EE569 - INTRODUCTION TO DIGITAL IMAGE PROCESSING
HW # 2 - 03/04/2018

Table of Contents

PROBLEM 1: GEOMETRIC IMAGE MODIFICATION

1.1 MOTIVATION :

A geometrical transformation of a source image to a desired image transforms the source image pixel locations to desired image pixel locations and assigns them with intensity values. Geometric transformations finds its applications in remote sensing, radiology, computer graphics, image stabilization. etc.
When an image of interest is taken from two different angles, geometric transformations can be applied to brings out points of coincidence.
It is usually made up of two steps - mapping coordinates of the input image to the point in the output image and, the obtained output coordinates should be calculated as real numbers as the coordinates obtained might not match the digital grid. Hence bilinear interpolation is necessary.
Some of the common geometric transformations are given below -

Transformation Name	Affine Matrix, T	Coordinate Equations	Example
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = v$ $y = w$	
Scaling	$\begin{bmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = c_x v$ $y = c_y w$	
Rotation	$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = v\cos\theta - w\sin\theta$ $y = v\cos\theta + w\sin\theta$	
Translation	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$	$x = v + t_x$ $y = w + t_y$	
Shear (vertical)	$\begin{bmatrix} 1 & 0 & 0 \\ s_v & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = v + s_v w$ $y = w$	
Shear (horizontal)	$\begin{bmatrix} 1 & s_h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = v$ $y = s_h v + w$	

1.2 APPROACH :

Image mapping procedure can be split into two methods - forward mapping and inverse mapping.

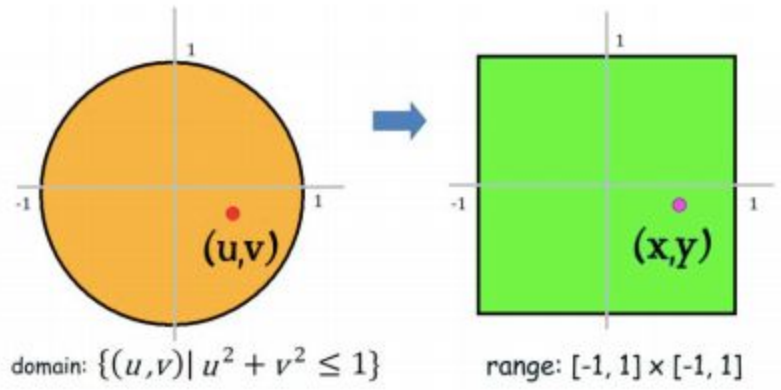
In forward mapping, the transformation is applied on the input pixel coordinates. Using the mapping function f(x,y) where x and y are the pixel coordinates of the input image is mapped to (u,v) of the output image.
As it might not be continuous mapping due to the discretization of the output coordinates, some of the pixels when mapped might produce holes and overlaps. Holes are patches of not defined pixel values at irregular positions. Overlaps occur when multiples input coordinates are mapped to the same coordinate in the desired image.

In inverse mapping, we are projecting each output coordinate into an input coordinate. This method guarantees that all output pixel coordinate values are computed.

1.2.a. Geometrical Warping

The given problem statement is to map the square image to a disc and convert it back to square upholding the requirements that pixels lying on boundaries should still lie on the boundaries of the circle, the center of the original and the morphed should match, and that the mapping should be reversible.

To map input coordinates to output coordinates, normalize the pixel coordinates to lie between [-1,1] with (0,0) as its center pixel. I have implemented a mapping called as elliptical grid mapping.



Square to disc mapping is achieved using the below formula. This is discussed in Nowell's blog in 2005. Here (u, v) are the coordinates of the disc image and (x, y) are the square image coordinates.
Consider an ellipse equation for some constant x :

$$1 = \frac{x'^2}{x^2} + \frac{y'^2}{b^2}$$

To account for the points along the curve on top between -1 and 1, ellipse should pass through the point

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \frac{x}{\sqrt{2}} \\ \sqrt{1 - \frac{x^2}{2}} \end{bmatrix}$$

Solving for b and then for x , we arrive at the equation :

Square to disc mapping:

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

For the reverse spatial warping, the circle coordinates were first converted to polar coordinates and then the trigonometric identities were replaced with equivalent length ratios

Disc to square mapping:

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

Source : Analytical Methods for Squaring the Disc , Chamberlain Fong, Seoul ICM 2014

1.2.a.1 ALGORITHM :

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.

4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
6. Calculate the current pixel location using the row and the column indices.
7. Normalize the input coordinates to lie between [-1,1] x [-1.1].
8. Apply the square to disc mapping formula obtain the (u,v) coordinates.
9. Multiply and add with the middle offset to the obtained (u,v) coordinates and write to a new output image
10. From the obtained (u,v) values, calculate the inverse spatial mapping using the disc to square mapping formula.

Multiply and add with the middle offset to the obtained new (x,y) coordinates and write to another new output image.

1.2.b.Homographic transformation and Image stitching

Images of a plane taken from different perspectives can be matched using homography transformation. 4 points pairs of interest were selected using the MATLAB cpselect tool. The homographic transform can be expressed in terms of 9 coefficients with assuming p33=1, without the loss of generality.

Linear transformation with matrix P

$$\bar{x}^* = P\bar{x} \qquad P = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & 1 \end{pmatrix} \qquad \begin{aligned} x^* &= p_{11}x + p_{12}y + p_{13} \\ y^* &= p_{21}x + p_{22}y + p_{23} \\ z^* &= p_{31}x + p_{32}y + 1 \end{aligned}$$

Perspective equivalence

$$\begin{aligned} x' &= \frac{p_{11}x+p_{12}y+p_{13}}{p_{31}x+p_{32}y+1} \\ y' &= \frac{p_{21}x+p_{22}y+p_{23}}{p_{31}x+p_{32}y+1} \end{aligned}$$

Multiply by denominator and reorganize terms

$$\begin{aligned} p_{31}xx' + p_{32}yx' - p_{11}x - p_{12}y - p_{13} &= -x' \\ p_{31}xy' + p_{32}yy' - p_{21}x - p_{22}y - p_{23} &= -y' \end{aligned}$$

Linear system, solve for P

$$\begin{pmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x_1' & y_1x_1' \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_2' & y_2x_2' \\ & & & \vdots & & & & \\ -x_N & -y_N & -1 & 0 & 0 & 0 & x_Nx_N' & y_Nx_2' \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y_1' & y_1y_1' \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_2' & y_2y_2' \\ & & & \vdots & & & & \\ 0 & 0 & 0 & -x_N & -y_N & -1 & x_Ny_N' & y_Ny_N' \end{pmatrix} \begin{pmatrix} p_{11} \\ p_{12} \\ p_{13} \\ p_{21} \\ p_{23} \\ p_{23} \\ p_{31} \\ p_{32} \end{pmatrix} = \begin{pmatrix} -x_1' \\ -x_2' \\ \vdots \\ -x_N' \\ -y_1' \\ -y_2' \\ \vdots \\ -y_N' \end{pmatrix}$$

Source : http://www.eng.utah.edu/~cs6640/geometric_trans.pdf

Here, all the primes (‘) belong to the output or the second image (Here the middle image) and the variables without primes are first image control points.

Based on the homography transformation above, we can see that we need P inverse matrix to find the points in the corresponding left and right image when traversing through the bigger embedded image.
The P inverse matrices (I have used an array) for left to middle transformation was found to be -

P_left_inv[9] = 100 * {0.007085522931281; -0.001410493508118;
-0.767094289296011;-0.000174795607964;0.005246967576371;-2.021456226406839;-0.000000316624742;-0.000005040236436;0.012032025806476}

P_right-inv[9] = 1000 * {0.001110981575213
;0.000320334477785;-0.702220261254909;0.000001011231121;0.001247497630304;-1.260823570050726;-0.000000041246898;0.000000950951874;0.000053206924333 }

1.2.b.1 ALGORITHM :

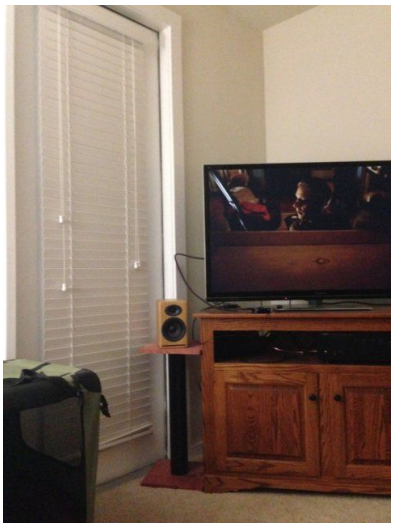
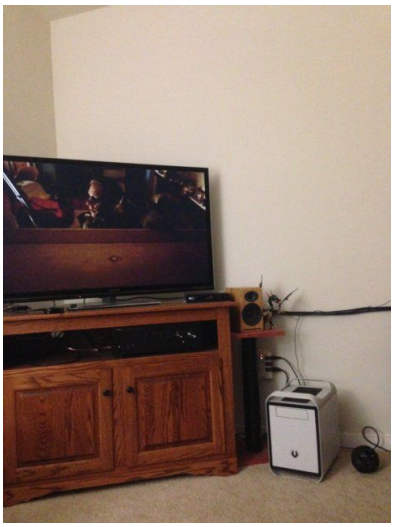
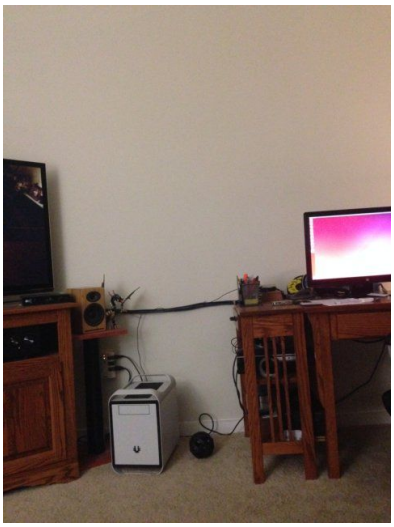
1. Load the input raw image using the function ‘load_image_from_file’ function.
2. Initialize the ‘Image’ class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested ‘for’ loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of ‘Image’ objects to be treated as outputs for the image processing operations.
6. Embed the middle.raw in a bigger image preferably of height = 1.5 times actual height and width almost 2.5 times of the actual width.
7. Run the for loops traversing the bigger embedded image and initialize the homogeneous coordinate vector with the corresponding row, col and a unit value.
8. Matrix multiply to find the corresponding side (left/right) homogeneous image coordinates.
9. Calculate the weight of the third homogeneous coordinate and multiply the weight with the first element of the homogeneous coordinate and the second element of the homogeneous coordinate.
10. Apply bilinear interpolation, to interpolate using the values of the neighboring pixels and then average out the pixel intensities in the left/right image with the middle image.
11. Do the same to the right.raw and obtain the stitched image.

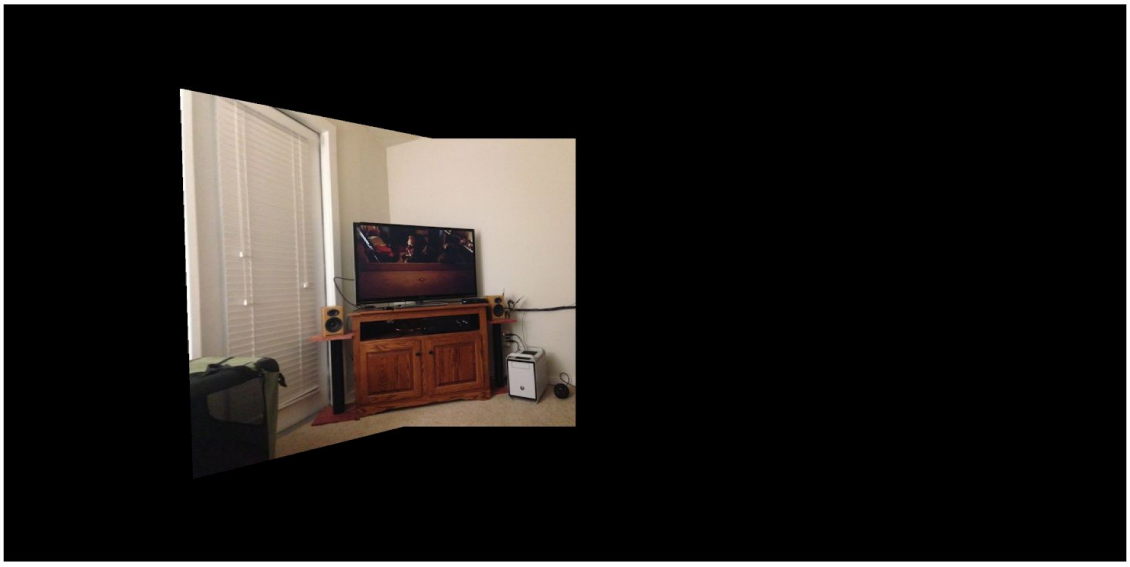
1.3 RESULTS :

1.3.a. Geometrical Warping

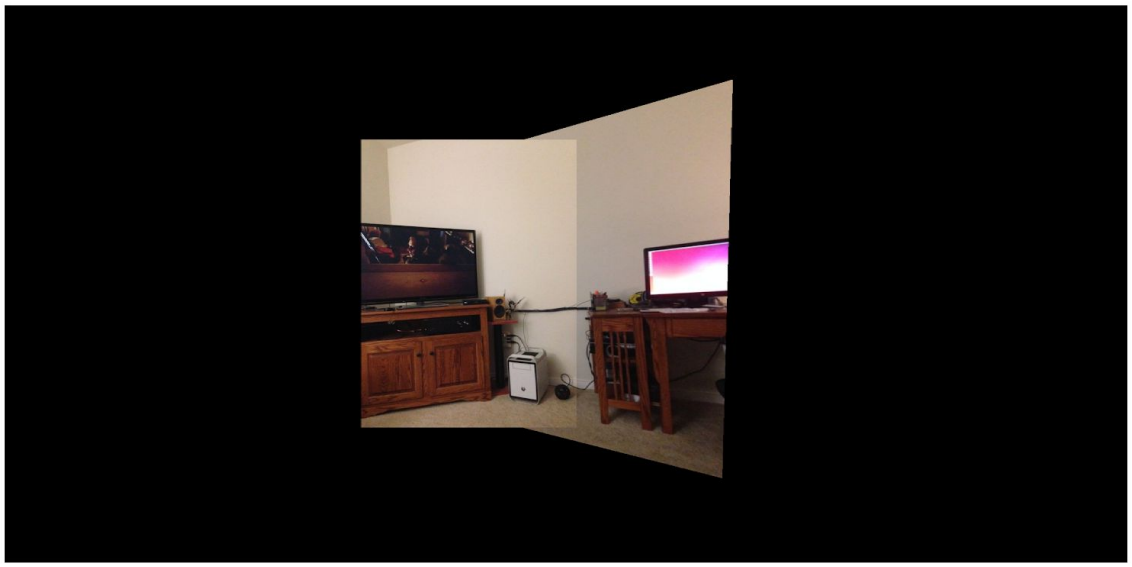
Original image	Square to disc mapping	Disc to square mapping
		
		
		

1.3.b.Homographic transformation and Image stitching

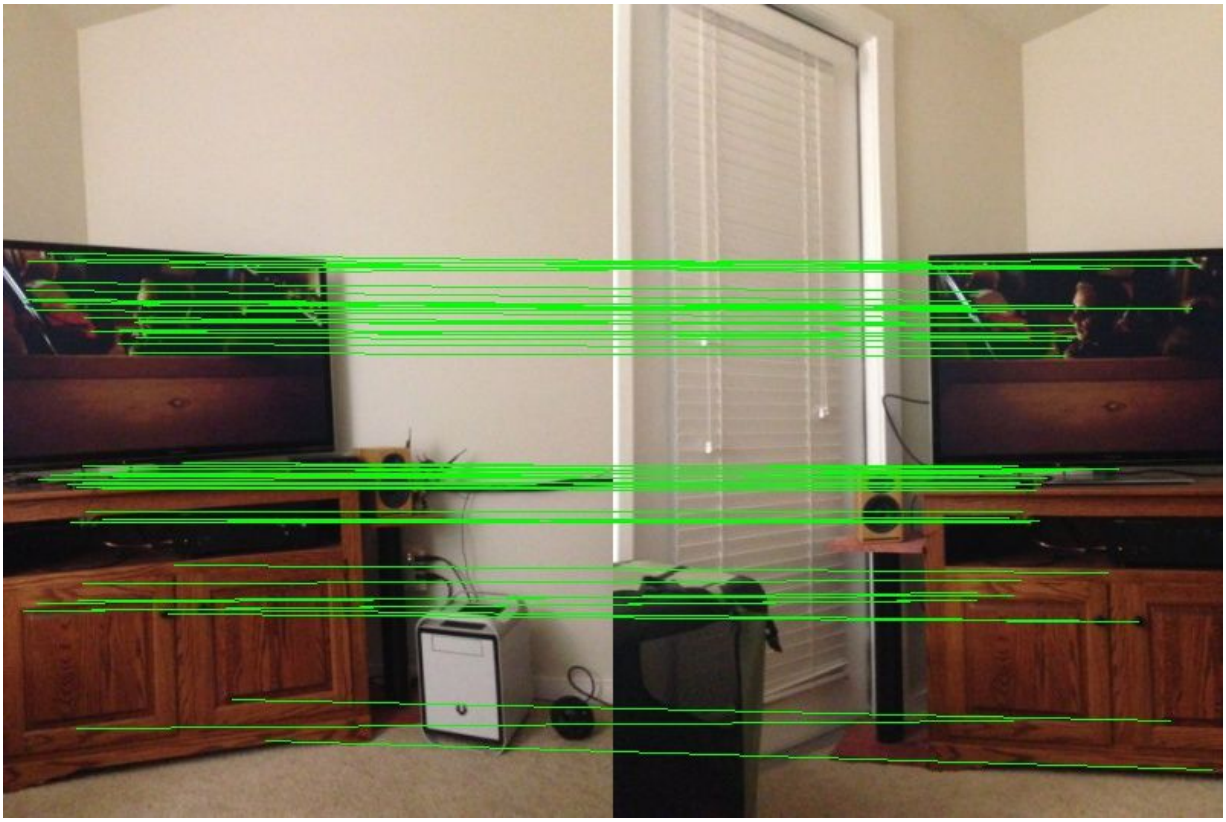




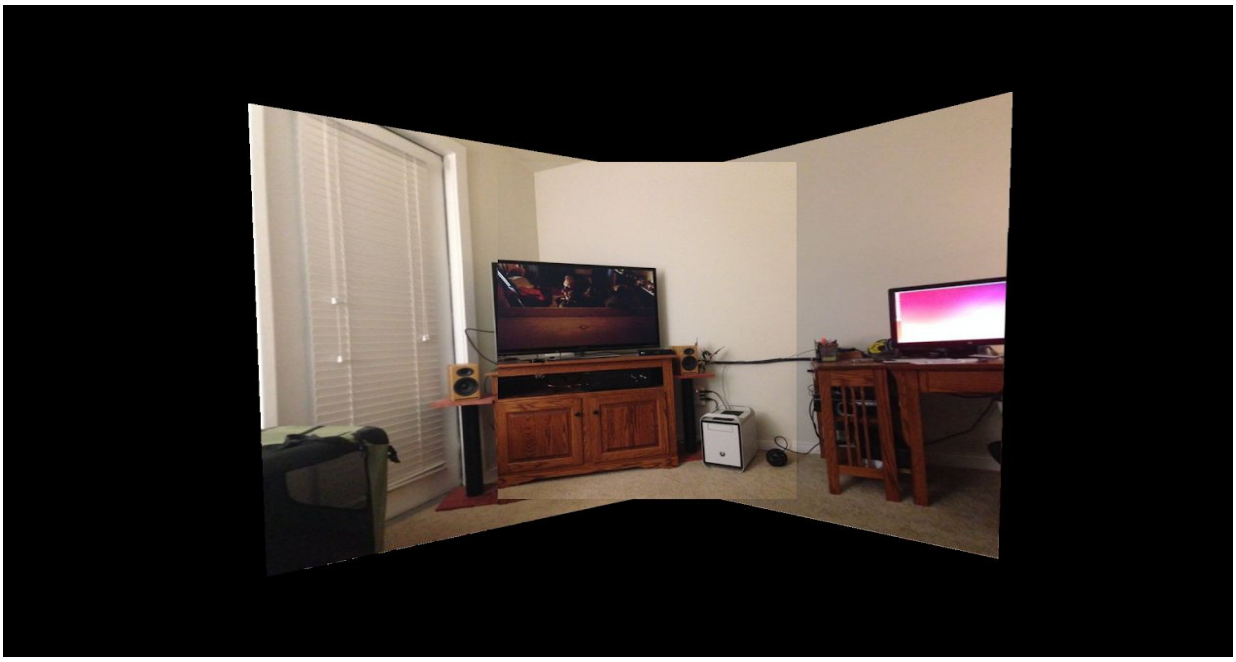


Left and middle transformation



Right and middle transformation





The final output image

1.4 DISCUSSION :

1.4.a. Geometrical Warping

	
ORIGINAL IMAGE	**SQUARE TO DISC MAPPING**

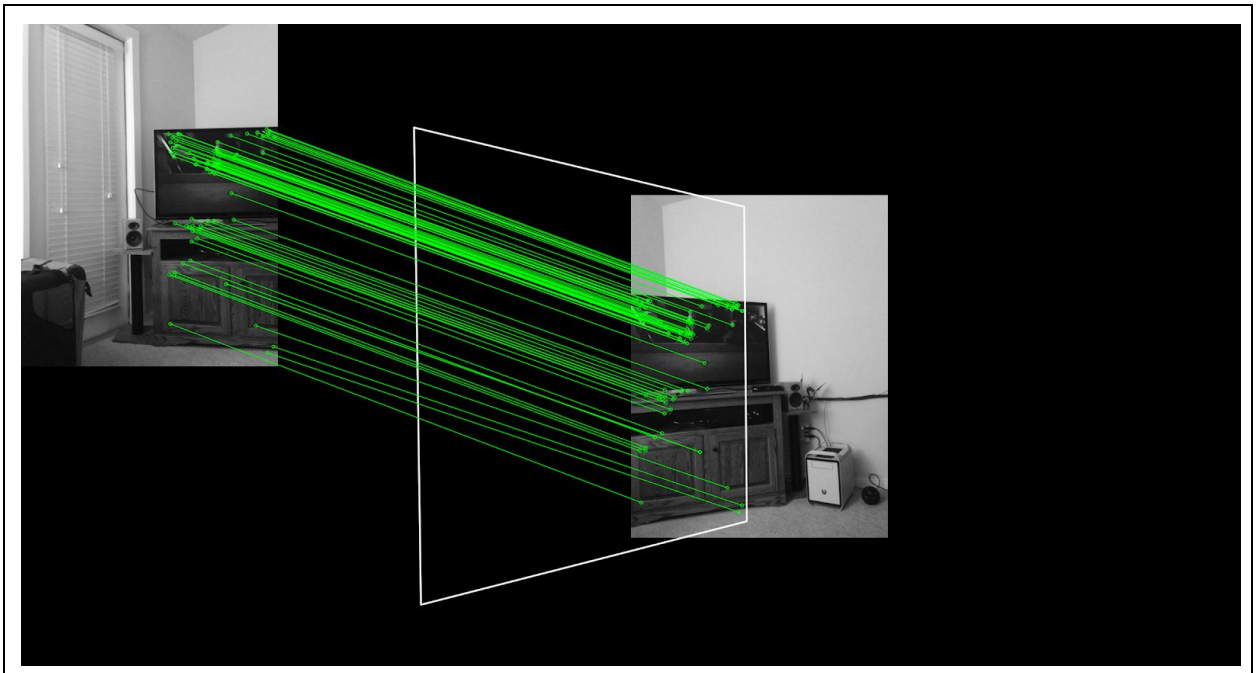
 **DISC TO SQUARE MAPPING**	

We can see from the above results for puppy.raw (it holds good for the other input original images- panda and tiger.raw) that the resultant image looks similar to the input image. When we are mapping from the input square image to the disc image, we are converting the indices to the normalized indices values, and then rounding off the obtained floating point values so that we get integral indices. In the disc to square elliptical mapping as well, we are rounding off the obtained index values, hence there are no non -integral values and we are mapping everything back again into the square image. As we are mapping it directly, we do not observe any losses near the corners.

1.4.b.Homographic transformation and Image stitching

For understanding this question better, it was first implemented in MATLAB using 4 (first choice) points which were selected using cpselect tool. As the resulting output was unsatisfactory, I coded a SIFT and RANSAC algorithm feature detector to give me the best matching points in the image that could be considered.

I have chosen 4 control points in the form of a trapezium. As we are taking 4 control points from each of the image, we have 8 control points and thus 8 equations in total. Hence applying the equations in the approach part of the section, we solve the matrix and obtain the homography transformation. RANSAC is an iterative algorithm, mostly an outlier detection method which returns the homography matrix using only the best matching points.



The best control points between the left image and the embedded middle image is shown above.

The control points that I have chosen are -
Row_offset = 300
Col_offset = 795

Embedding width = 2500
Embedding height = 1240

Left.raw	Embedded middle image
(473,282)	482+row_offset, 822)
(424,338)	(424+row_offset, 883)
(362,350)	(360+row_offset, 895)
(204,293)	(195+row_offset, 831)

Right.raw	Embedded middle image

(366,258)	372+row_offset, 1264)
(525,258)	(544+row_offset, 1262)
(436,125)	(436+row_offset, 1119)
(224,47)	(237+row_offset, 1049)

These points form a trapezium, hence it is an enclosed polygon and the mapping is effective.

PROBLEM 2: DIGITAL HALF-TONING

2.1 MOTIVATION :

Many computer imaging displays do not have the capability to reproduce digital images. Hence to render digital images dithering was introduced.
Halftoning had been practiced widely in printing press to display continuous tone images with only black or white dots. Effective digital halftoning can cut down the cost and at the same improve the quality of rendered images.

Dithering is one of the techniques of thresholding where the input pixel intensity is converted to a bunch of foreground and background pixels. Dithering is one of the primary halftoning techniques which tricks the human eye by replacing a bunch of gray levels with suitable placed black and white dots. It improves the colordept of the image by using the existing color palette.

It was found that blue noise introduced unpleasant artifacts in the image. Hence, to get rid of the generated artifacts, error diffusion methods were introduced.

Therefore, the three most common methods of generating digital halftoning images :

- Patterning
- Dithering
- Error diffusion

2.2 ABSTRACT :

2.2.a.1 DITHERING - Fixed Thresholding

This is one of the simplest methods of dithering. This uses a fixed threshold to compare with the input pixel intensity. If the input image intensity is less than the fixed threshold, then the output image at that exact location gets a black(0) pixel, else it gets a white (255) pixel. As the threshold is not dynamic, this results in a lot of loss of detail and edges contouring. This reduces our perception of gray levels. For example, consider majority of input pixels lie in the range of 60% - 70% of the highest gray level (255). Then all of this will be mapped to 255 if the threshold is set at 50%.Hence our gray perception is lost.
The formula for Fixed thresholding is -

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

ALGORITHM FOR FIXED THRESHOLDING:

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
6. Choose the value of threshold as 127. According to fixed thresholding algorithm, if the intensity of the input pixel lies between 0 and the fixed threshold value, then the output image gets the intensity value of 0.
7. If the input pixel intensity is greater than the threshold value, then the output image at that particular location gets a white(255) intensity pixel.
8. The obtained output is written to a file and the output is checked.

2.2.a.2 DITHERING - Random Thresholding

As the previous method of fixed thresholding was not dynamic enough, this method was introduced. Each pixel intensity is compared with a random threshold generated whose intensities lie between (0,256) - between the white and black gray levels). From the results, we can see that lot of noise is generated and tends to consume the actual input image.

$$G(i,j) = \begin{cases} 255 & \text{if } rand(i,j) \leq F(i,j) \\ 0 & \text{if } rand(i,j) > F(i,j) \end{cases}$$

ALGORITHM FOR RANDOM THRESHOLDING:

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
6. Generate a random number lying in the intensity range (0-255) and set it as the

threshold. This random threshold is generated for every pixel. According to random thresholding algorithm, if the intensity of the input pixel is greater than the random threshold value, then the output image gets the intensity value of 255.

7. If the input pixel intensity is less than the random threshold value, then the output image at that particular location gets a black(0) intensity pixel.
8. The obtained output is written to a file and the output is checked.

2.2.a.3 DITHERING -

Ordered Dithering - Dithering Matrix

Ordered dithering consists of comparing blocks of pixels for input pattern matching. This is called as dither pattern. Each value of the input pixel intensity lying inside the block is quantized according to the calculated threshold value from a series of calculations from the dither matrix.

A dither matrix, also called as the index matrix determines which pixels (dots in dithering) needs to be "turned on". The initial Bayer Index matrix is given by -

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

A 0 indicates the first pixel to turn on, and correspondingly 3 indicates the last pixel to turn on. Bayer also introduced a series of recursive index matrices given by the formula -

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

Where I_{2n} is the new (2N x 2N) matrix and I_n is the previous NxN matrix.

After the recursive calculation of the Index matrix, the threshold matrix is calculated which is used to halftone the image. The thresholding formula is given by -

$$T(x,y) = \frac{I(x,y) + 0.5}{N^2}$$

Where N^2 is the total number of elements in a matrix. This produces evenly distributed thresholds. As the input image is much larger in size compared to the Threshold matrix, we find the equivalent mod values of it and compare with the original pixel intensity. The comparison is given by -

$$G(i,j) = \begin{cases} 1 & \text{if } F(i,j) > T(i \bmod N, j \bmod N) \\ 0 & otherwise \end{cases}$$

G(i,j) is the output pixel intensity at the location (i,j).

ALGORITHM FOR DITHERING MATRIX:

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
6. Initialize the 2x2 Bayer matrix. Use recursion to find the 4x4 and 8x8 Bayer Matrix using the above formula.
7. Run two for loops traversing the length and breadth of the input image.
8. Set up the required Index Bayer Matrix for 2x2, 4x4 or 8x8.
9. Convert the Index Bayer Matrix to Threshold matrix using the given formula.
10. If the input pixel intensity at (i,j) is greater than the obtained threshold value at (i mod numberOfNeighbor, j mod numberOfNeighbor), then the output pixel at that pixel location gets a white pixel.
11. If the threshold condition is not satisfied, then the value at the pixel location in the output image gets a black pixel.
12. The obtained output is written to a file and the output is checked.
13. To display using only 4 intensity levels, check if the input value lies between a specified range of say (0,85) with 85 being excluded, then the intensity value at the pixel location is given to be 0.
14. Write the 4 levels image to a file and check the output.

ALGORITHM FOR 4 LEVELS :

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.

6. Get the intensity value at the current pixel. If it is between 0 and 85 and if it is less than midway to 85, then the output pixel get a 0 else a 85.
7. If the intensity of input pixel is between 85 and 170, and if it is less than midway it gets 85, else it get 170.
8. If the intensity of input pixel is between 170 and 255, and if it is less than midway it gets 170 else it get 255.
9. This way the grayscale levels are quantized to four levels. Write the obtained output to a file.

2.2.b.ERROR DIFFUSION

Error diffusion produces one of the best outputs for digital halftoning. They display a variety of pleasing randomness without human perceiving the underlying dots. The low frequency component is also suppressed in error diffusion.

Error diffusion has an easy implementation. For every pixel intensity in the image, we calculate the nearest neighboring intensity (0 or 255) and then take the error with respect to the original pixel intensity. Then the error is diffused using the below Floyd-Steinberg/ JJN/ Stucki matrices to the neighboring pixels. Then the new error diffused values are updated in the same image. Hence this is a cumulative method, where in every step the error is traced. The method of traversing the image here is 'serpentine' - going left to right for the first row and then right to left for the next row and so on. This serpentine fashion of reading the image makes sure that the error is not accumulated on the right side corner but it spread evenly through the entire image.

The Floyd-Steinberg filter :

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

In the above matrix, consider that the current pixel is the always the center pixel in the Floyd Steinberg matrix. We can see that there are factors to the right of the center pixel. This is a traditional way of scanning the image - that is from left to right. For serpentine scanning, as we read the image from right to left also, we flip the matrix in those iterations.

In the left to right scanning from the above matrix we see that the pixel immediately to the right gets a 7/16 of the error values, the pixel below it gets a 5/16 of the error value and its

diagonally adjacent pixels get a 3/16 and a 1/16 of the error respectively.

This selection of the filter produces a checkerboard pattern in areas of threshold being 127.

The Jarvis, Judice, and Ninke filter :

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

The Floyd-Steinberg filter did not account for the narrowness of their filter. Hence the JJN filter was introduced which is a 5x5 filter and it is shown above. As this filter is wider, this gave a wider distribution of error margin.

We can also see that the numbers in the matrix are not in the form of powers of 2 Hence this leads to slower computation.

Stucki filter:

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

This is similar to JJN filter with the exception that the values inside the matrix is even. Hence the computation time is faster than JJN.

ALGORITHM FOR ERROR DIFFUSION :

1. Load the input raw image using the function ‘load_image_from_file’ function.
2. Initialize the ‘Image’ class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested ‘for’ loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of ‘Image’ objects to be treated as outputs for the image processing

operations.

6. Initialize the Floyd- Steinberg, JJN and Stucki error diffusion matrices.
7. Run two for loops to run through the entire length and breadth of the input image.
8. Check if row is odd or even. If the row is even, then flip the error diffusion matrices and then traverse the input image starting from maximum width to the least column.
9. At the current location, set the threshold by checking the value of the current pixel intensity, if it greater than 127, set the threshold as 255 else set it as 0. Find the error by taking the difference between the original input pixel intensity from the threshold.
10. Get the neighbors of the current pixel and using the Floyd Steinberg /JJN/ Stucki matrix, diffuse the error to the neighboring pixel. Therefore the value of the neighboring pixels are now going to be - original value + (Floyd or JJN or Stucki Multiplying factor) * error.
11. Diffuse the error in the same input image, so that the neighboring values in the input images are updated for the next iterations according to the selected error diffusion technique - either Floyd, JJN, or Stucki.
12. Write the obtained Floyd-Steinberg, JJN and Stucki outputs to a file and check them.

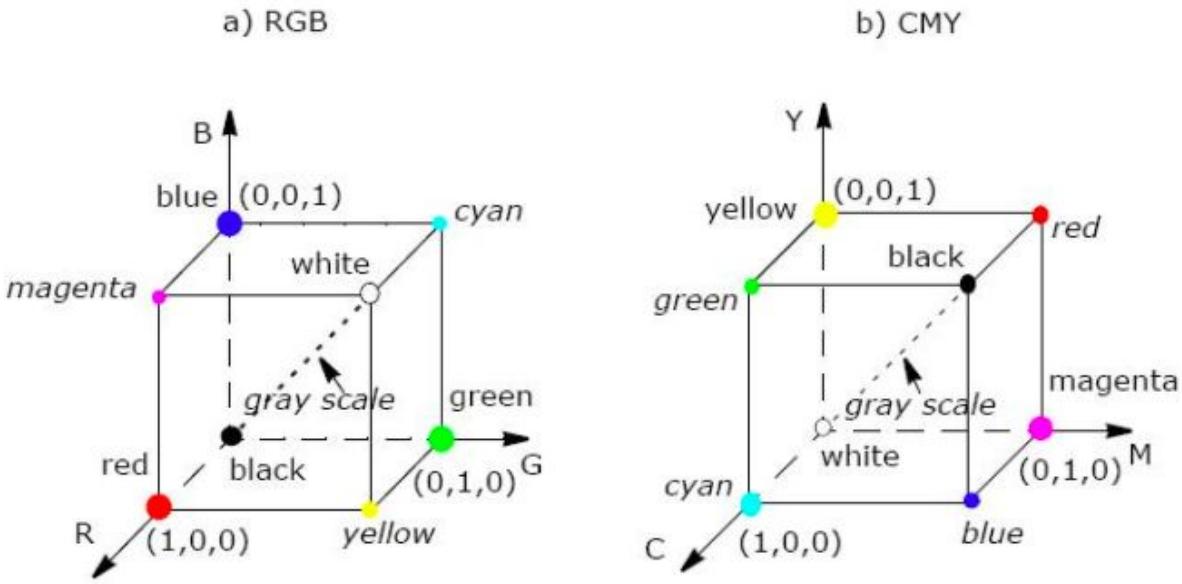
2.2.c COLOR HALFTONING WITH ERROR DIFFUSION

Color error diffusion was introduced to overcome the limitations placed by the grayscale error diffusion. The two primary methods in which color halftoning can be achieved is - Separable Error diffusion and Minimal Brightness Variance Quadruples. To replace the black and white dots prevalent in grayscale halftoning - cyan, magenta and yellow were introduced additionally.

To diffuse error separately, the input image needs to split into its respective channels and then the error needs to be diffused separately. Whereas MBVQ considers the human color perception too.

A given color in the RGB cube can be rendered using 8 corners of the color cube represented by the basic colors. Hence, averaging out values in the error diffusion gets perceptually close to the original undithered image. As MBVQ considered the closest vector which is perceptually identical, MBVQ is often favoured over separable error diffusion.

2.2.C.1. Separable Error Diffusion



Source : https://software.intel.com/en-us/ipp-dev-reference-color-models

In Separable error diffusion, we separate the input image into its respective channels and then convert it into its CMY equivalent where it is quantized separately. In every channel, error diffusion is applied separately and then it is combined to give the final output image.

2.2.C.1.Minimal Brightness Variation Quadruples

Separable error diffusion produces an output with a lot of noise characteristics. To overcome them, MBVQ[2] was introduced. The method divides the RGB cube into 6 quadruples. These 6 quadruples have minimum intensity variation between them and hence can be approximated

to the given input intensity.

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

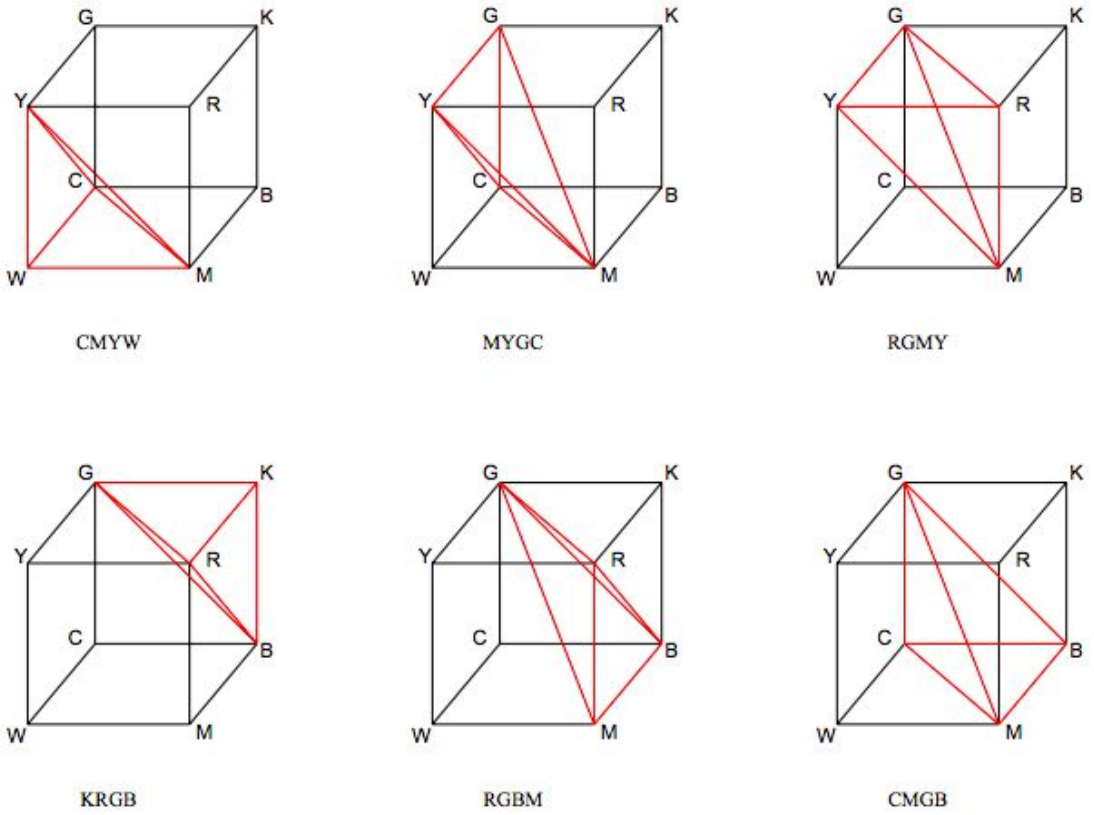


Figure 2: The partition of the *RGB* de toube to six tetrahedral volumes, each of which the convex hull of the MBVQ used to render colors in it. Note that all the tetrahedra are of equal volume, but are not congruent.

The input image is converted to its CMY complements. The quadruple which it belongs to is given by the decision rule given below :

```
pyramid MBVQ(BYTE R, BYTE G, BYTE B)
{
  if((R+G) > 255)
    if((G+B) > 255)
```

5

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

The distance to the vertices in the tetrahedron is calculated and the vertex with the minimum distance is assigned to the output image at the current location. The error is calculated separately for all the three channels and then later combined.

ALGORITHM FOR SEPARABLE ERROR DIFFUSION :

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of 'Image' objects to be treated as outputs for the image processing

operations.

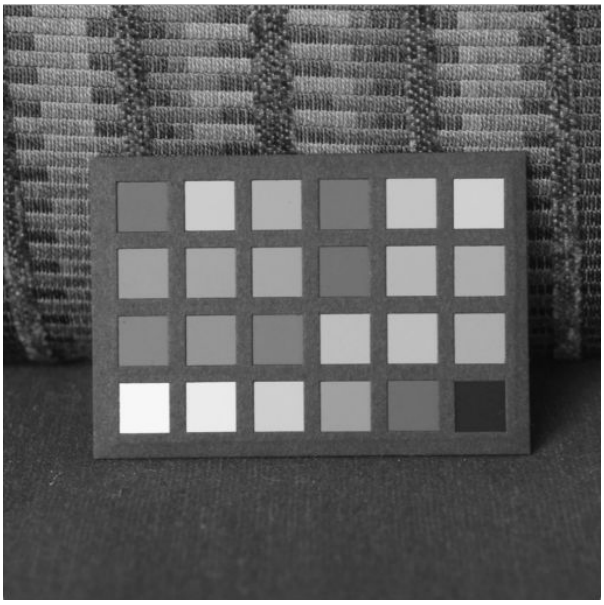
6. Convert the input RGB image to a CMY image. Separate the C,M,and Y channels.
7. Apply Floyd-Steinberg error diffusion for all the 3 channels separately.
8. Combine the three channels and convert back to RGB.
9. Write the obtained output to a file.

ALGORITHM FOR MBVQ :

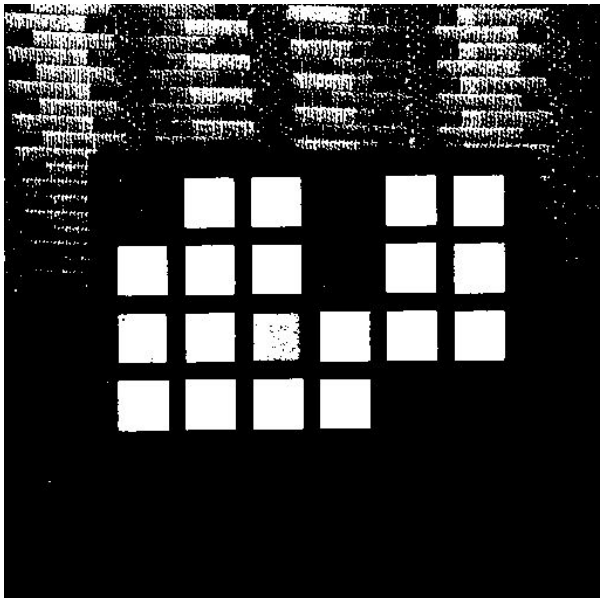
1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
6. Pick the quadruple which the input pixel intensity belongs to by applying the decison rule.
7. After choosing the corresponding quadruple, pass the CMY components of the RGB image to find the minimum distance from the vertices in the respective tetrahedron.
8. Choose the vertex with the minimum distance and assign it to the output pixel at the current location.
9. Apply Floyd-Steinberg error diffusion for all the 3 channels separately.
10. Combine the three channels and convert back to RGB.
11. Write the obtained output to a file.

2.3. RESULTS :

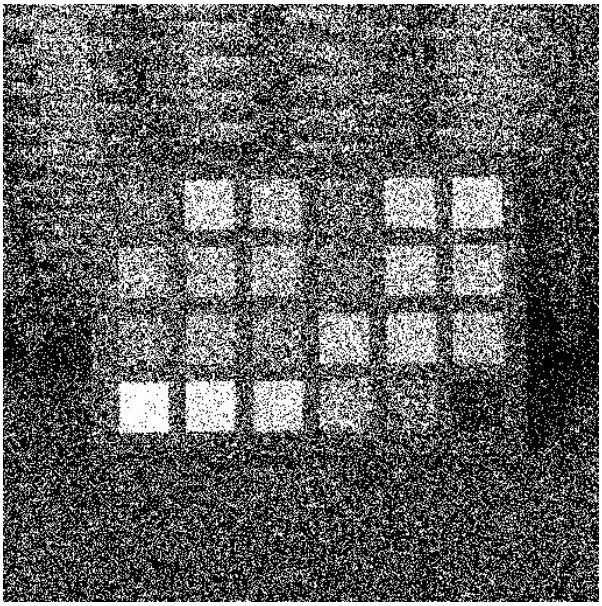
2.3.a.Dithering Matrix -



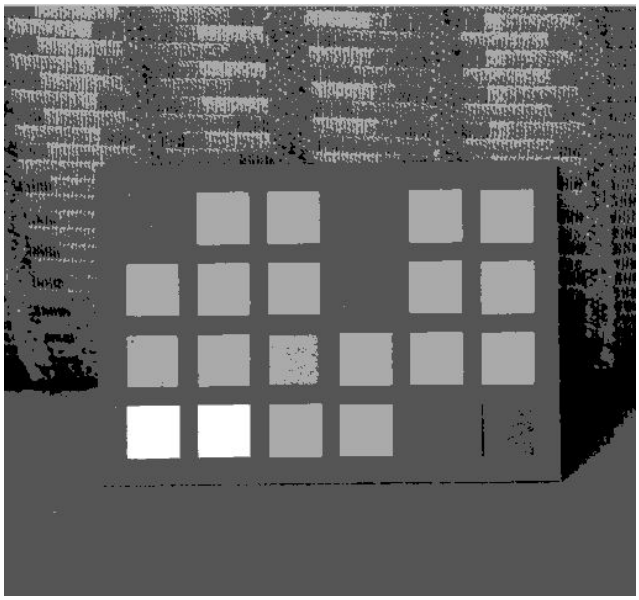
INPUT IMAGE



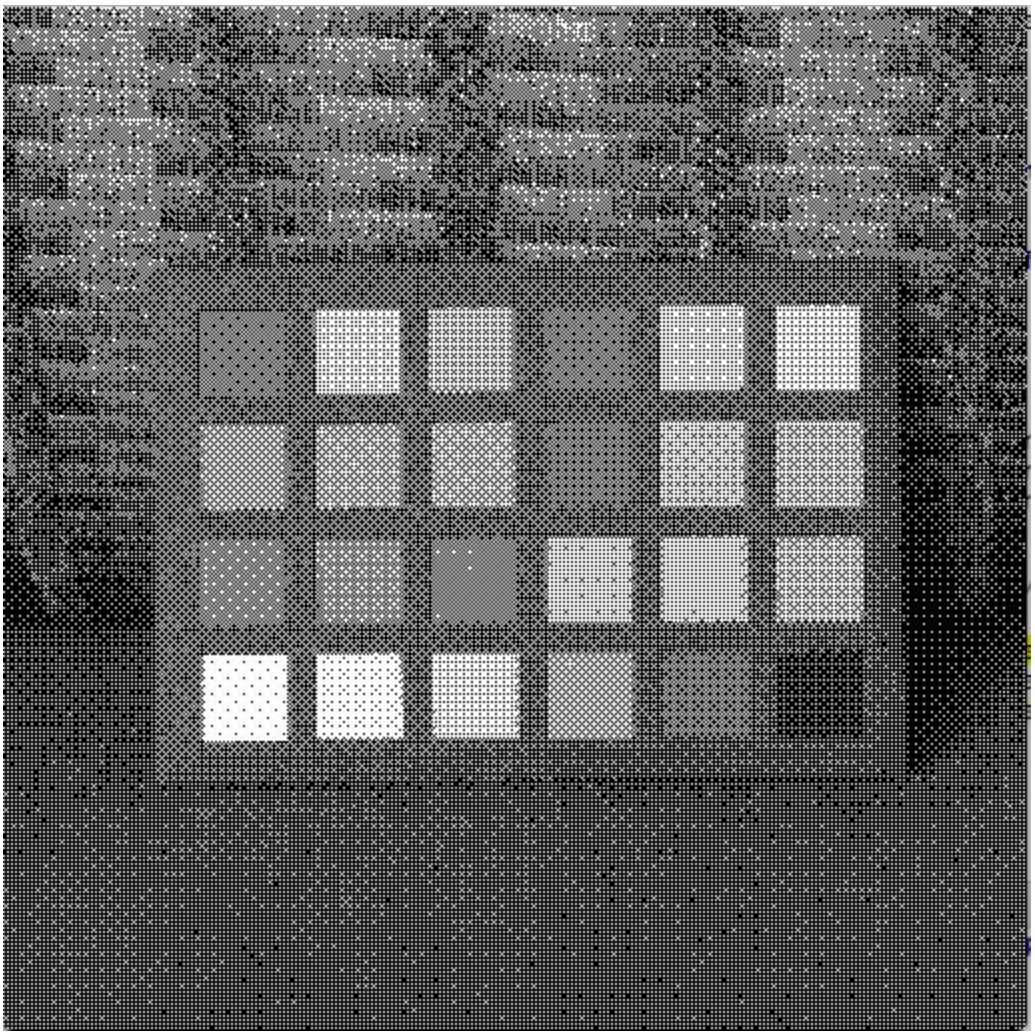
FIXED THRESHOLDING



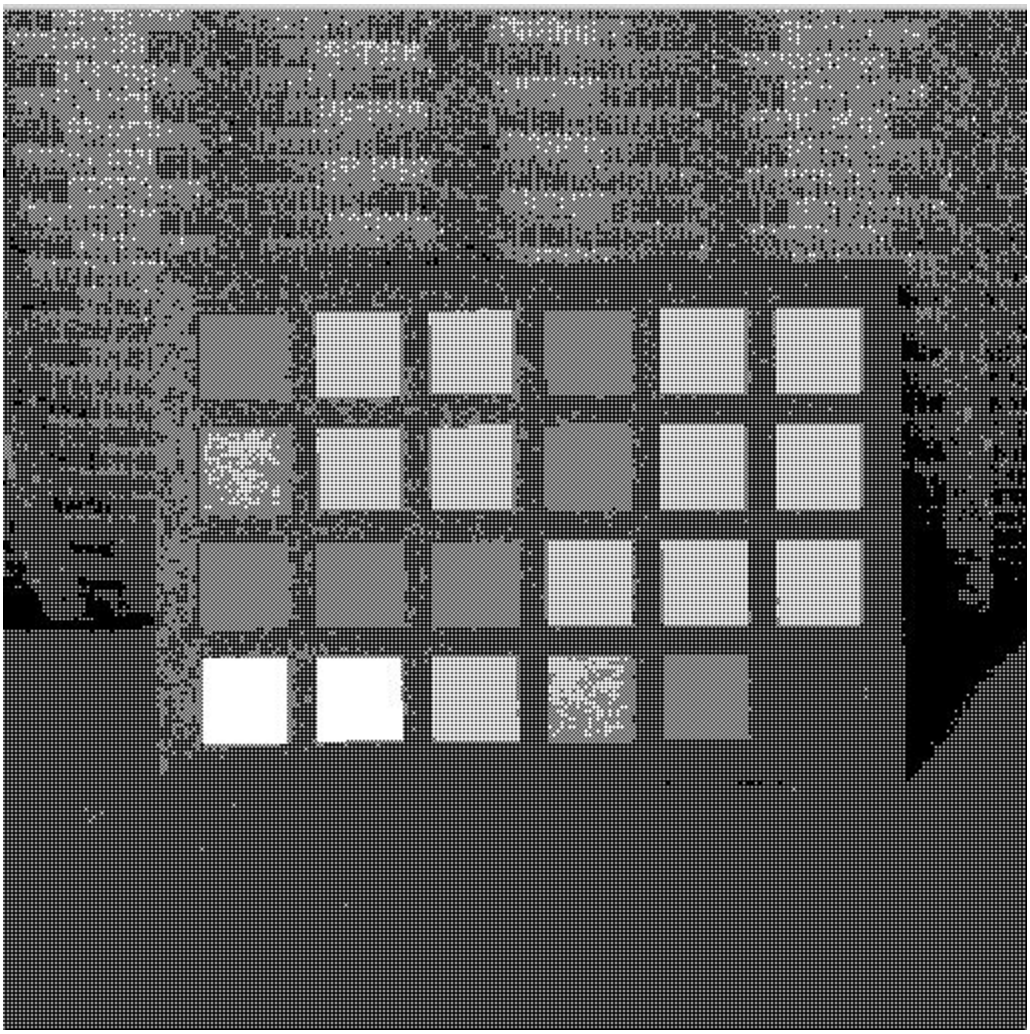
RANDOM THRESHOLDING



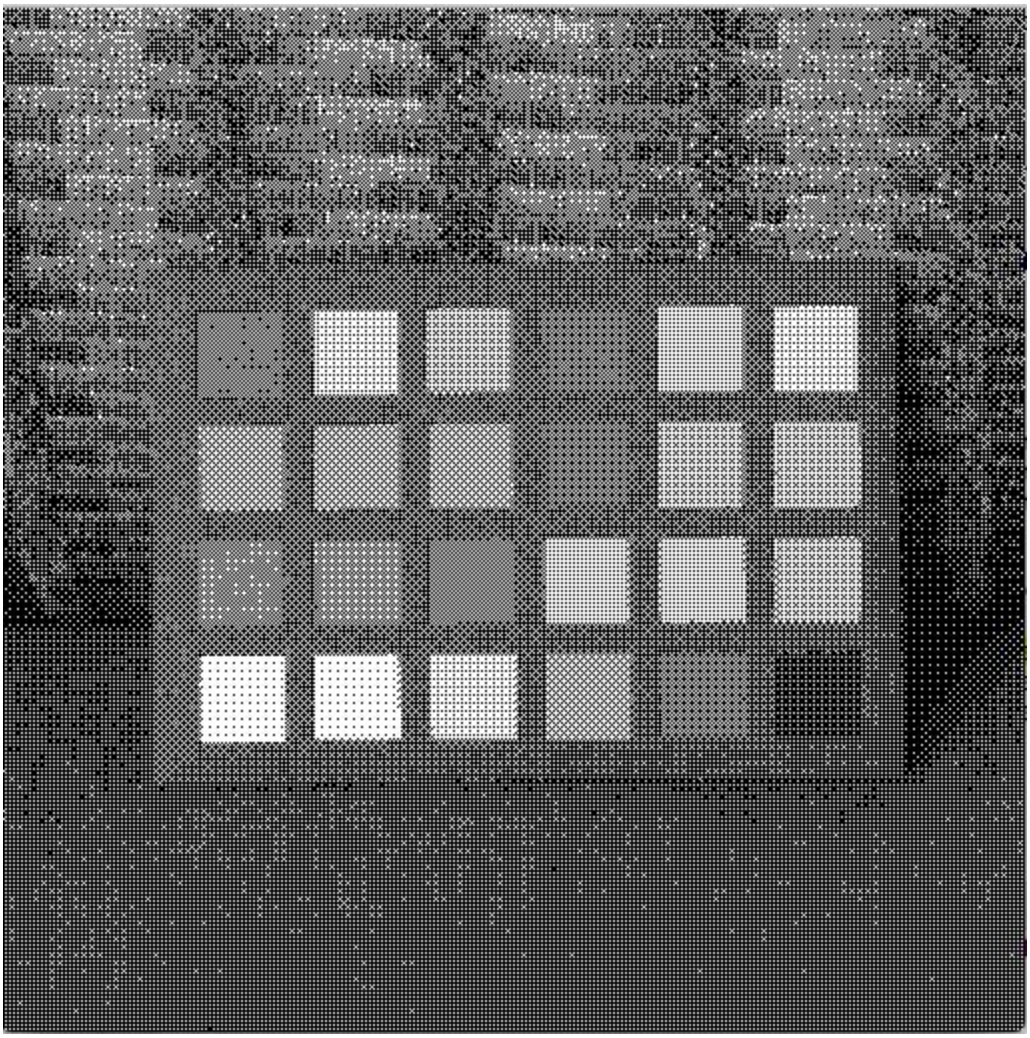
4 LEVELS IMAGE



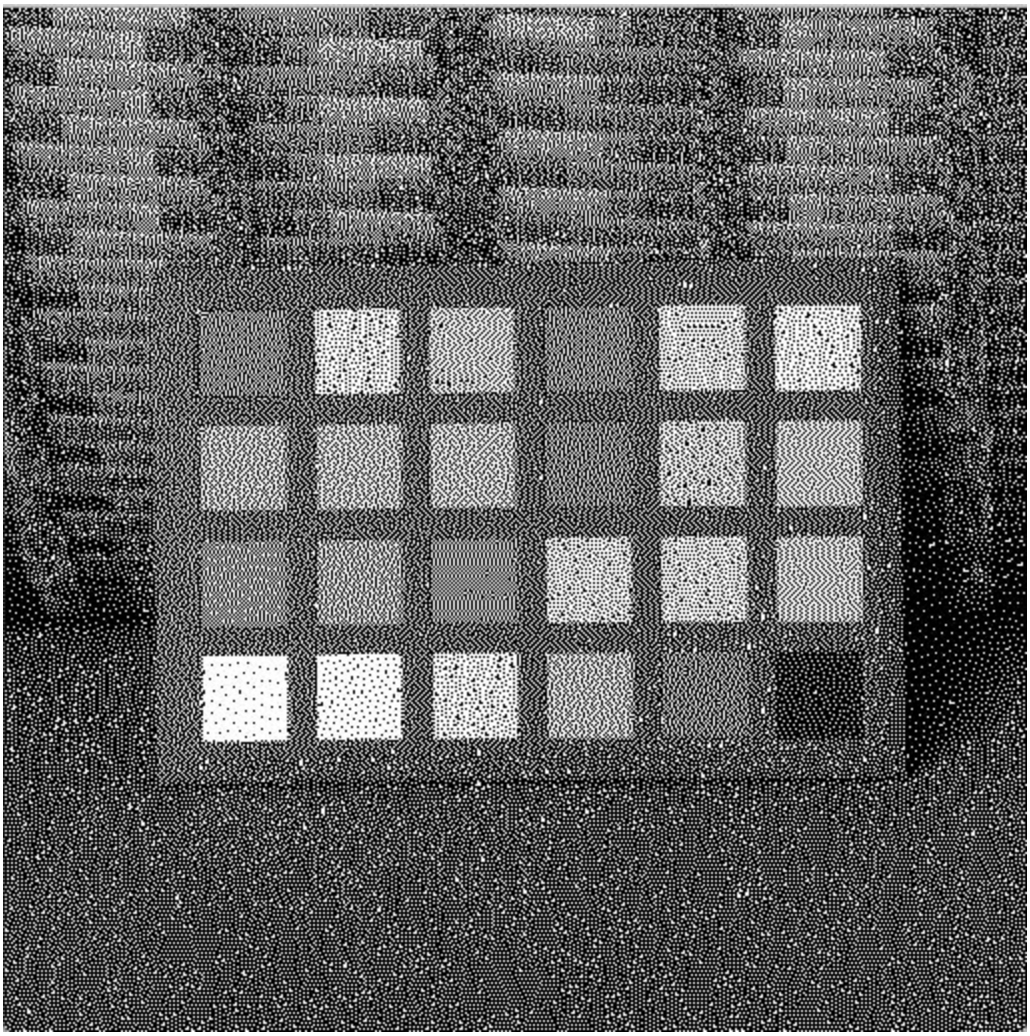
BAYER MATRIX SIZE = 8



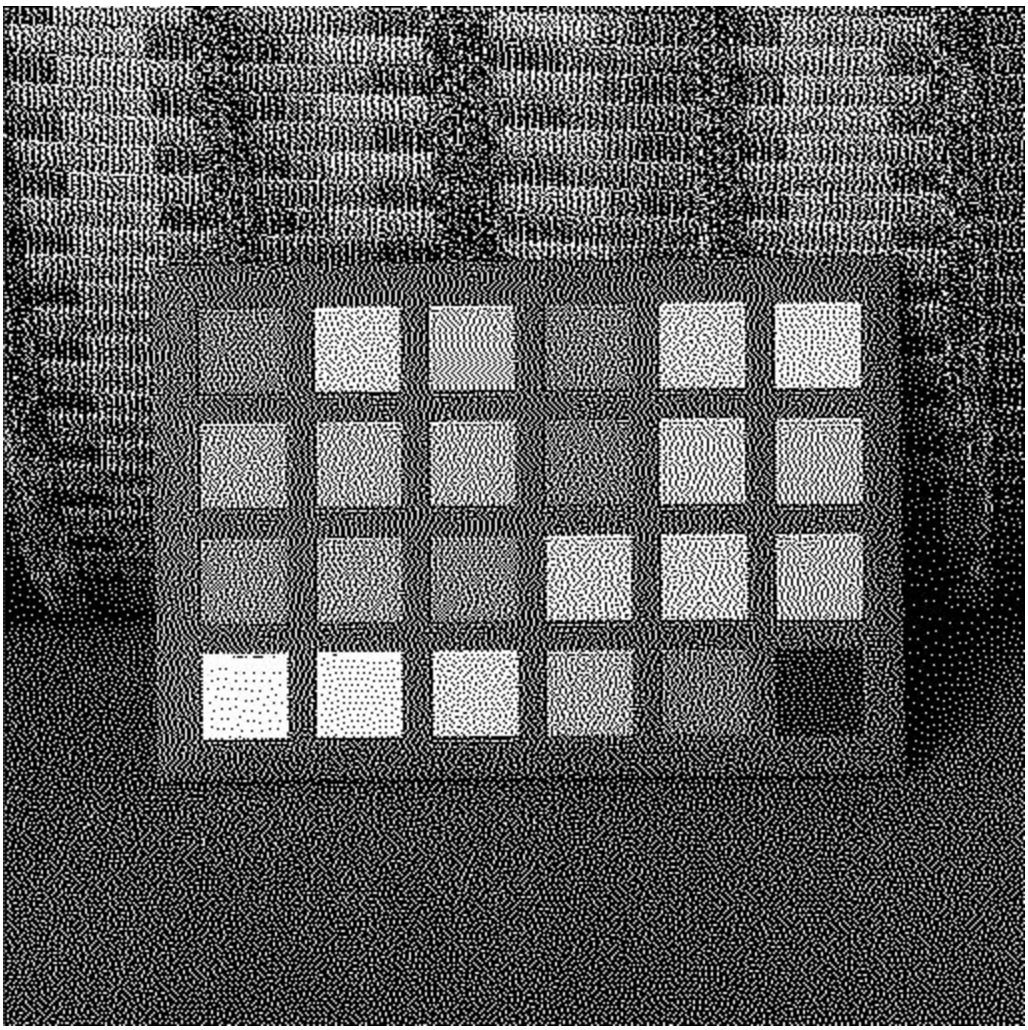
BAYER MATRIX SIZE = 2



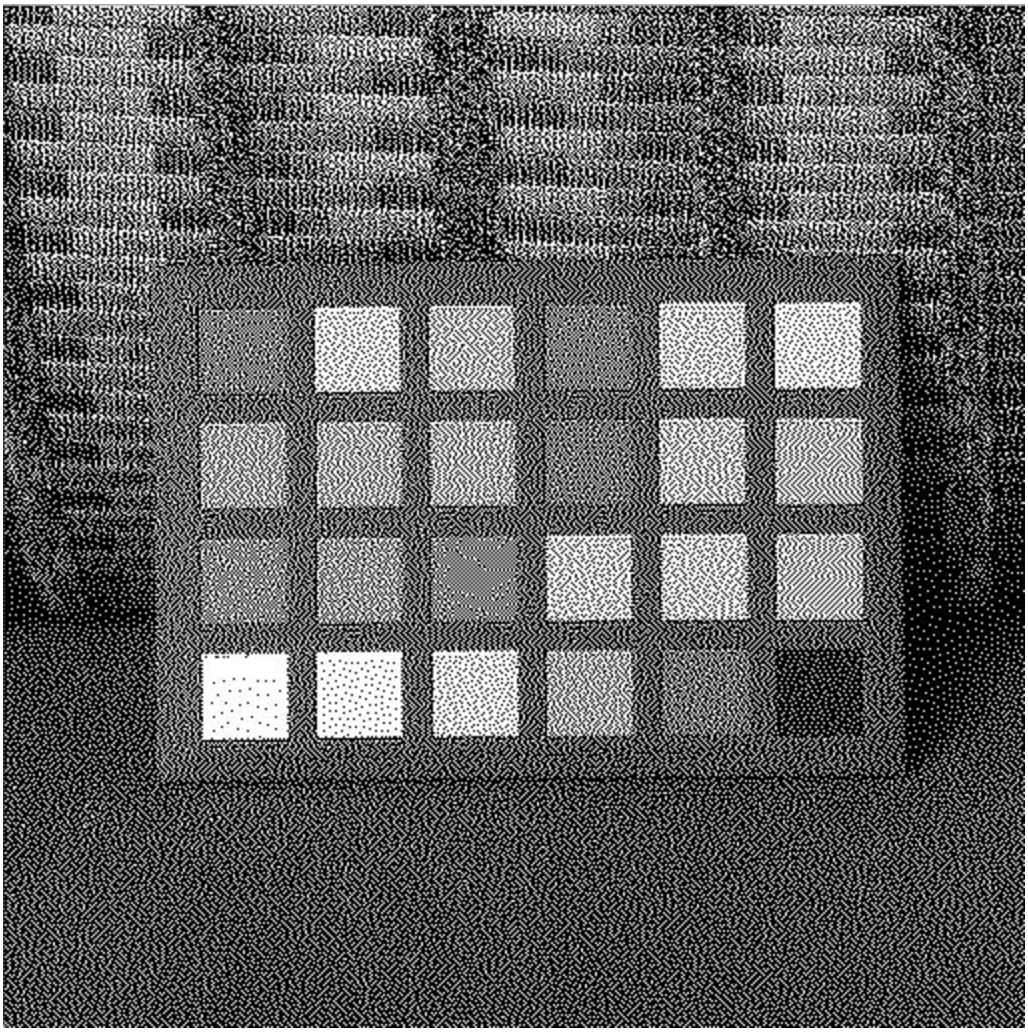
BAYER MATRIX SIZE = 4



FLOYD STEINBERG

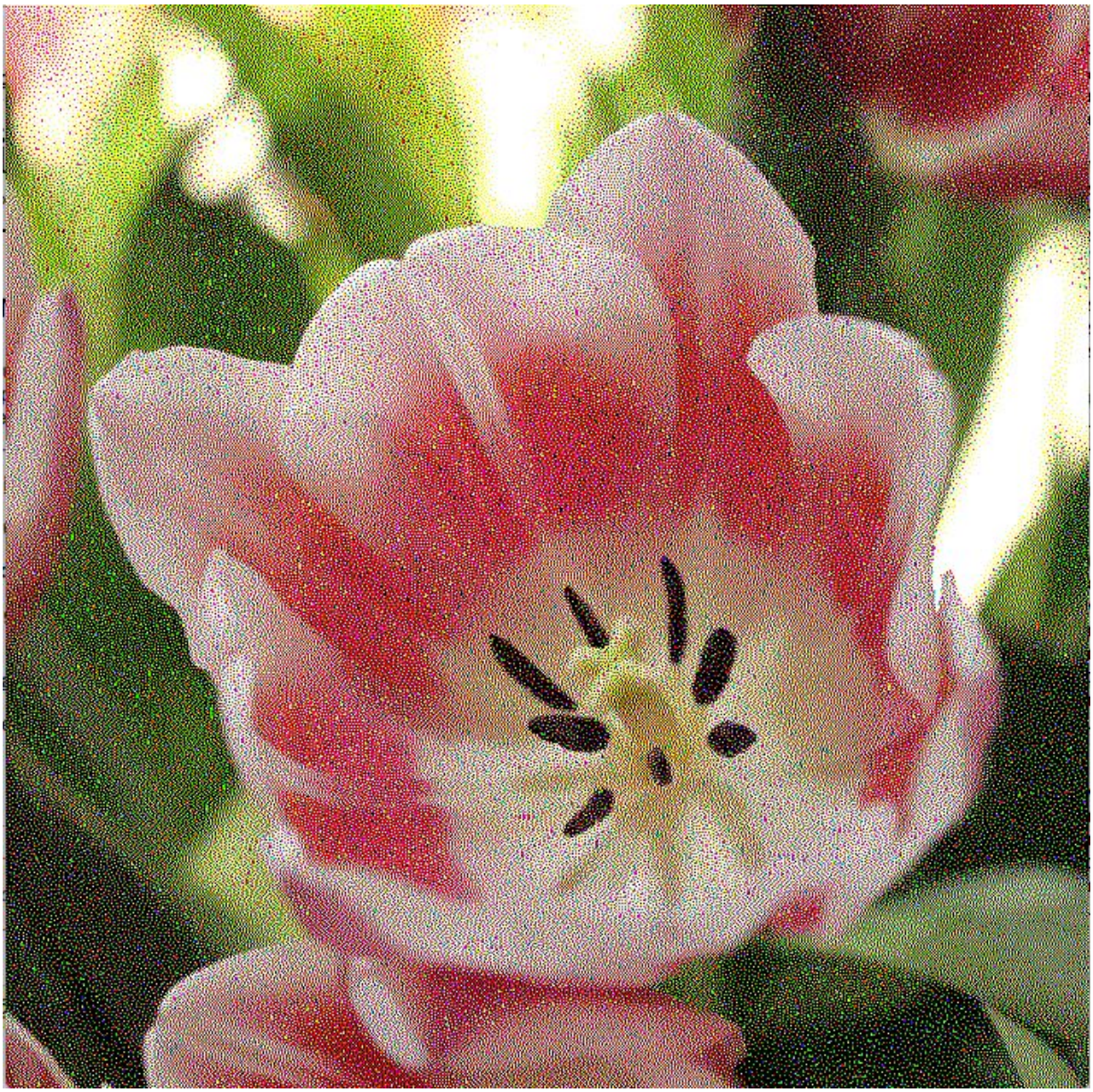


JJN



STUCKI

2.3.b.Color Halftoning-



SEPARABLE ERROR DIFFUSION

MBVQ

2.4. DISCUSSION :

From the results section above, we observe from the fixed thresholding that our perception of the gray levels is lost as the threshold is fixed at a constant of 127. This way we are losing out on a lot of details and contours in the image. The gradual variation in intensity is lost as the value of the threshold is fixed. A large amount of error noise is present.

In random thresholding, as we are generating random numbers in uniform distribution and as they are set as threshold, the output is comparatively better. This introduces a lot of "white noise". This method is useful in reproducing low frequency ranges. A gradient screen of varying gray levels would look better with a random dither compared to a fixed dither.

The Bayer ordered dither performs best when compared to the above two. As discussed in the approach section above, "A 0 indicates the first pixel to turn on, and correspondingly 3 indicates the last pixel to turn on. Bayer also introduced a series of recursive index matrices" we see that this method produces evenly distributed thresholds. But we can see the presence of square grid patterns in the dithered image.

Floyd-Steinberg error diffusion introduced the serpentine scanning of images and then diffusing the error to its future pixels and then updating the input image. We can see that the output is better than the Bayer Matrix. The downside is that Floyd-Steinberg diffusion method is Floyd-Steinberg uses narrow window. We can also see the presence of some salt and pepper noise in the image.

JJN error diffusion looks into more neighbors compared to the Floyd Steinberg method, hence it improves the error diffusion while in serpentine scanning. It is slower compared to the above methods as it need to perform a lot of computations for error diffusion at every pixel.

We can see that Stucki performs the best as it has taken out the residual pepper noise out of the image. The computation is also somewhat faster compared to JJN as it can be left shifted or right shifted as they are all even numbers.

Separable Color diffusion

As dithering intentionally adds noise and then removes it using error diffusion, we can see that in Separable color diffusion we see the presence of noise and other artifacts. We see the presence of square like grids in the image as well.

MBVQ performs better than separable error diffusion as there is elimination of square grids from the image. Also, as MBVQ follows a decision rule, and as the most minimum distance vector is found from the quadruple, this also ensures that minimum variance in the intensity levels is considered instead of randomly assigning the output pixels with new values as in separable error diffusion.

PROBLEM 3

3.1 MOTIVATION :

Morphological operations render the shape of the underlying object. They are a set of non-linear processing techniques which bring out the shape of the object. These operations are usually performed on binary images. In a binary image only a single bit is assigned to a pixel - 0(Black) or 1(White). The motivation behind converting an image into a binary image is that the processing is very fast. The output boolean values - 0 or 1 might indicate the presence of absence of certain features in the image. These values might also throw some light on the detection of an underlying object, edges, and smoothness of the image. As most devices use 8 bits to represent color, 0 is mapped to 0 and 1 is mapped to 255. The foreground pixel is the white and the background pixel is the black.
To convert a grayscale image to a binary image, thresholding at 127 (mid-value) is applied. Binary Morphological operations are usually applied on images which have exhibit bimodal histograms. These histograms signify that there maybe dark objects on a bright background or vice versa.
Morphological operations also consider a structuring element which traverses through the image and performs respective logical operations on the image underneath it.

3.2 APPROACH :

One of the most general operations performed on a binary image is the “hit and miss”. A structuring element containing both foreground and background pixels are placed over the image as it traverses through the whole image. If the structuring element pixels match the underlying pixels accurately, a value is written into a new binary image at the same location with its intensity as that of the foreground pixel. If it doesn’t match, it is set to the value of the background pixel.

To accurately find the structuring element and process the binary image, bond count is calculated for every pixel. There are two major types of pixel neighborhood connectivities - 4 connectivity and 8 connectivity. For 4-connectivity, if the center pixel has a neighbor either in its top, bottom, left or right, then the weight for the bond count is given to be 2. If the center pixel shares its connectivity with either its top diagonal left and right, or bottom diagonal left and right, then the weight for those corresponding pixels are valued at 1.

After the calculation of the bond connectivity, corresponding conditional masks are evoked for those particular operations. The conditional and unconditional masks are given in the table below

Table 14.3-1 Shrink, Thin and Skeletonize Conditional Mark Patterns (M=1 if hit)

Type	Bond	Patterns							
S	1	0 0 1	1 0 0	0 0 0	0 0 0				
		0 1 0	0 1 0	0 1 0	0 1 0				
		0 0 0	0 0 0	1 0 0	0 0 1				
S	2	0 0 0	0 1 0	0 0 0	0 0 0				
		0 1 1	0 1 0	1 1 0	0 1 0				
		0 0 0	0 0 0	0 0 0	0 1 0				
S	3	0 0 1	0 1 1	1 1 0	1 0 0	0 0 0	0 0 0	0 0 0	0 0 0
		0 1 1	0 1 0	0 1 0	1 1 0	1 1 0	0 1 0	0 1 0	0 1 1
		0 0 0	0 0 0	0 0 0	0 0 0	1 0 0	1 1 0	0 1 1	0 0 1
TK	4	0 1 0	0 1 0	0 0 0	0 0 0				
		0 1 1	1 1 0	1 1 0	0 1 1				
		0 0 0	0 0 0	0 1 0	0 1 0				
STK	4	0 0 1	1 1 1	1 0 0	0 0 0				
		0 1 1	0 1 0	1 1 0	0 1 0				
		0 0 1	0 0 0	1 0 0	1 1 1				
ST	5	1 1 0	0 1 0	0 1 1	0 0 1				
		0 1 1	0 1 1	1 1 0	0 1 1				
		0 0 0	0 0 1	0 0 0	0 1 0				
ST	5	0 1 1	1 1 0	0 0 0	0 0 0				
		0 1 1	1 1 0	1 1 0	0 1 1				
		0 0 0	0 0 0	1 1 0	0 1 1				
ST	6	1 1 0	0 1 1						
		0 1 1	1 1 0						
		0 0 1	1 0 0						
STK	6	1 1 1	0 1 1	1 1 1	1 1 0	1 0 0	0 0 0	0 0 0	0 0 1
		0 1 1	0 1 1	1 1 0	1 1 0	1 1 0	1 1 0	0 1 1	0 1 1
		0 0 0	0 0 1	0 0 0	1 0 0	1 1 0	1 1 1	1 1 1	0 1 1
STK	7	1 1 1	1 1 1	1 0 0	0 0 1				
		0 1 1	1 1 0	1 1 0	0 1 1				
		0 0 1	1 0 0	1 1 1	1 1 1				
STK	8	0 1 1	1 1 1	1 1 0	0 0 0				
		0 1 1	1 1 1	1 1 0	1 1 1				
		0 1 1	0 0 0	1 1 0	1 1 1				
STK	9	1 1 1	0 1 1	1 1 1	1 1 1	1 1 1	1 1 0	1 0 0	0 0 1
		0 1 1	0 1 1	1 1 1	1 1 1	1 1 0	1 1 0	1 1 1	1 1 1
		0 1 1	1 1 1	1 0 0	0 0 1	1 1 0	1 1 1	1 1 1	1 1 1
STK	10	1 1 1	1 1 1	1 1 1	1 0 1				
		0 1 1	1 1 1	1 1 0	1 1 1				
		1 1 1	1 0 1	1 1 1	1 1 1				
K	11	1 1 1	1 1 1	1 1 0	0 1 1				
		1 1 1	1 1 1	1 1 1	1 1 1				
		0 1 1	1 1 0	1 1 1	1 1 1				

Table 14.3-2 Shrink and Thin Unconditional Mark Patterns

Spur	0	0	M	M	0	0							
	0	M	0	0	M	0							
	0	0	0	0	0	0							
Single	0	0	0	0	0	0							
4-connection	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	
	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	0	M	M	M	M	0	0	M	0	0	0	M	
Offset	M	M	0	0	M	M	0	M	M	0	M	M	
	0	0	0	0	0	0	0	0	M	0	M	0	
Spur corner	0	A	M	M	B	0	0	0	M	M	0	0	
Cluster	0	M	B	A	M	0	A	M	0	0	M	B	
	M	0	0	0	0	M	M	B	0	0	A	M	
Corner	M	M	D										
Cluster	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	M	M	M	M	M	M	
	D	0	0	0	0	D	0	M	D	D	M	0	
	D	M	D	0	M	0	0	M	0	D	M	D	
	M	M	0	M	M	0	0	M	M	0	M	M	
	0	M	0	D	M	D	D	M	D	0	M	0	
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	D	M	0	0	M	D	D	0	M	M	0	D	
Branch	0	M	M	M	M	0	M	M	0	0	M	M	
	M	0	D	D	0	M	0	M	D	D	M	0	

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

Table 14.3-3 Skeletonize Unconditional Mark Patterns

Spur	0	0	0	0	0	0	0	0	M	M	0	0
	0	M	0	0	M	0	0	M	0	0	M	0
	0	0	M	M	0	0	0	0	0	0	0	0
Single	0	0	0	0	0	0	0	0	0	0	M	0
4-connection	0	M	0	0	M	M	M	M	0	0	M	0
	0	M	0	0	0	0	0	0	0	0	0	0
L Corner	0	M	0	0	M	0	0	0	0	0	0	0
	0	M	M	M	M	0	0	M	M	M	M	0
	0	0	0	0	0	0	0	M	0	0	M	0
Corner	M	M	D	D	D	D						
Cluster	M	M	D	D	M	M						
	D	D	D	D	M	M						
Tee Branch	D	M	D	D	M	D	D	D	D	D	M	D
	M	M	M	M	M	D	M	M	M	D	M	M
	D	D	D	D	M	D	D	M	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	D	M	0	0	M	D	D	0	M	M	0	D
Branch	0	M	M	M	M	0	M	M	0	0	M	M
	M	0	D	D	0	M	0	M	D	D	M	0

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

3.2.a Shrinking

Shrinking essentially erases all the background pixels in the image such that an object with no holes shrinks into a pixel and an object with holes shrinks into a ring surrounding it. There are two stages to perform shrinking - in the stage 1 using conditional masks for the respective pixel Bonds. This gives us an overall idea as to how many pixels can be erased without losing the connectivity of the pixel. The flowchart for the binary conditional operation is shown below. For the second stage considering the unconditional masks, the value at the current pixel in the output image is calculated from the neighborhood of the input image and the corresponding structuring element.
Logical representation of shrinking - $G(j, k) = X \cap [M \cup P(M, M0, \ldots, M7)]$,
where $P(M, M0, \ldots, M7)$ is an erasure inhibiting logical variable.

The operation is usually applied multiple times until we end up with a shrunk image.

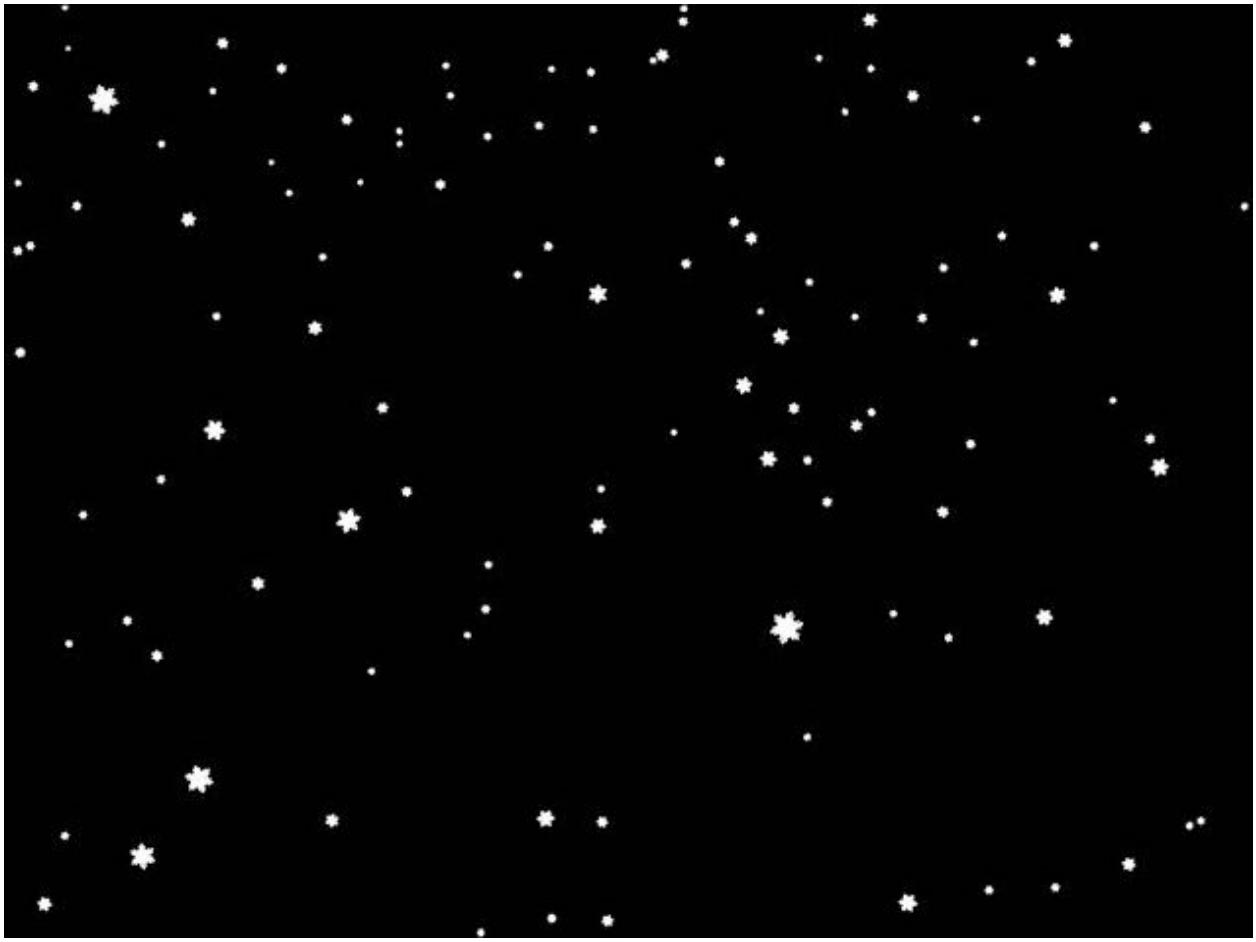
3.2.a.1.ALGORITHM :

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
6. Calculate the current pixel location using the row and the column indices.
7. If the normalized intensity value of the pixel is 1 at the location, then the bond count is calculated using the above procedure. If the pixel intensity is 0, the value in the output image is copied from the input at the same location.
8. After calculation of the bond count, it is matched with the corresponding conditional mask.
9. If the conditional mask matches the underlying neighborhood pixel values perfectly, then the value in the output image at the current location is set to the foreground value. Else it is set to the background value.
10. For the stage two of the process, the first stage output is passed as input and the original input is also passed as input.
11. If the center normalized pixel value in the first stage output is 0, then the intensity values from the original input image to copied to the second stage output image.
12. The neighborhood pixel intensities are compared with the unconditional masks, and if

there is a hit found then the thresholded binarized value from the original image is found. If the value is 1, then the output image at that location gets a binarized 1 else it gets a 0. Then the value is updated in the image.

13. Repeat steps 6-12 for a fixed number of iterations - For this problem, I have fixed the number of iterations to be 10.
14. Write the obtained images to a '.raw' file and check the output.
15. To calculate the total number of stars - at the end of shrinking iteration count the number of pixels which are 1 and have all 0s around them.
16. To calculate different star sizes - calculate at the end of each iteration how many center pixels have a 1 with all 0s around them and subtract them with the previous size count as the count is cumulative.

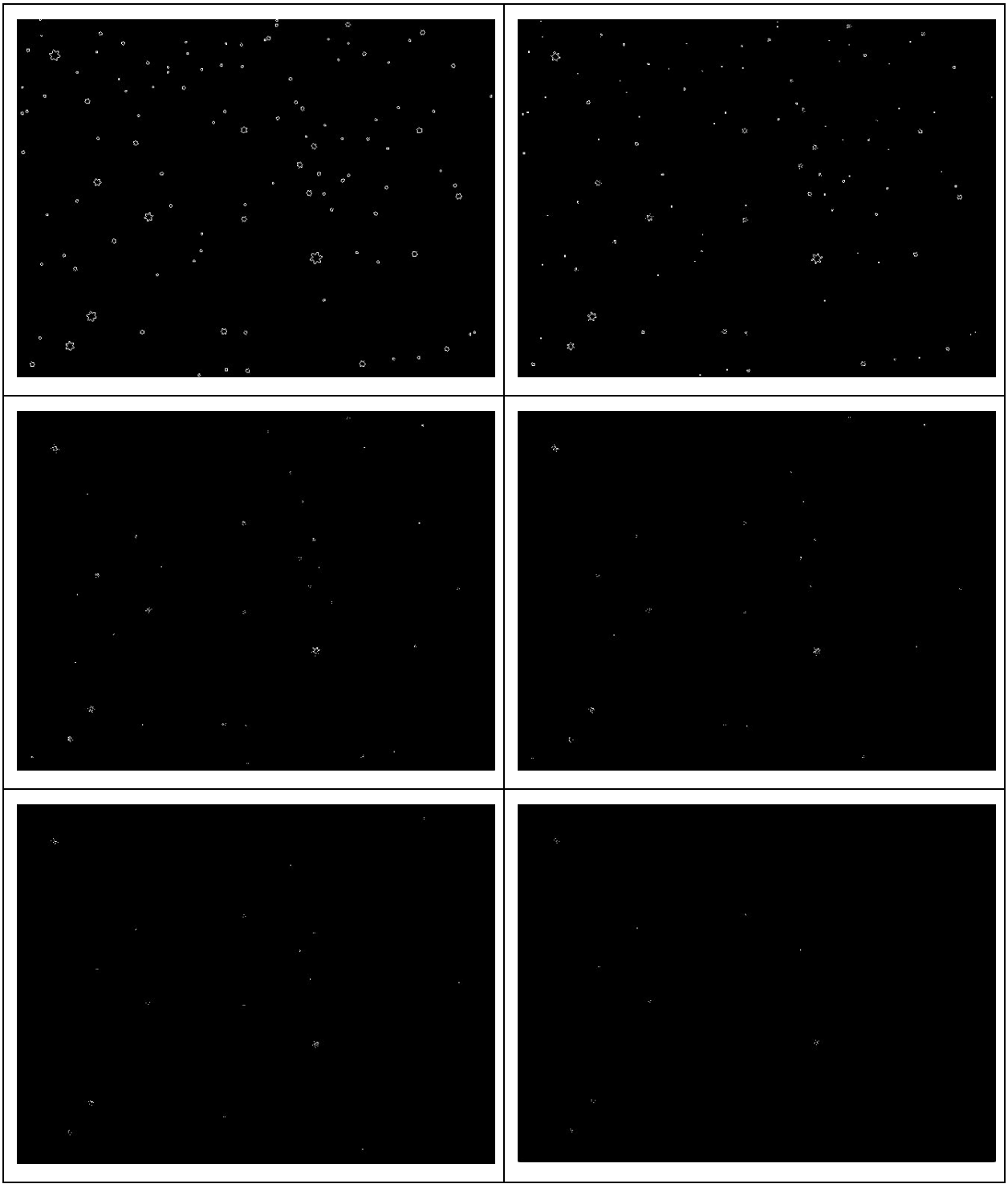
3.2.a RESULTS :



Input stars.raw

Stage 1 outputs -

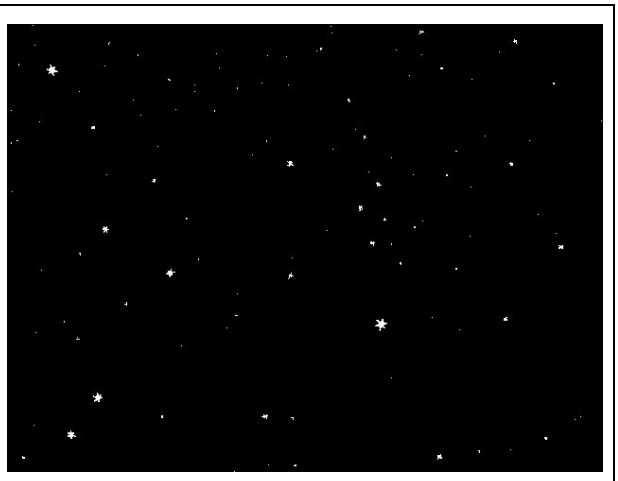
Number of iterations start at 10 goes to 0. Output is shown for every iteration.

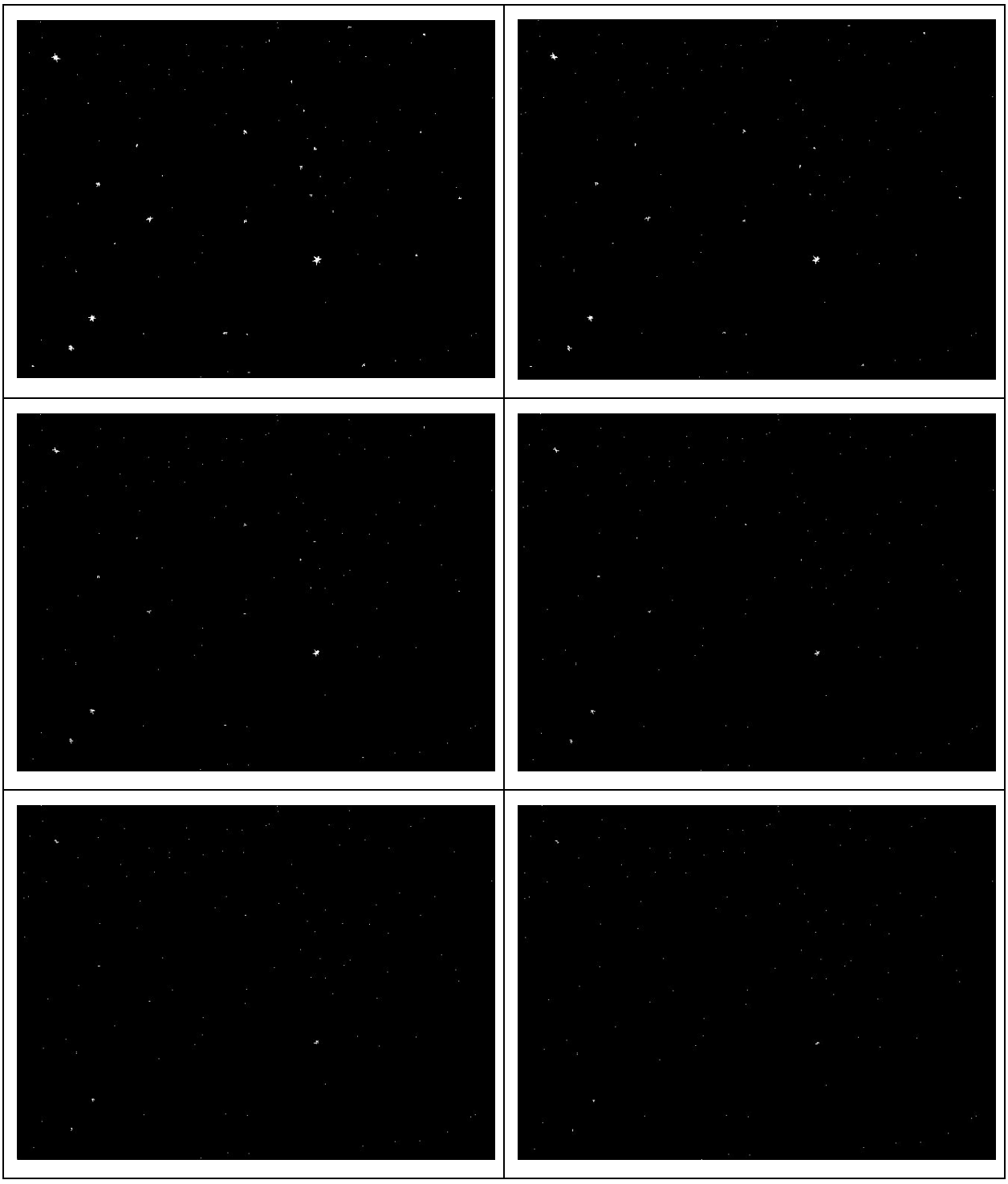


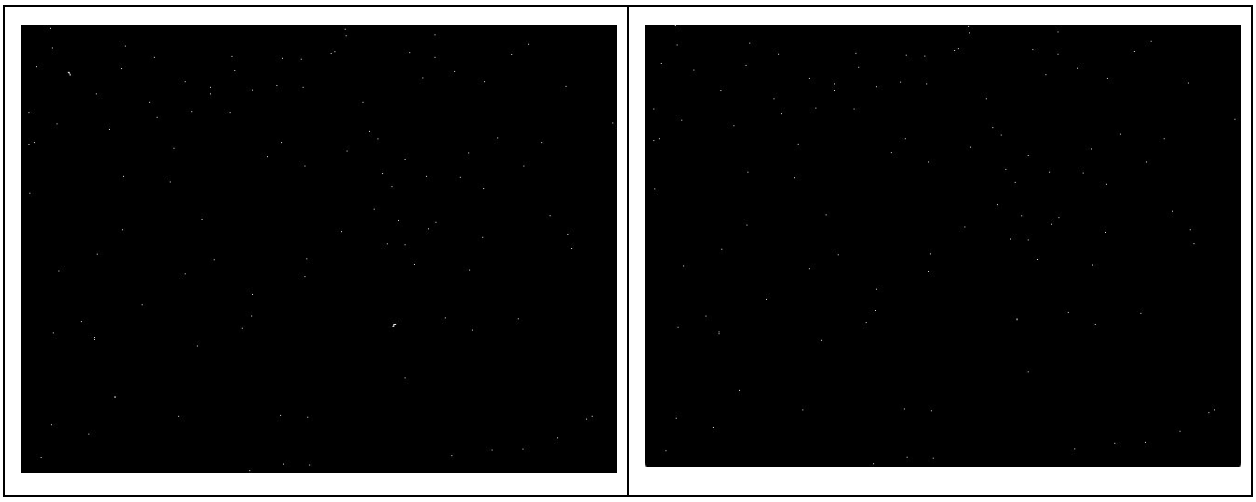



Stage 2 outputs -

Number of iterations start at 10 goes to 0. Output is shown for every iteration.





3.4.a DISCUSSION :

The number of stars in the image is found to be 113. It was found that one of stars split when the thresholding is greater than 127 but not equal to. Hence the total number of stars obtained was 113. Different sizes of stars for every iteration is shown with the count variable.

```
For iteration: 0
Count is: 9
For iteration: 1
Count is: 50
For iteration: 2
Count is: 19
For iteration: 3
Count is: 12
For iteration: 4
Count is: 6
For iteration: 5
Count is: 8
For iteration: 6
Count is: 2
For iteration: 7
Count is: 3
For iteration: 8
Count is: 1
For iteration: 9
Count is: 1
For iteration: 10
Count is: 1
For iteration: 11
Count is: 1
Total number of stars : 113

Process finished with exit code 0
```

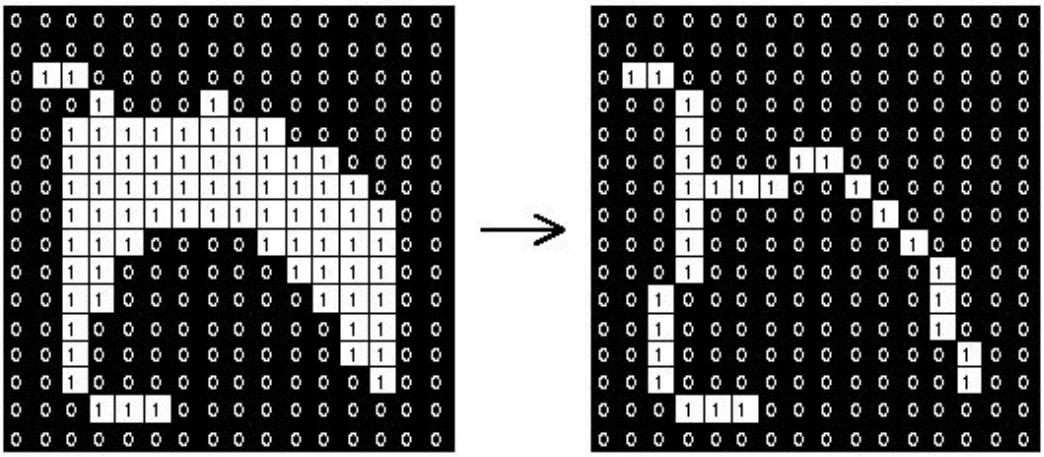
3.2.b Thinning

Thinning is a morphological operation which reduces the black pixels in an object in a way that an object in an image without any holes reduces to a line which is equidistant from its boundaries. If the object has a hole like structure, then it reduces to a connected ring halfway between the hole and its nearest boundary.

The technique is similar to the thinning operation. Thinning and shrinking share the same unconditional masks.

Thinning when applied to the background of an image can be used to distinguish between several objects in the same image. As thinning reduces all the pixels in an object to a line, we can clearly classify different objects.

The operation is usually applied multiple times until we end up with a thinned image.



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

3.2.b.1.ALGORITHM :

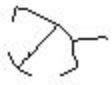
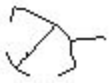
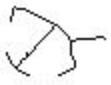
1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.

4. Read the input array as a single dimension array inside two nested ‘for’ loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. As the input has its object in background pixels the image needs to be inverted.
6. Initialize a set of ‘Image’ objects to be treated as outputs for the image processing operations.
7. Calculate the current pixel location using the row and the column indices.
8. If the normalized intensity value of the pixel is 1 at the location, then the bond count is calculated using the above procedure. If the pixel intensity is 0, the value in the output image is copied from the input at the same location.
9. After calculation of the bond count, it is matched with the corresponding conditional mask.
10. If the conditional mask matches the underlying neighborhood pixel values perfectly, then the value in the output image at the current location is set to the foreground value. Else it is set to the background value.
11. For the stage two of the process, the first stage output is passed as input and the original input is also passed as input.
12. If the center normalized pixel value in the first stage output is 0, then the intensity values from the original input image to copied to the second stage output image.
13. values from the original input image to copied to the second stage output image.
14. The neighborhood pixel intensities are compared with the unconditional masks, and if there is a hit found then the thresholded binarized value from the original image is found. If the value is 1, then the output image at that location gets a binarized 1 else it gets a 0. Then the value is updated in the image.
15. Repeat steps 6-12 for a fixed number of iterations - For this problem, I have fixed the number of iterations to be 10.
16. Write the obtained images to a ‘.raw’ file and check the output.

3.3.b RESULTS :

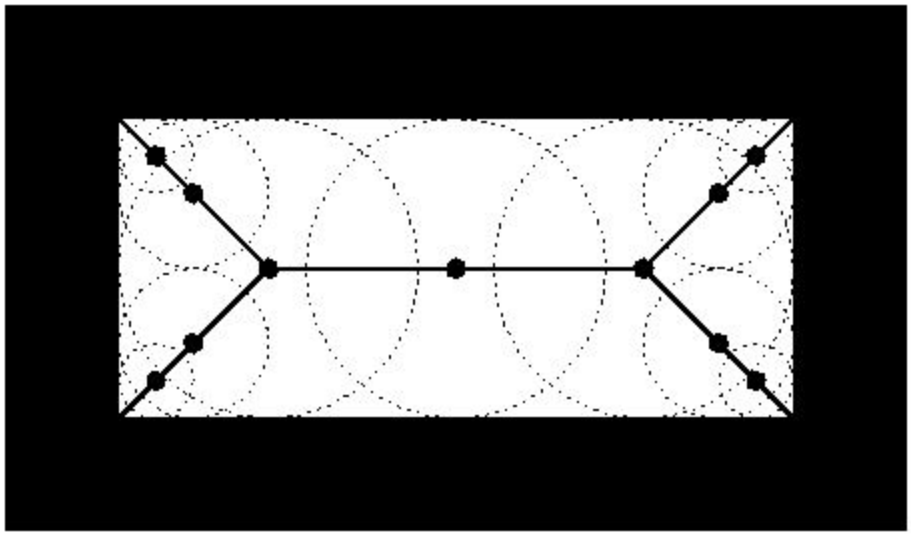
N = starting iteration index

 Original	 N = 35

 N = 25	 N = 15
 N = 5	 N = 1

3.1.c Skeletonizing

Skeletonizing produces a blueprint of the underlying structure. This technique is very useful to find cracks or not connected components in printed circuit boards.
This technique reduces foreground pixels to its residual skeleton. To understand this process better, imagine the the white pixels in the binary image are made of a uniform flammable material. It uniformly burns from all the four corners and then extinguishes when it meets. This gives us an idea as to how skeletonization works.
Skeletonization uses Medial Axis transform to find closest boundary point in each object.

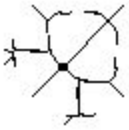
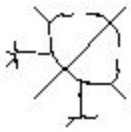
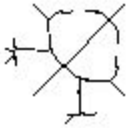
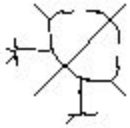


Source : https://homepages.inf.ed.ac.uk/rbf/HIPR2/skeleton.htm

3.1.c.1.ALGORITHM :

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. As the input has its object in background pixels the image needs to be inverted.
6. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
7. Calculate the current pixel location using the row and the column indices.
8. If the normalized intensity value of the pixel is 1 at the location, then the bond count is calculated using the above procedure. If the pixel intensity is 0, the value in the output image is copied from the input at the same location.
9. After calculation of the bond count, it is matched with the corresponding conditional mask.
10. If the conditional mask matches the underlying neighborhood pixel values perfectly, then the value in the output image at the current location is set to the foreground value. Else it is set to the background value.
11. For the stage two of the process, the first stage output is passed as input and the original input is also passed as input.
12. If the center normalized pixel value in the first stage output is 0, then the intensity values from the original input image to copied to the second stage output image.
13. values from the original input image to copied to the second stage output image.
14. The neighborhood pixel intensities are compared with the unconditional masks, and if there is a hit found then the thresholded binarized value from the original image is found. If the value is 1, then the output image at that location gets a binarized 1 else it gets a 0. Then the value is updated in the image.
15. Repeat steps 6-12 for a fixed number of iterations - For this problem, I have fixed the number of iterations to be 35.
16. Write the obtained images to a '.raw' file and check the output.

3.3.c RESULTS :

Original 	 N = 35
 N = 25	 N = 15
 N = 5	 N = 1

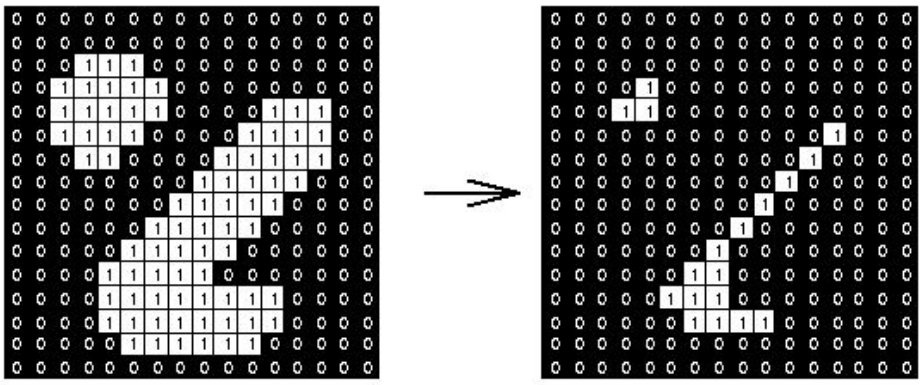
3.2.a Counting game

Morphological operations when performed on an image give away a lot of information about the underlying objects and shapes in the binary image. Hence, these processes can be used to differentiate between different objects in the image and also find the number of unique objects in the image.

The algorithm adapted for finding the number of jigsaws in the binary image is -

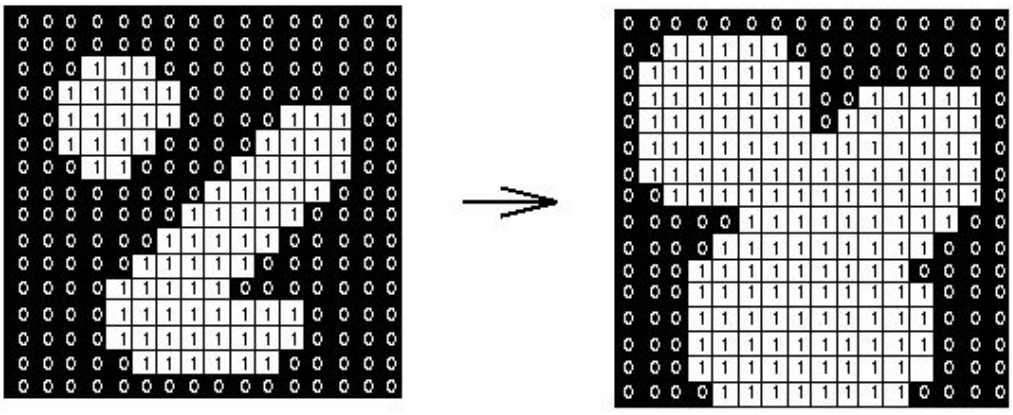
- Erosion using a 11x11 structuring element
- Dilation using a 27x27 structuring element
- Apply shrinking 35 times

EROSION : The structuring element used here is a 11x11 mask of all foreground pixels. As erosion reduces, the number of foreground pixels to one representative pixel in that object, it is a logical AND operation. If every pixel in the structuring element matches exactly with the underlying foreground pixels, then the intensity value at the same location in the output image is set to the foreground value, else it is set to the background value.



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

DILATION : Dilation is the process of expanding the boundary of white pixels in the image. This in turn reduces the size of the holes. The structuring element chosen here is a 27x27 mask of all ones. This process implements a logical OR. If any of the pixels in the underlying image matches with the 27x27 structuring element, then the value at the center pixel in the output image is set to the foreground value. Thus this process enlarges the image.



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

SHRINKING : Shrinking is described in the section (). The same algorithm is applied here.

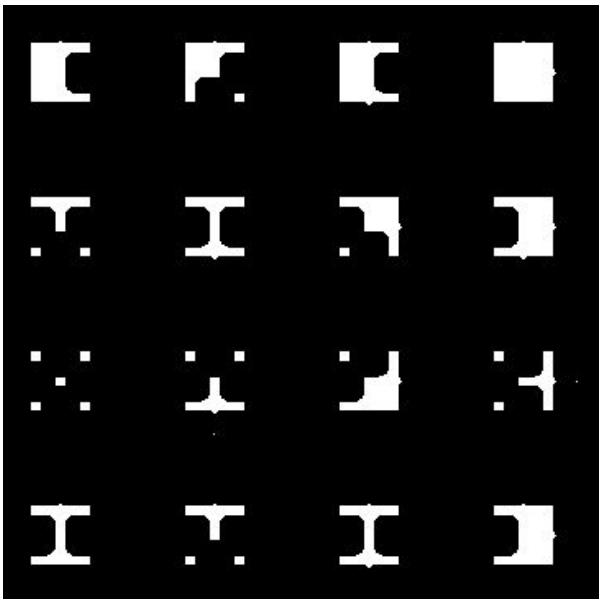
3.4.c.1.ALGORITHM :

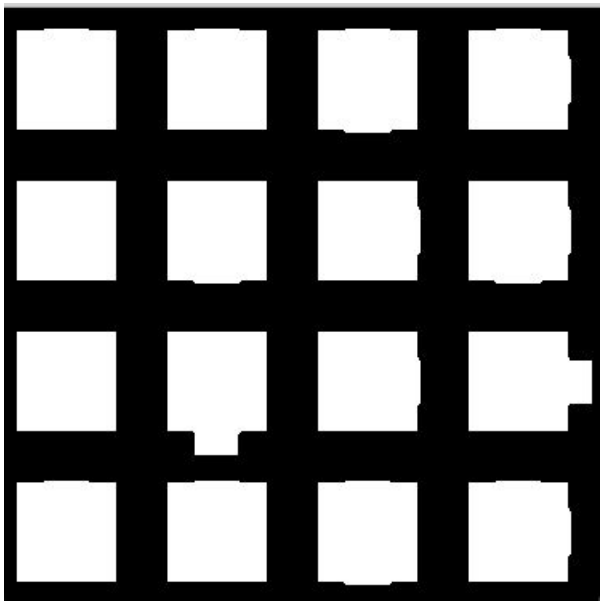
1. Load the input raw image using the function 'load_image_from_file' function.

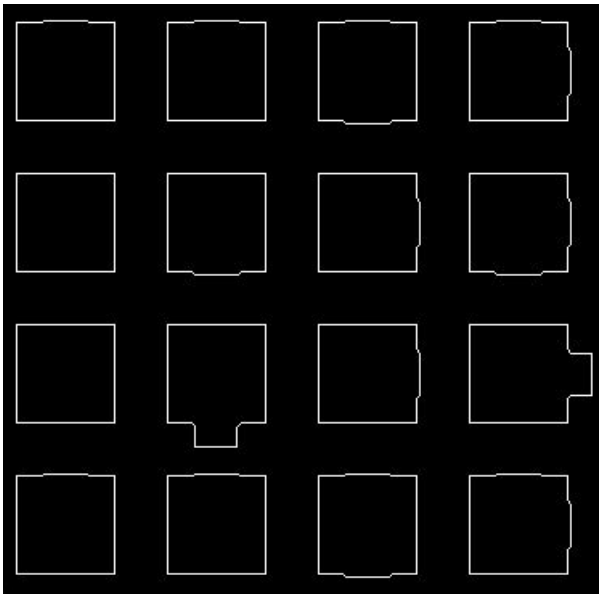
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. As the input has its object in background pixels the image needs to be inverted.
6. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
7. Calculate the current pixel location using the row and the column indices.
8. If the normalized intensity value of the pixel is 1 at the location, then the bond count is calculated using the above procedure. If the pixel intensity is 0, the value in the output image is copied from the input at the same location.
9. After calculation of the bond count, it is matched with the corresponding conditional mask.
10. If the conditional mask matches the underlying neighborhood pixel values perfectly, then the value in the output image at the current location is set to the foreground value. Else it is set to the background value.
11. For the stage two of the process, the first stage output is passed as input and the original input is also passed as input.
12. If the center normalized pixel value in the first stage output is 0, then the intensity values from the original input image to copied to the second stage output image.
13. values from the original input image to copied to the second stage output image.
14. The neighborhood pixel intensities are compared with the unconditional masks, and if there is a hit found then the thresholded binarized value from the original image is found. If the value is 1, then the output image at that location gets a binarized 1 else it gets a 0. Then the value is updated in the image.
15. Repeat steps 6-12 for erosion, dilation and shrinking. Repeat steps 6-12 for a fixed number of iterations for shrinking - For this problem, I have fixed the number of iterations to be 35.
16. Write the obtained images to a '.raw' file and check the output.

3.4.b RESULTS : Counting game

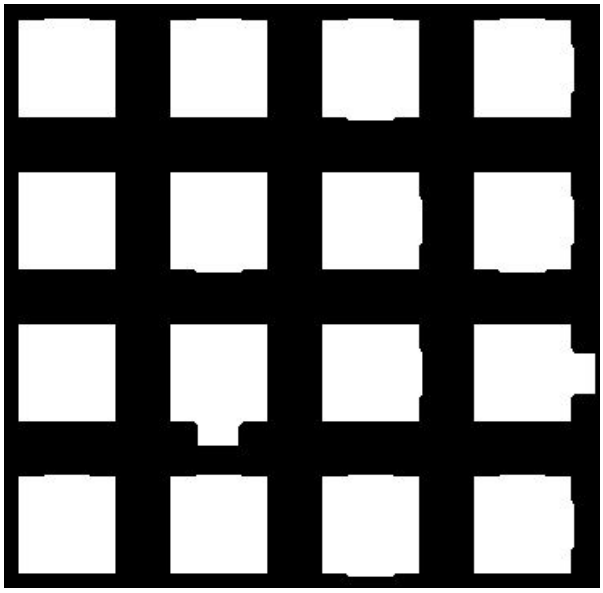
Part 1 -



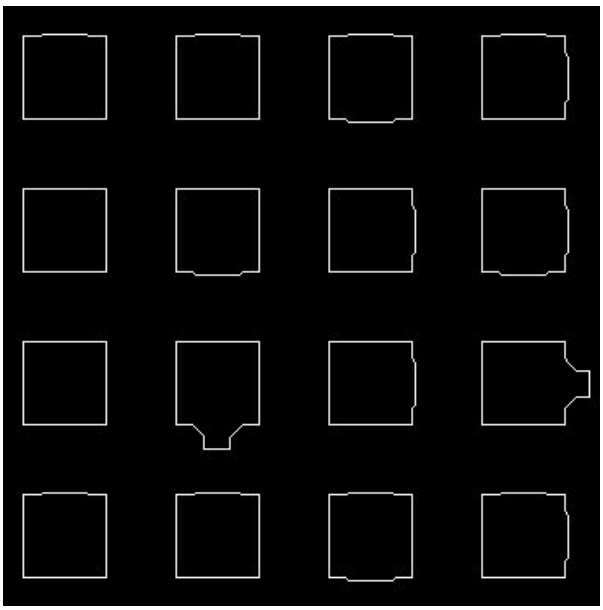




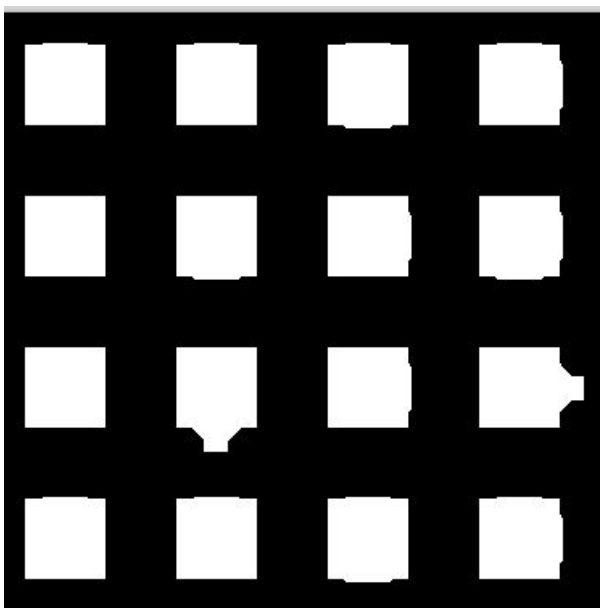
N = 35



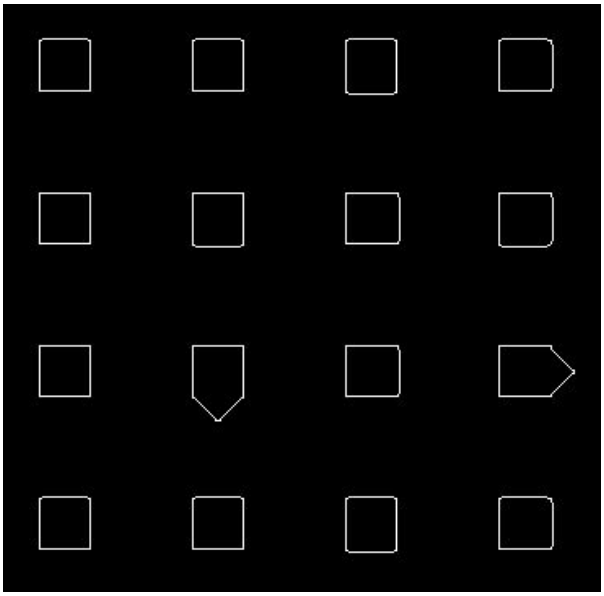
N = 35



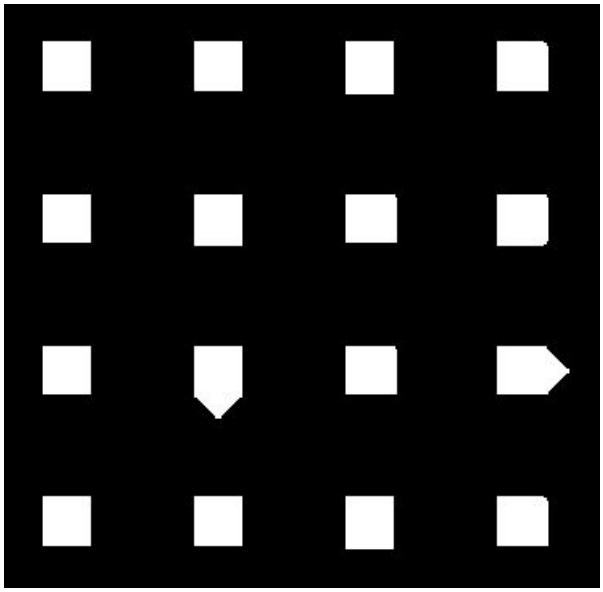
N = 25



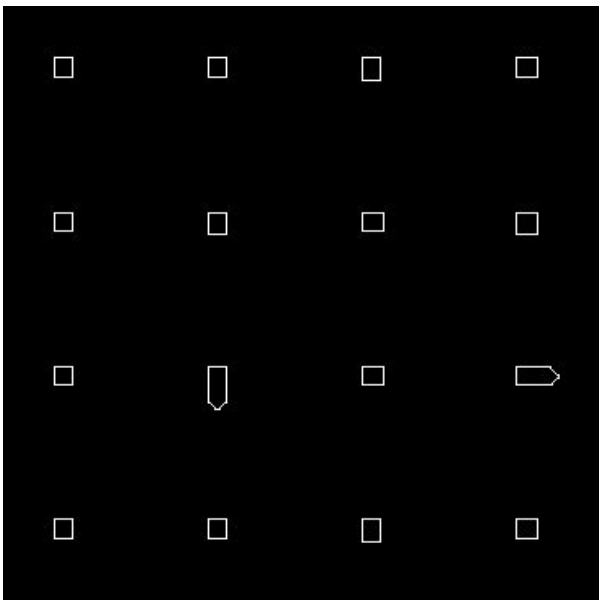
N = 25



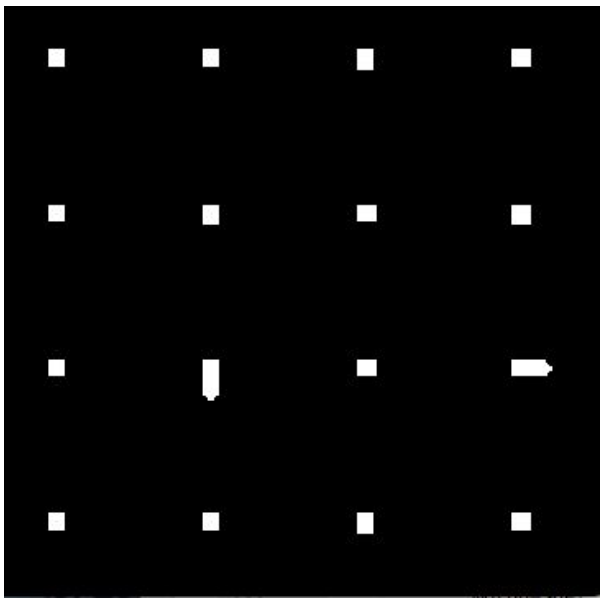
N = 15



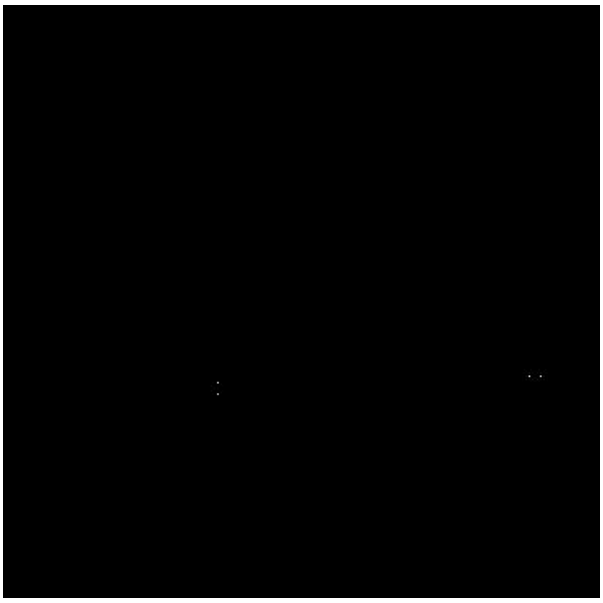
N = 15



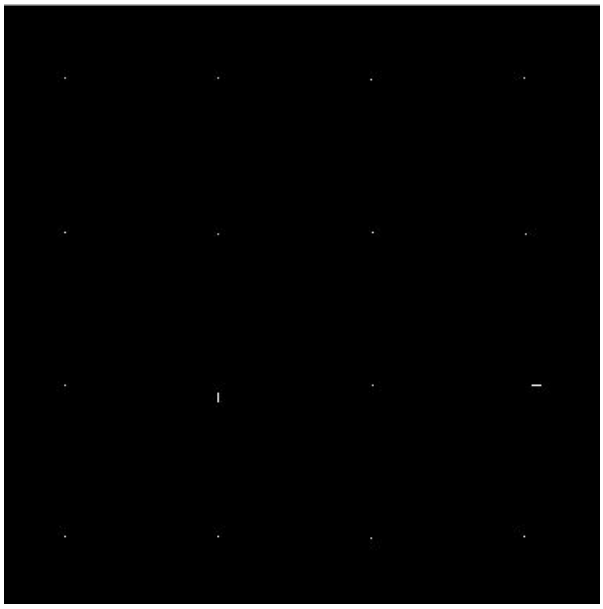
N = 10



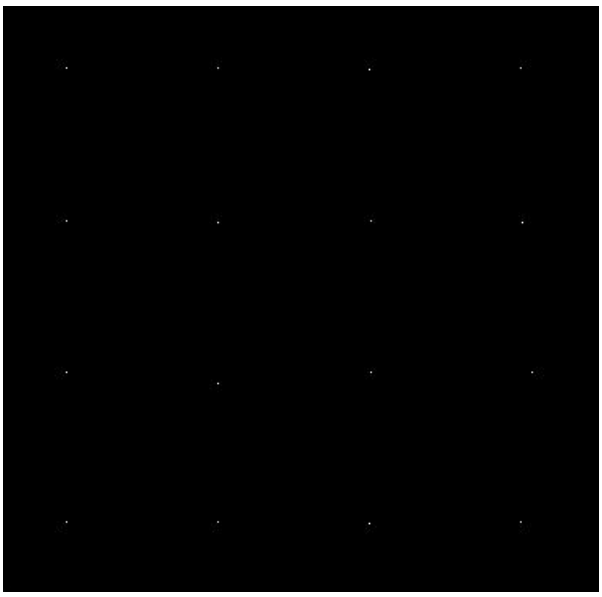
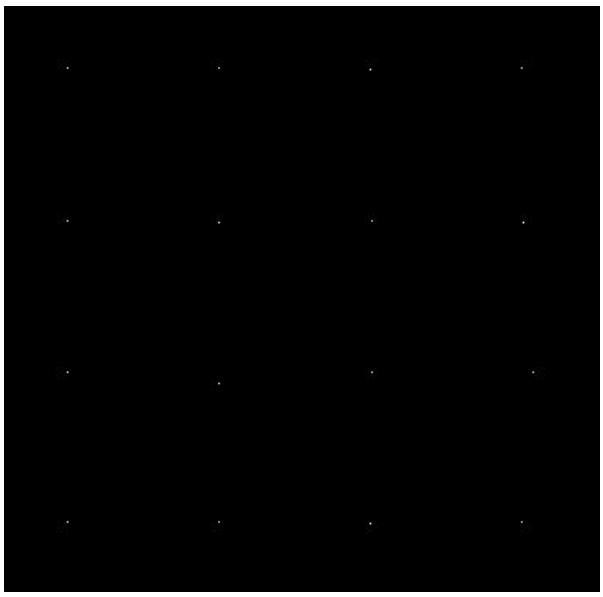
N = 10



N = 5



N = 5

	
N = 1	N = 1

COUNT UNIQUE NUMBER OF JIGSAWS :

As the problem statement is to find unique number of jigsaws, I chose to implement the concept of connected components labelling to make my program identify different objects in my image and label them accordingly.

After labelling them, my program found the width and height of the jigsaw barring the protrusions and holes. As it was found to be of equal width and height, I initialized a structuring element of all ones of the square size.

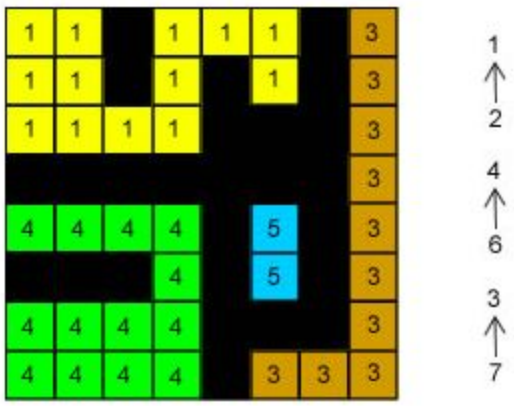
Next, my program the middle point of every jigsaw and placed the structuring element on top of it. I XOR'd the structuring element with the underlying pixels and thus this operation resulted in the display of holes and protrusions in the image.

Next, my program found the distance of a hole or a protrusion from the middle point and the holes and protrusion count array was saved. Later, it was rotated and reflected to see if it matches to any other jigsaw pattern in the image. If it didn't, it was considered to be unique.

CONNECTED COMPONENTS LABELLING :

Assuming 8 connectivity, we scan the image for the neighboring pixels. If the intensity value of the pixel at that location is 1 (255), then it inspects the neighbors and determines its labels. Finally it gives the label of the most frequent label in its neighborhood. If there is a ties for the most frequent neighborhood label, then the minimum value of the label is assigned. If the neighborhood pixels are all 0s, then a new label is initialized for the current pixel. This is the first pass of the algorithm. For the second pass, equivalent label pairs are generated and sorted into their equivalence classes and a unique label is assigned to each class. In the second pass, the equivalence class takes over the predefined label sets.

1	1	0	1	1	1	0	1
1	1	0	1	0	1	0	1
1	1	1	1	0	0	0	1
0	0	0	0	0	0	0	1
1	1	1	1	0	1	0	1
0	0	0	1	0	1	0	1
1	1	1	1	0	0	0	1
1	1	1	1	0	1	1	1



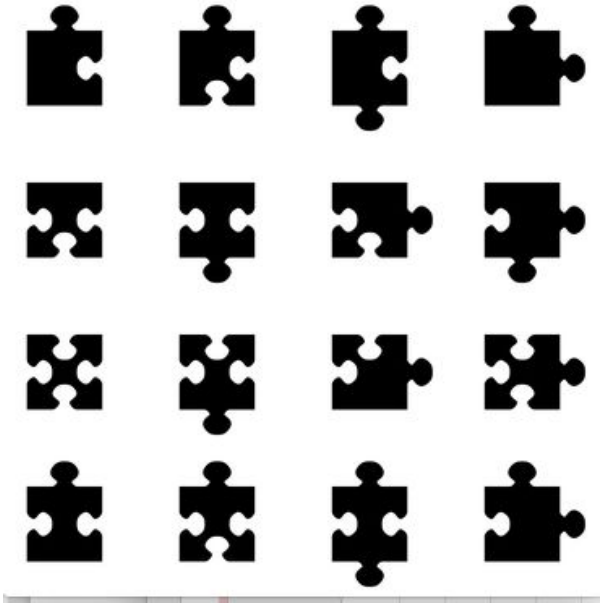
The final result

Source : http://aishack.in/tutorials/labelling-connected-components-example/

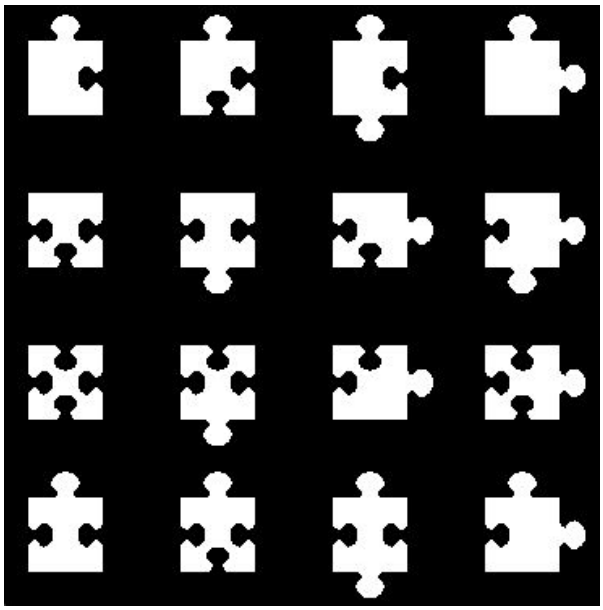
ALGORITHM :

1. Load the input raw image using the function 'load_image_from_file' function.
2. Initialize the 'Image' class variables with respective input image parameters i.e, width, height, and number of channels.
3. Allocate memory for the input image.
4. Read the input array as a single dimension array inside two nested 'for' loops iterating over the rows(till the height) and columns(till the width) of the input image.
5. As the input has its object in background pixels the image needs to be inverted.
6. Initialize a set of 'Image' objects to be treated as outputs for the image processing operations.
7. Calculate the current pixel location using the row and the column indices.
8. Initialize a 2D label array of all zeros, call the compare the neighbor function recursively to find the neighboring labels of that pixel and initialize the most frequent label or the minimum label or a new label value to the pixel. As this happens recursively, comparison of all the values in the neighborhood is exhausted until a value is fixed to the current pixel.
9. It was observed that there was 16 labels (from 1 to 16). In the labelled image, find the least column for every jigsaw (as there are no protrusions to the left in the given image) and then find the center pixel for every jigsaw.
10. Initialize an XOR array of the same size of the jigsaw and XOR with the underlying pixels. After XOR operation, it only leaves the holes and the protrusions untouched.
11. After obtaining the holes and the protrusions, traverse to the center pixel and find the holes and the protrusion for that jigsaw.
12. If we hit a 1 within the size of the jigsaw square, we consider it to be a hole. If we hit a 1 after the size of jigsaw square, we consider it as a protrusion.
13. The holes and protrusions for every jigsaw are saved in an array.
14. They are rotated and reflected and compared to every other jigsaw pattern to see if there is a match. If there is a match, a counter is incremented. Else it is unique
15. It was found that there are 10 unique elements in the image.

Part 2 -



Input image



Labelled image

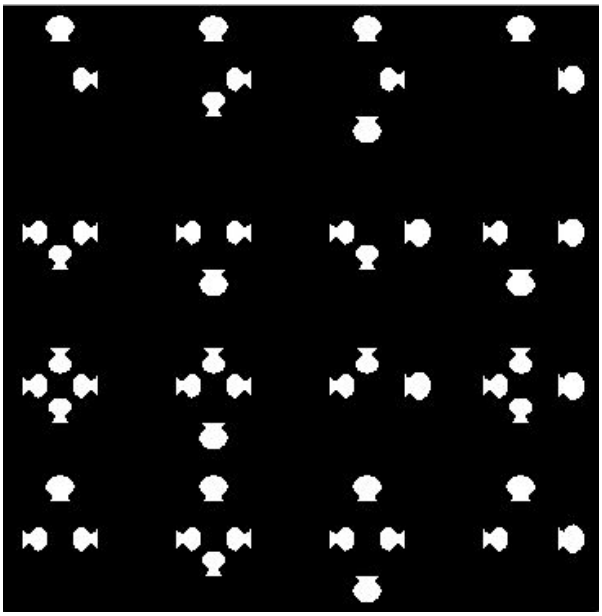
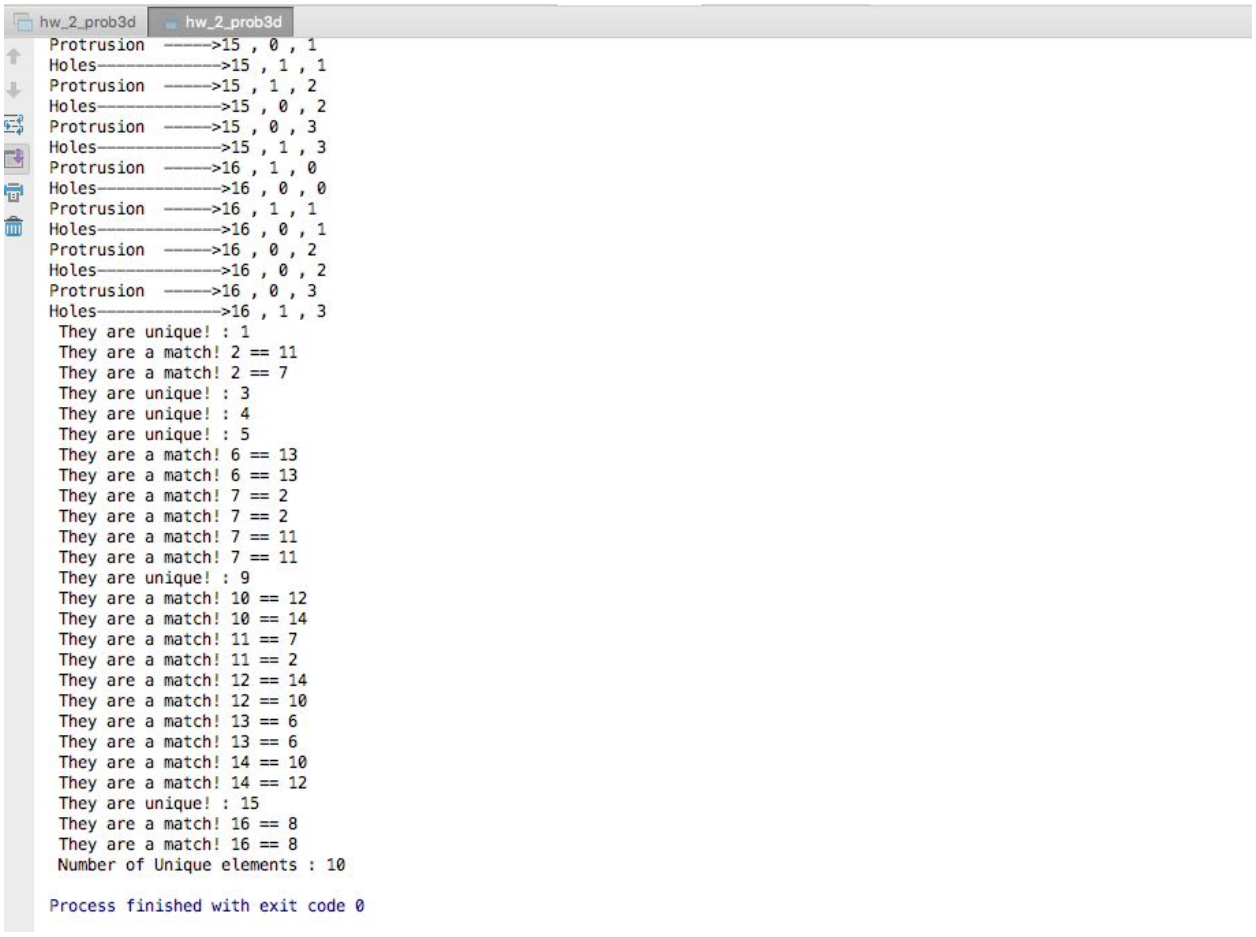


Image showing holes and protrusions of jigsaws

3.4.DISCUSSION :



The number of unique jigsaws were found to be 10. The matching jigsaws are shown in the image.
By implementing multiple morphological operations, we could determine the size of the jigsaw, initial point of the image, after performing XOR operations, we could identify the number of holes and protrusions in every jigsaw, and then rotate and compare with other jigsaws.

This problem gave me insights into how morphological operations can be used to determine and understand underlying object characteristics.

References :

1 . William . K . Pratt - Digital Image Processing.

2. D. Shaked, N. Arad, A. Fitzhugh, I. Sobel, "Color Diffusion: Error-Diffusion for Color Halftones",HP Labs Technical Report, HPL-96-128R1, 1996.