Figure 4E

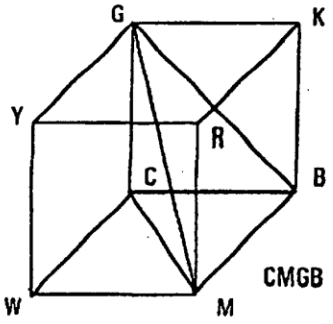
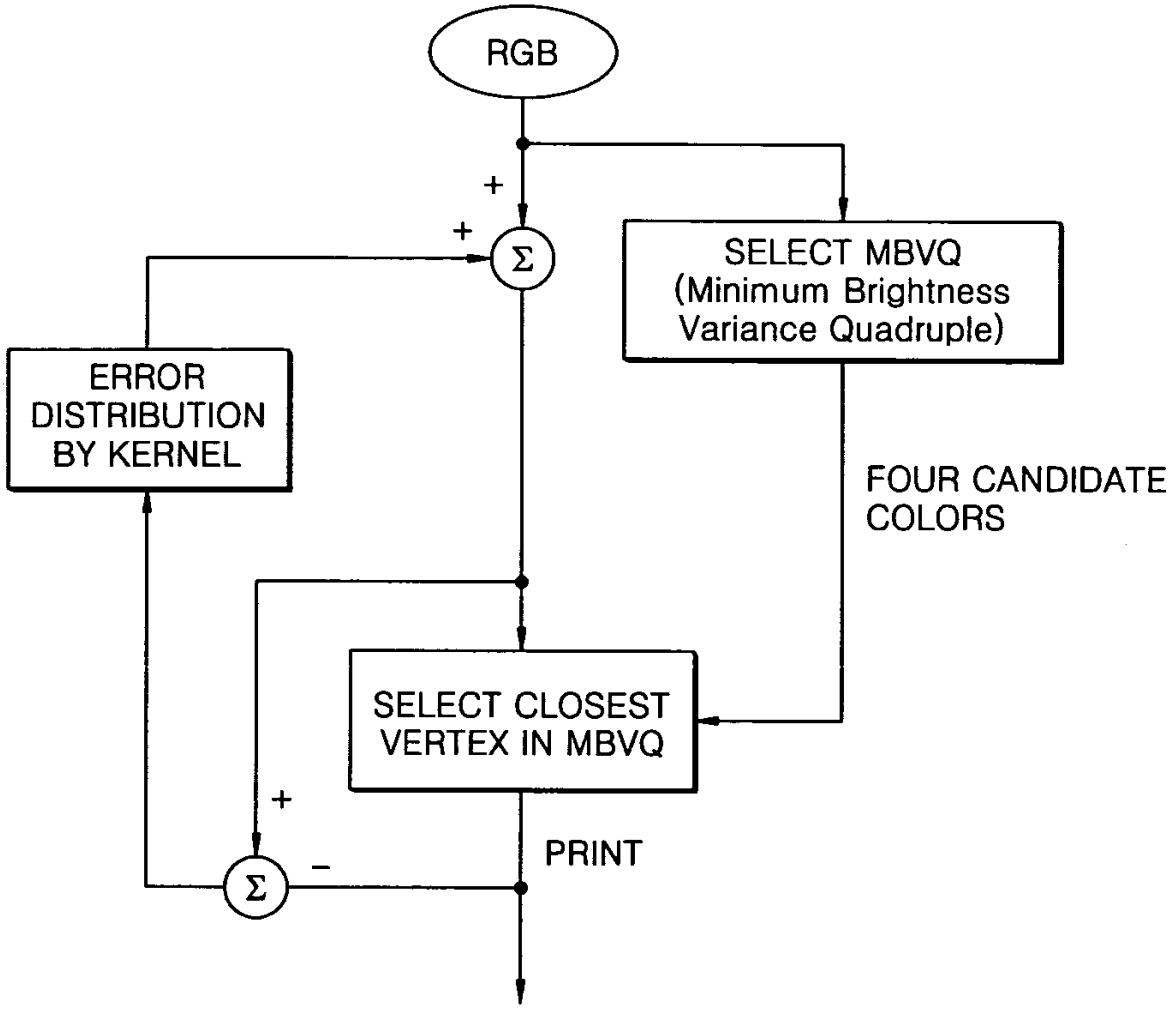


Figure 4F

II. Approach and Procedures

I used C++ to write the code for the question. The algorithm is as follows:

FIG. 1 (PRIOR ART)



```
inline pyramid position (Byte R, Byte G, Byte B)
{
        if ((R+G) & 256)
                if ((G+B) & 256)
                        if ((R+G+B) & 512)          return CMYW;
                        else                        return MYGC;
                else                                return RGMY;
        else
                if(!((G+B) & 256))
                        if(!((R+G+B) & 256))        return KRGB;
                        else                        return RGBM;
                else                                return CMGB;
}
```

The pixels of the input image is scanned in serpentine order. For every pixel, I saw it's location in the cube and found out which MBVQ it belonged to using it's RGB values. Using the vertex of the MBVQ tetrahedron closest to the pixel, we find it's value plus the error. Then this vertex value got is subtracted from the pixel's original CMY value and the error. This gives the new quantization error. Then this quantized error is distributed to the future pixel error buckets using the standard FS error diffusion matrix algorithm.

III. Result and Discussion

Original Image

Half-toned image using MBVQ method

Compared to method discussed in Problem 2.c.2, the color of the details in the halftoned image is better, though the image is still granular in nature. The noise is less than in previous method and the luminance is better, resulting in a better image.

In this case rather than applying error diffusion to each channel separately, we do the error diffusion at the same time. Thus a more accurate smaller half tone set is chosen for the error diffusion step, reducing the error between the real color and quantized color of the image. Thus, noise is reduced in this method compared to the previous one.

Problem 3.(a)

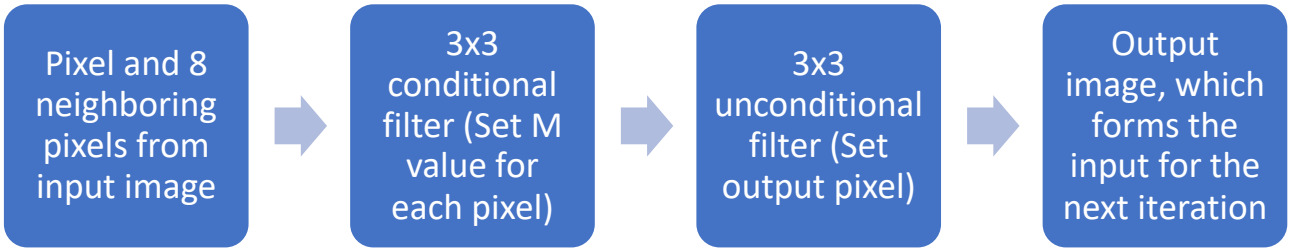
I. Abstract and Motivation

In this problem, we were asked to apply the shrinking filter to the stars image. Shrinking is a morphological operation. Morphological processing techniques are used to remove imperfections in binary images by using non linear operations to work on the form and structure of the image.

II. Approach and Procedures

I used C++ to write the code for the question. The algorithm is as follows:

First, the grayscale image is converted to binary by thresholding the image and setting all values below 127 as 0 and above as 255. The code uses two sets of filters to match and shrink the image : conditional and unconditional filters. Using these predefined filters, each pixel of the image is scanned along with it's eight neighboring pixels and checked whether they match the filters or not(hit or miss filters). Based on match, we take decision on the pixel, i.e., if match, we change the pixel value according to problem statement and otherwise, we copy the original pixel value for the output image.



We use 3x3 filters despite there being a possibility of over erasure leading to loss of connectivity in object in image. 5x5 filters would give better result but the total number of filters in that case would be much larger. To accommodate for this we use cascaded two 3x3 filters.

When the input image pixel goes through the initial 3x3 conditional filter, it checks whether to erase or not. If there's a hit(match) between the input and conditional filter, we set M=1 for the pixel which indicates that the pixel value may be changed from 1 to 0. If it doesn't match, we set M=0, meaning the pixel value remains as it is. This is done for the entire input image and a temporary weight_m image is created to store the corresponding M values. This new temporary image is passed through the 3x3 unconditional masks to check whether we need to erase the pixel value or not. If it matches the

unconditional filter, we set value of new output image as 1, i.e., retain pixel value. Otherwise we set it to 0 and the pixel is erased. This output image is the input for the next iteration. This is done repeatedly till we find that there occurs no change during an iteration.

$$x_i \begin{cases} x_i = 0 \rightarrow pixel\ value\ remained \\ x_i = 1 \begin{cases} miss\ filter \rightarrow pixel\ value\ remained \\ hit\ filter \rightarrow M = 1 \begin{cases} hit\ filter \rightarrow pixel\ value\ remained \\ miss\ filter \rightarrow pixel\ value\ changed \end{cases} \end{cases} \end{cases}$$

The masks are as follows:

Type	Bond	Patterns
S	1	001 100 000 000 010 010 010 010 000 000 100 001
S	2	000 010 000 000 011 010 110 010 000 000 000 010
S	3	001 011 110 100 000 000 000 000 011 010 010 110 110 010 010 011 000 000 000 000 100 110 011 001
TK	4	010 010 000 000 011 110 110 011 000 000 010 010
STK	4	001 111 100 000 011 010 110 010 001 000 100 111
ST	5	110 010 011 001 011 011 110 011 000 001 000 010
ST	5	011 110 000 000 011 110 110 011 000 000 110 011
ST	6	110 011 011 110 001 100
STK	6	111 011 111 110 100 000 000 001 011 011 110 110 110 110 011 011 000 001 000 100 110 111 111 011
STK	7	111 111 100 001 011 110 110 011 001 100 111 111
STK	8	011 111 110 000 011 111 110 111 011 000 110 111
STK	9	111 011 111 111 111 110 100 001 011 011 111 111 110 110 111 111 011 111 100 001 110 111 111 111
STK	10	111 111 111 101 011 111 110 111 111 101 111 111
K	11	111 111 110 011 111 111 111 111 011 110 111 111

Conditional masks for Shrinking, Thinning and Skeletonizing

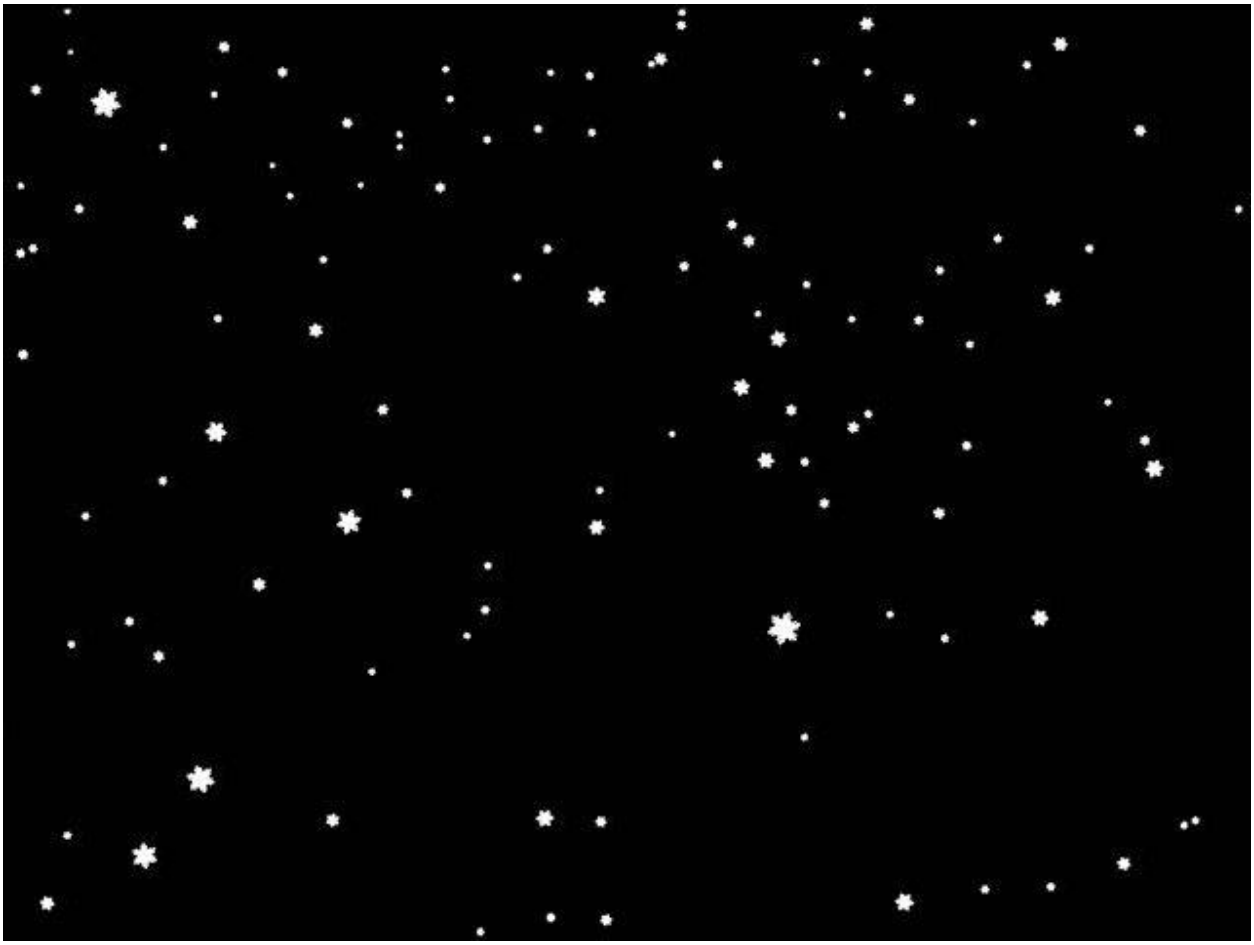
Spur	0	0	M	M	0	0						
	0	M	0	0	M	0						
	0	0			0	0						
Single 4-connection	0	0	0	0	0	0						
	0	M	0	0	M	M						
	0	M	0	0	0	0						
L Cluster	0	0	M	0	M	M	M	M	0	M	0	0
	0	M	M	0	M	0	0	M	0	M	M	0
	0	0	0	0	0	0	0	0	0	0	0	0
	M	M	0	0	M	0	0	M	0	0	M	M
	M	0	0	M	M	0	0	M	M	0	0	M
4-connected Offset	0	M	M	M	M	0	0	M	0	0	0	M
	M	M	0	0	M	M	0	M	M	0	M	M
	0	0	0	0	0	M	0	0	M	0	M	0
						0						
Spur corner Cluster	0	A	M	M	B	0	0	0	M	M	0	0
	0	M	B	A	M	0	A	M	0	0	M	B
	M	0	0	0	0	M	M	B			A	M
Corner Cluster	M	M	D									
	M	M	D									
	D	D	D									
Tee Branch	D	M	0	0	M	D	0	0	D	D	0	0
	M	M			M	M			M	M		
	D	0	M	M	0	D	M	M	D	D	M	M
			0	0			0	M			M	0
	D	M	D	0	M	0	0	M	0	D	M	D
	M	M	0		M			M		0	M	M
	0	M	0	M	M	0	0		M	0	M	0
				D		D	D	M	D			
Vee Branch	M	D	M	M	D	C	C	B	A	A	D	M
	D	M	D	D	M	B	D	M	D	B	M	D
	A	B	C	M	D	A	M	D	M	C	D	M
Diagonal Branch	D	M	0	0	M	D	D	0	M	M	0	D
	0	M			M	0	M		0	0		M
	M	0	M	M	0	M	0	M	D	D	M	0
			D	D				M			M	
A or B or C = 1 D = 0 or 1 A or B = 1												

Unconditional Masks for Shrinking and Thinning

To count the number of stars in the image, after the iterations are completed, just count the total number of white pixels left in the image. This will give the total number of stars in the image. This is because the shrinking operation reduces each star to a single pixel.

Also, to show the different star sizes and their frequencies, I used the BFS algorithm. Breadth First Search (BFS) is a way of going through the nodes in a tree one by one. In this method, we define a double ended queue. Whenever we hit a white pixel, we enter it's coordinates into the queue and check for the neighboring four pixels at up, down, left and right. If any of these also contain white, then their coordinates are also entered in the queue. Then the original pixel is popped out and we do the same algorithm for the next value in the queue. Whenever we push a coordinate into the queue, we turn that pixel black to shrink the star. Only the original pixel is kept white. Using a counter to count the total number of pixels entered into the queue gives the size of the star in terms of pixels contained in it.

III. Result and Discussion



Original Image

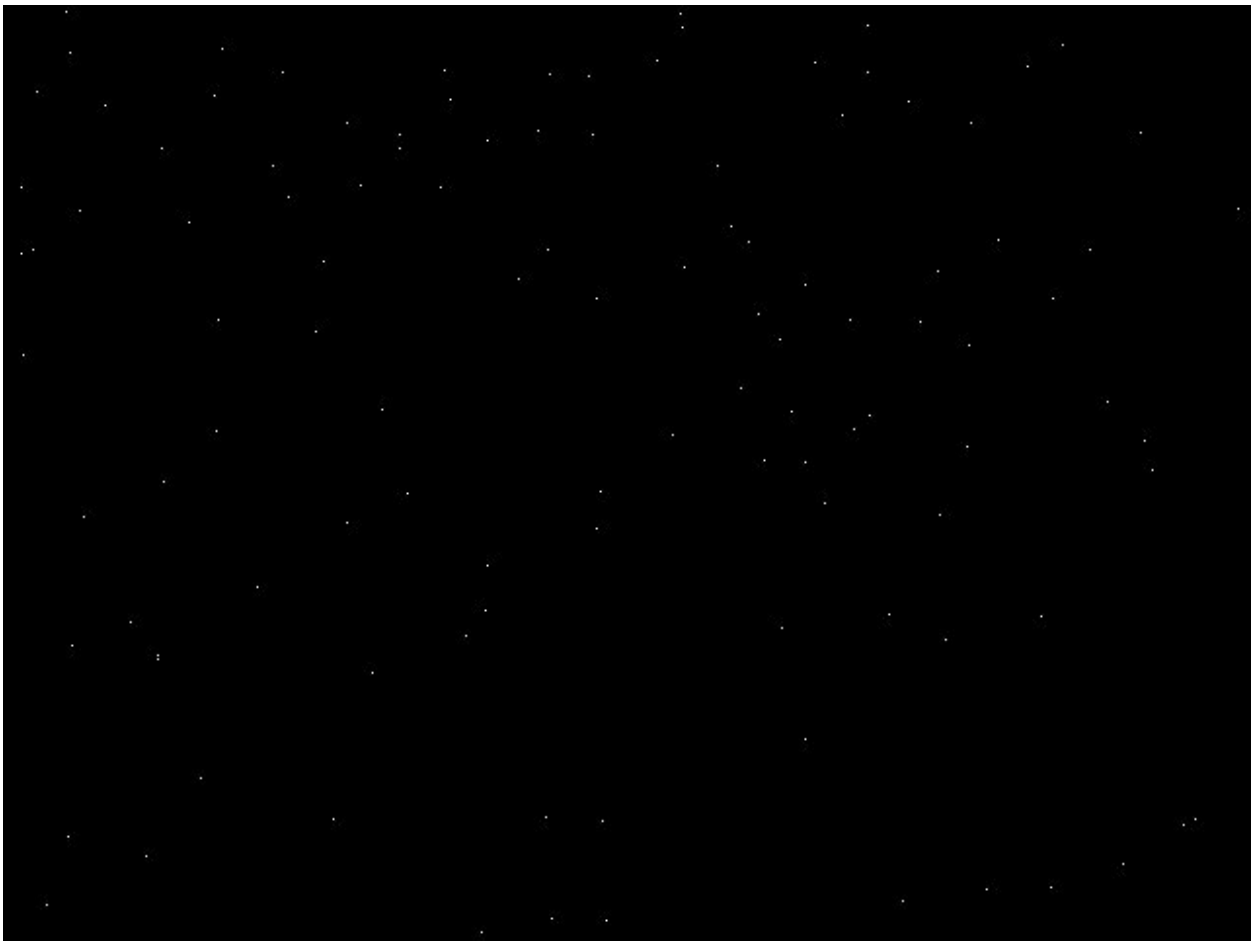
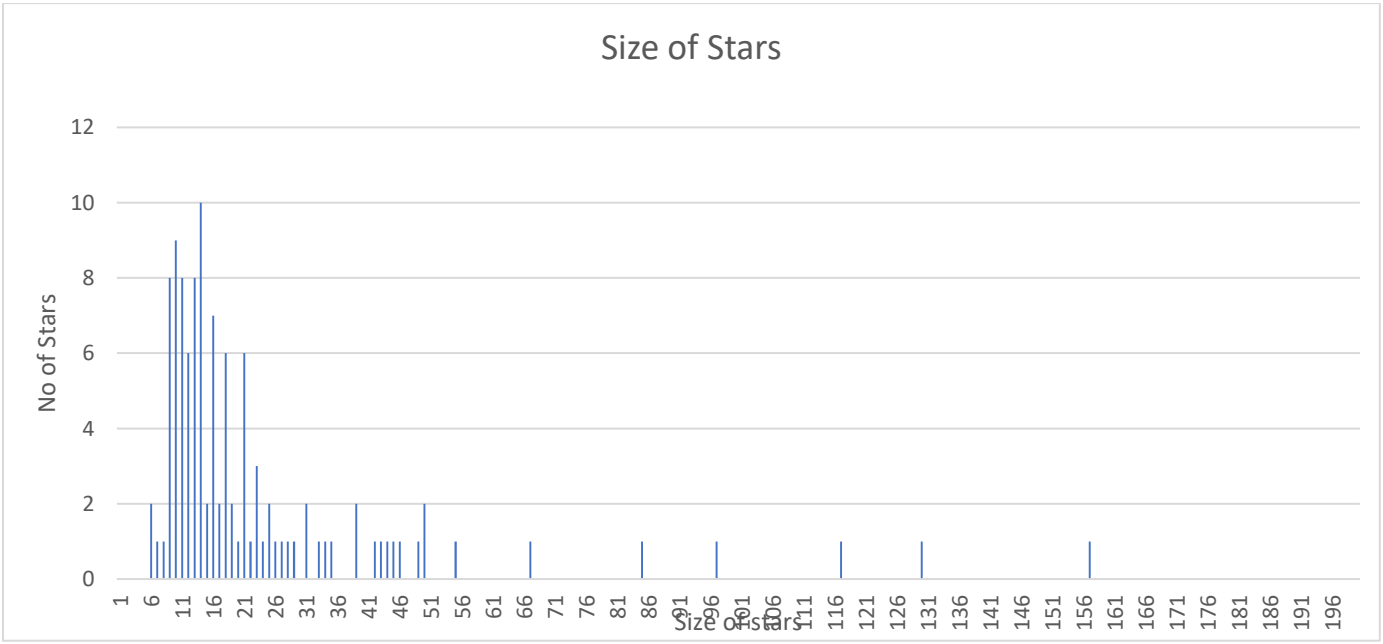


Image after shrinking operation

Total number of stars : 112



Problem 3.(b)

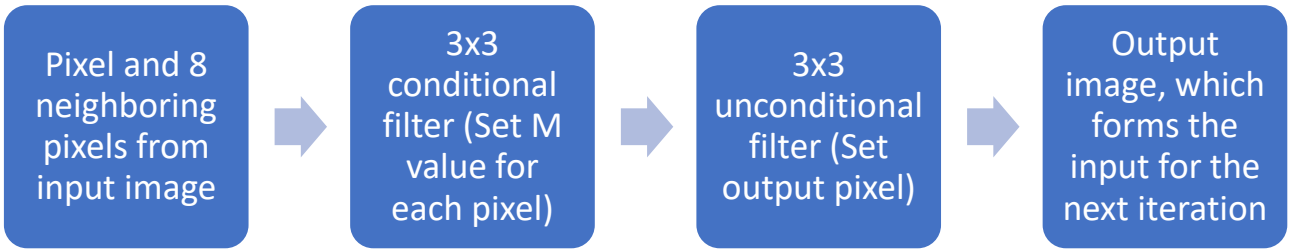
I. Abstract and Motivation

In this problem, we were asked to apply the thinning filter to the jigsaw_1 image. Thinning is a morphological operation. Morphological processing techniques are used to remove imperfections in binary images by using non linear operations to work on the form and structure of the image.

II. Approach and Procedures

I used C++ to write the code for the question. The algorithm is as follows:

First, the grayscale image is converted to binary by thresholding the image and setting all values below 127 as 0 and above as 255. The code uses two sets of filters to match and shrink the image : conditional and unconditional filters. Using these predefined filters, each pixel of the image is scanned along with it's eight neighboring pixels and checked whether they match the filters or not(hit or miss filters). Based on match, we take decision on the pixel, i.e., if match, we change the pixel value according to problem statement and otherwise, we copy the original pixel value for the output image.



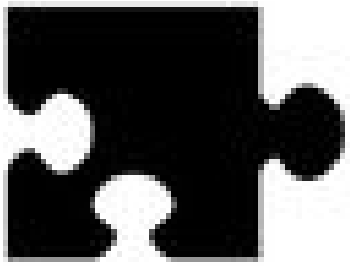
We use 3x3 filters despite there being a possibility of over erasure leading to loss of connectivity in object in image. 5x5 filters would give better result but the total number of filters in that case would be much larger. To accommodate for this we use cascaded two 3x3 filters.

When the input image pixel goes through the initial 3x3 conditional filter, it checks whether to erase or not. If there's a hit(match) between the input and conditional filter, we set M=1 for the pixel which indicates that the pixel value may be changed from 1 to 0. If it doesn't match, we set M=0, meaning the pixel value remains as it is. This is done for the entire input image and a temporary weight_m image is created to store the corresponding M values. This new temporary image is passed through the 3x3 unconditional masks to check whether we need to erase the pixel value or not. If it matches the unconditional filter, we set value of new output image as 1, i.e., retain pixel value. Otherwise we set it to 0 and the pixel is erased. This output image is the input for the next iteration. This is done repeatedly till we find that there occurs no change during an iteration.

The conditional and unconditional filters to be used are as shown in the approach of problem 3.a.

III. Result and Discussion

We were given the jigsaw_1 image to apply the thinning filter:



Original Image

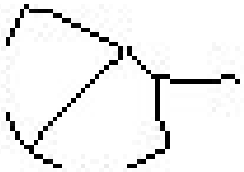


Image after applying thinning Filter.

Problem 3.(c)

I. Abstract and Motivation

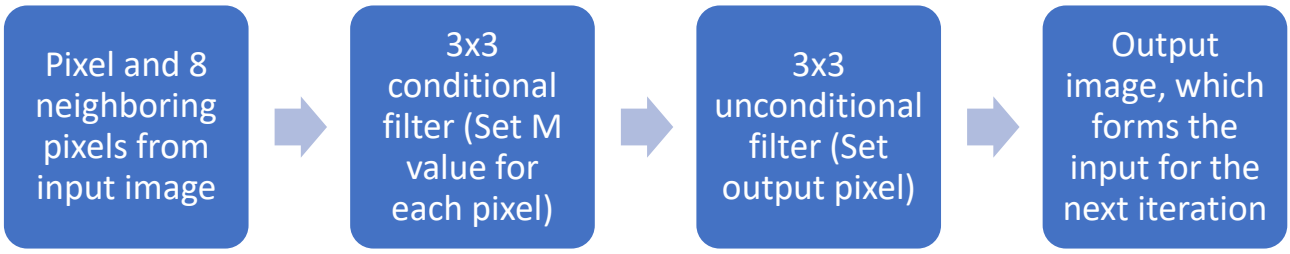
In this problem, we were asked to apply the skeletonizing filter to the jigsaw_2 image. Skeletonizing is a morphological operation. Morphological processing techniques are used to remove imperfections in binary images by using non linear operations to work on the form and structure of the image.

II. Approach and Procedures

I used C++ to write the code for the question. The algorithm is as follows:

First, the grayscale image is converted to binary by thresholding the image and setting all values below 127 as 0 and above as 255. The code uses two sets of filters to match and shrink the image : conditional and unconditional filters. Using these predefined filters, each pixel of the image is scanned along with it's eight neighboring pixels and checked whether they match the filters or not(hit or miss filters). Based

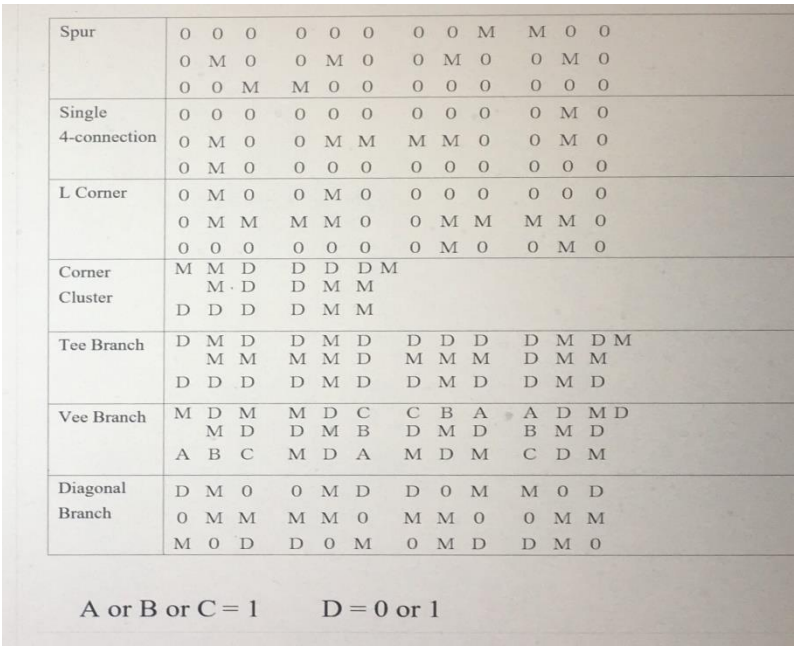
on match, we take decision on the pixel, i.e., if match, we change the pixel value according to problem statement and otherwise, we copy the original pixel value for the output image.



We use 3x3 filters despite there being a possibility of over erasure leading to loss of connectivity in object in image. 5x5 filters would give better result but the total number of filters in that case would be much larger. To accommodate for this we use cascaded two 3x3 filters.

When the input image pixel goes through the initial 3x3 conditional filter, it checks whether to erase or not. If there's a hit(match) between the input and conditional filter, we set M=1 for the pixel which indicates that the pixel value may be changed from 1 to 0. If it doesn't match, we set M=0, meaning the pixel value remains as it is. This is done for the entire input image and a temporary weight_m image is created to store the corresponding M values. This new temporary image is passed through the 3x3 unconditional masks to check whether we need to erase the pixel value or not. If it matches the unconditional filter, we set value of new output image as 1, i.e., retain pixel value. Otherwise we set it to 0 and the pixel is erased. This output image is the input for the next iteration. This is done repeatedly till we find that there occurs no change during an iteration.

The conditional to be used are as shown in the approach of problem 3.a. The unconditional filters are as follows:

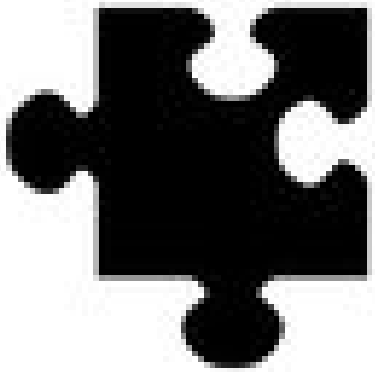


Spur	0 0 0 0 M 0 0 0 M	0 0 0 0 M 0 M 0 0	0 0 M 0 M 0 0 0 0	M 0 0 0 M 0 0 0 0
Single 4-connection	0 0 0 0 M 0 0 M 0	0 0 0 0 M M 0 0 0	0 0 0 M M 0 0 0 0	0 M 0 0 M 0 0 0 0
L Corner	0 M 0 0 M M 0 0 0	0 M 0 M M 0 0 0 0	0 0 0 0 M M 0 M 0	0 0 0 M M 0 0 M 0
Corner Cluster	M M D M D D D D	D D D M D M M D M M		
Tee Branch	D M D M M D D D	D M D M M D D M D	D D D M M M D M D	D M D M D M M D M D
Vee Branch	M D M M D A B C	M D C D M B M D A	C B A D M D M D M	A D M D B M D C D M
Diagonal Branch	D M 0 0 M M M 0 D	0 M D M M 0 D 0 M	D 0 M M M 0 0 M D	M 0 D 0 M M D M 0

A or B or C = 1 D = 0 or 1

III. Result and Discussion

We were given the jigsaw_2 image to apply the thinning filter:



Original Image

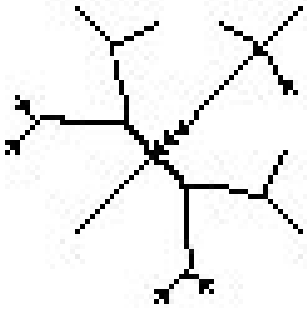


Image after applying the skeletonizing filters

Problem 3.(d)

I. Abstract and Motivation

In this place, we were given the image board.raw which has certain puzzles on it. We were supposed to do two things with the image:

a) Count the total number of pieces on the board image.
b) Find the number of unique pieces on the board. For this we had to check for the rotation, flipping and other geometrical operations on the puzzles in the image.

II. Approach and Procedures

To count the number of puzzles , I used the BFS algorithm. Breadth First Search (BFS) is a way of going through the nodes in a tree one by one. In this method, we define a double ended queue. Whenever we hit a black pixel, we enter it's coordinates into the queue and check for the neighboring four pixels at up, down, left and right. If any of these also contain black, then their coordinates are also entered in the queue. Then the original pixel is popped out and we do the same algorithm for the next value in the queue. Whenever we push a coordinate into the queue, we turn that pixel white to shrink the puzzle. Only the original pixel is kept black. Using a counter to count the total number of black pixels in the image, we can get the total number of puzzles in the box.

Now, to count the total number of unique puzzles in the box, I first applied BFS algorithm to segment and classify each puzzle present in the image separately. This gives me a new matrix with the labels for each segment in increasing order.

My thinking was that, once labelled, I can apply shrinking filter on the whole image. Then if two puzzles are same, despite being rotated or flipped, they are going to give the same output after application of shrinking filter or skeletonizing filters. So they must have the same number of pixels for similar puzzles, give or take some margin. But this does not seem to be the case because the puzzles originally are of different sizes, despite being similar.

So I found out the top left coordinate of each labeled puzzle and created an image with just the square black puzzles of size 45*45. Then I did the XOR operation between this image and my original image. This gives me an image with just the protrusions and holes of each puzzle in the output.

Then I made a matrix of size 16*4 which stores whether there's a protrusion or a hole in each side of a puzzle. This was got by checking whether the pixel above or below is white for each side of the puzzles. If there's a protrusion, a 1 was saved in the matrix corresponding to the puzzle and side. Similarly, if there's a hole, a 2 was saved in the matrix corresponding to the puzzle and side. If the side is plain, then 0 is stored in the array. This was done by once storing in array Puzzle by going in counterclockwise manner and in Puzzleanti by going in anticlockwise manner. This is done because we have to check for both rotation and flipping.

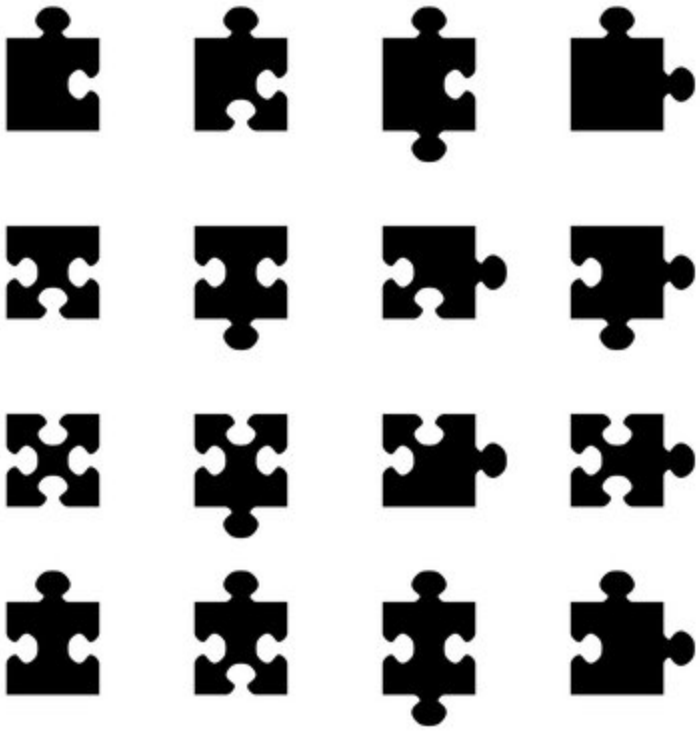
So for example, the first puzzle has a protrusion on the top of the square and the hole on the right side. So the first row of the array Puzzle which is read in clockwise manner is given by 1200. Whereas, the first row of the array Puzzleanti which is read in anticlockwise manner is given by 1002.

Once these arrays were created, the total number of unique puzzles were calculated by going through each array and comparing each rotation of each row of array. So creating a number by

```
num1 = puzzle[i][0]*1000+puzzle[i][1]*100+puzzle[i][2]*10+puzzle[i][3]*1
```

and then comparing this number with different numbers generated by rotating the position of each element throughout the arrays. This gives the total number of unique arrays in the image.

III. Result and Discussion



Original Image

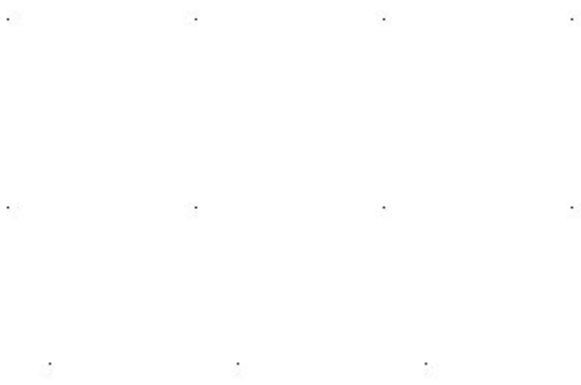
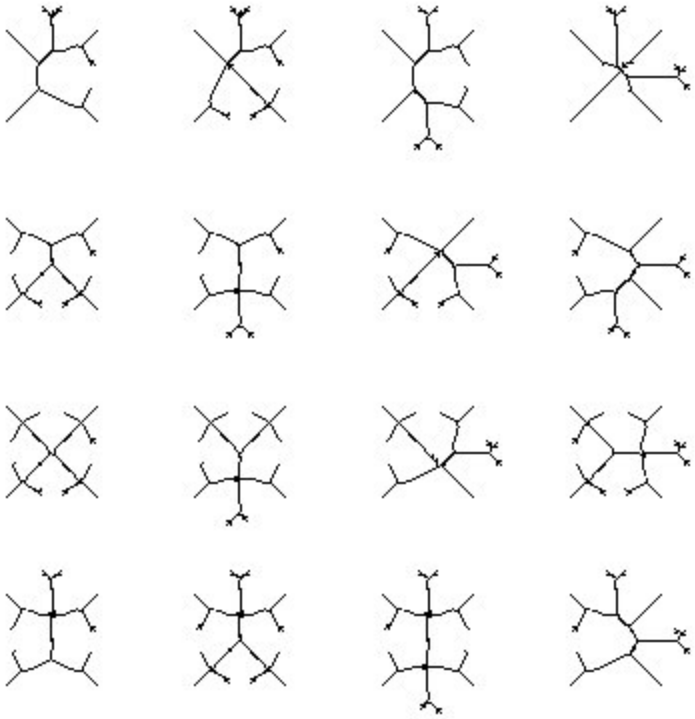
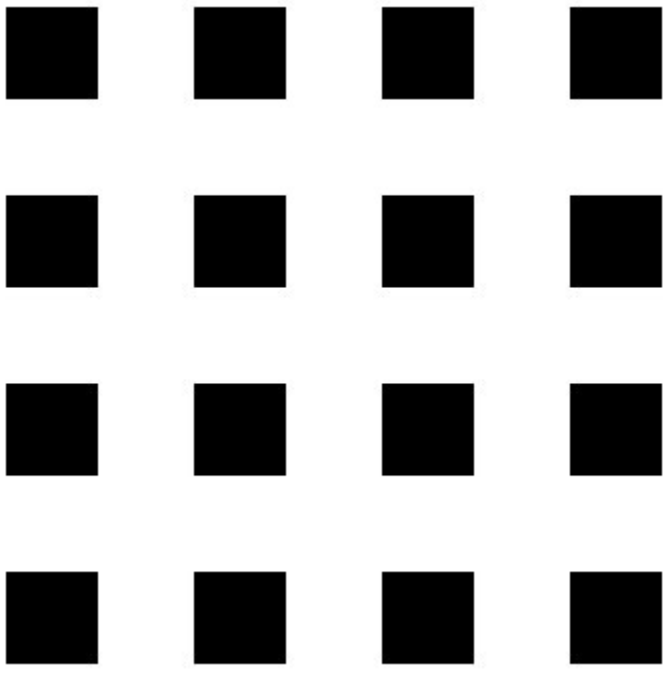


Image after applying BFS algorithm showing the shrunken puzzles as 16 dots.

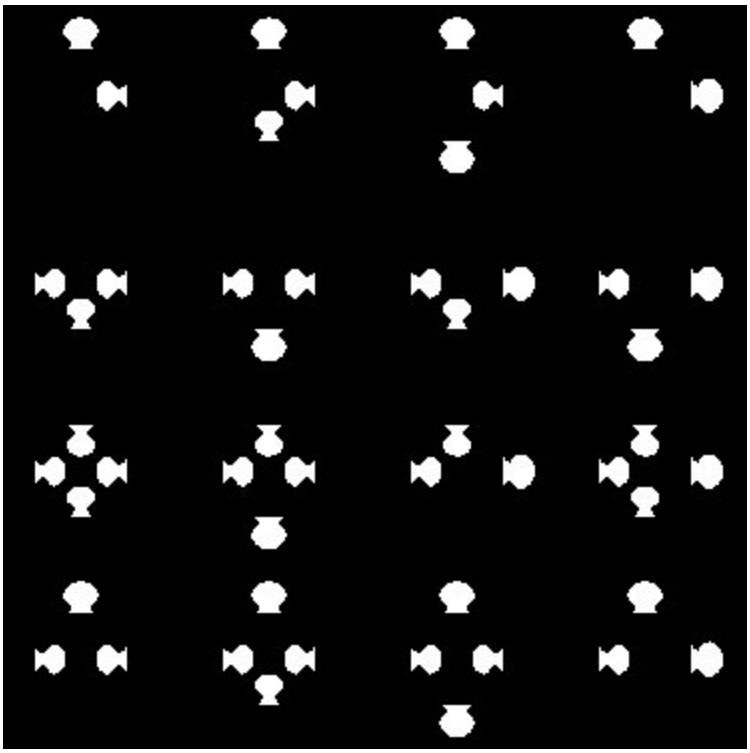
This gives us that the total number of pieces in the board image are 16.



Skeletonization applied to original image



Squares of the puzzles – image formed by marking starting point



Output image got by XOR of original image and the previous image.

Using this, we found out that the total number of unique pieces in the board image are 10.

References:

1. Wikipedi.org
2. Analytical Methods for Squaring the Disc Chamberlain Fong spectralfft@yahoo.com Seoul ICM 2014
3. Digital Color Halftoning – Farhan A. Baqai, Je-HO Lee, A Ufuk Agar and Jan P Allebach
4. D Shaked, N Arad, A Fitzhugh, I Sobel “Color Diffusion : error diffusion for color halftones”, HP Labs Technical report, HPL-96-128R1, 1996