EEE 569 Digital Image Processing
Homework #2

Abinaya Manimaran
manimara@usc.edu

Spring 2018

Submitted on 03/04/2018

PROBLEM 1
GEOMETRIC IMAGE MODIFICATION

A) GEOMETRICAL WARPING:

➢ ABSTRACT AND MOTIVATION:

Geometric transformation is a method in Image Processing, in which image pixel locations are modified and not the actual intensity levels. There are various applications of Geometrical Image Warping. Some of them are:

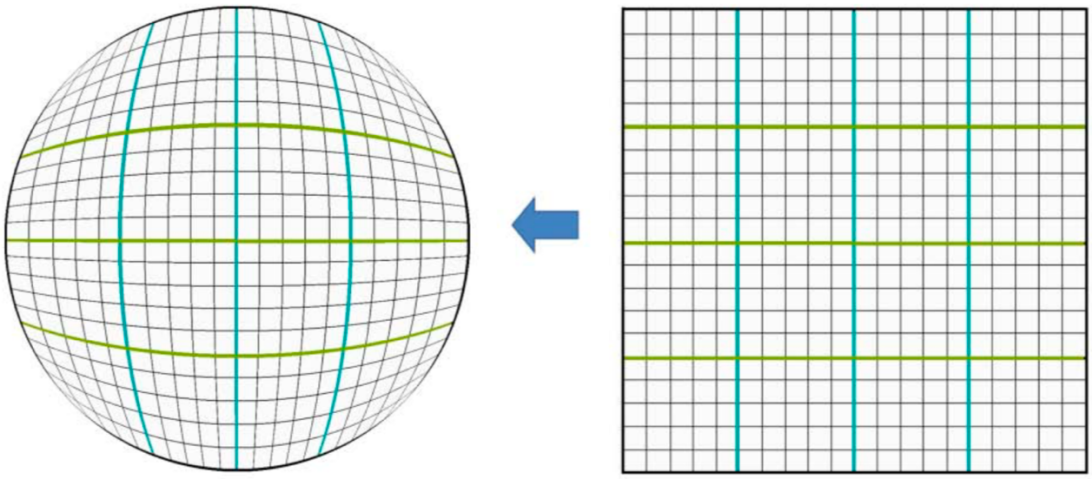
- To create special effects in an image
- To register/combine two images taken at the same scene at different times
- To morph one image into another

Warping is a process that describes the destination (x,y) using the source (u,v). In this portion of the question, I tried to create a disk-shaped special effect.

➢ APPROACH AND PROCEDURES:

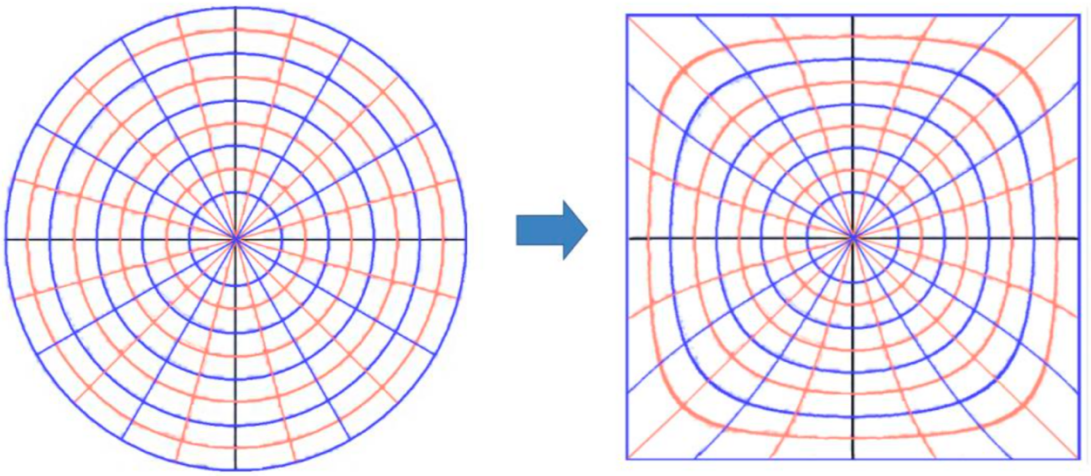
Geometrical Warping given in the question is warping of the square image to a disk-shaped image. The mid points in vertical and horizontal sides of the image remain at the same location. But the corner points are mapped to a different location. This kind of Geometrical Image Modification is given by Elliptical Warping.

(Source: https://arxiv.org/pdf/1509.06344.pdf)



Square to disc mapping:

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

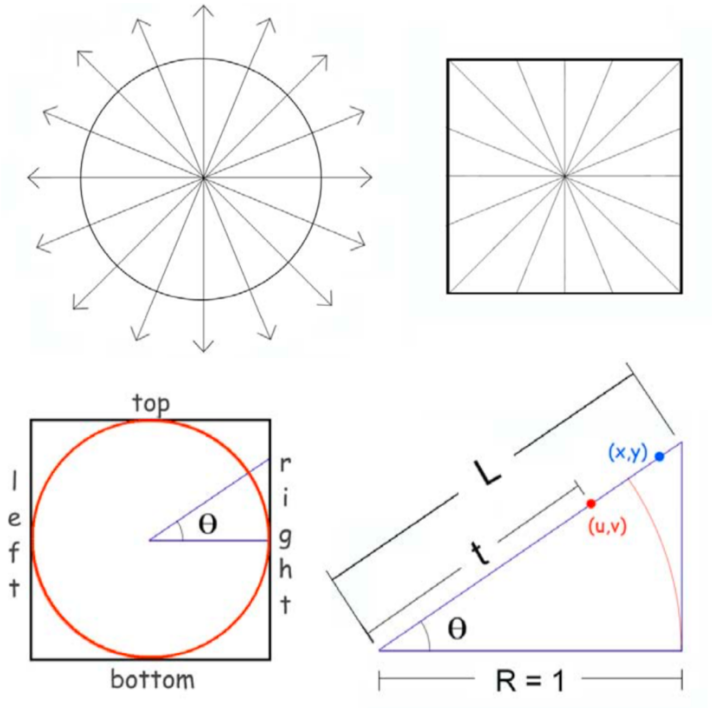


Disc to square mapping:

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

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

This Elliptical Mapping is a special method in Radial mapping. The derivation starts from the below given initial procedure:



The derivation of Elliptical Image Warping is given below:
(Source: https://arxiv.org/pdf/1509.06344.pdf)

We now derive an inverse of Nowell's Elliptical Grid mapping. The derivation involves 3 steps which is summarized as

Step 1: Convert circular coordinates (u,v) to polar coordinates and introduce trigonometric variables α and β
Step 2: Find an expression for (x,y) in terms of α and β
Step 3: Find an expression for (x,y) in terms of u and v

Start with Nowell's equations:

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

Step 1: Convert circle coordinates (u,v) *to polar form* (r, θ)

$$r = \sqrt{u^2+v^2}$$

$$\tan\theta = \frac{v}{u} \qquad \sin\theta = \frac{v}{\sqrt{u^2+v^2}} \qquad \cos\theta = \frac{u}{\sqrt{u^2+v^2}}$$

We introduce intermediate trigonometric angles α and β such that

$$\cos\alpha = r\ \cos(\theta+\frac{\pi}{4})$$
$$\cos\beta = r\ \cos(\theta-\frac{\pi}{4})$$

Note that, since r≤ 1 and the cosine value is ≤ 1, there exist angles α and β that satisfies this. We can then expand these expressions using trigonometric expansion formulas.

$$\cos\alpha = r\ \cos(\theta+\frac{\pi}{4}) = r\left(\cos\theta\cos\frac{\pi}{4}-\sin\theta\sin\frac{\pi}{4}\right) = r\frac{\sqrt{2}}{2}(\cos\theta-\sin\theta)$$

Likewise,

$$\cos\beta = r\ \cos(\theta-\frac{\pi}{4}) = r\left(\cos\theta\cos\frac{\pi}{4}+\sin\theta\sin\frac{\pi}{4}\right) = r\frac{\sqrt{2}}{2}(\cos\theta+\sin\theta)$$

Furthermore, we can find expressions for cos α and cos β in terms of u and v.

$$\cos\alpha = r\frac{\sqrt{2}}{2}(\cos\theta-\sin\theta) = r\frac{\sqrt{2}}{2}\left(\frac{u}{\sqrt{u^2+v^2}}-\frac{v}{\sqrt{u^2+v^2}}\right) = \sqrt{u^2+v^2}\,\frac{\sqrt{2}}{2}\frac{u-v}{\sqrt{u^2+v^2}} = \frac{\sqrt{2}}{2}(u-v)$$

$$\cos\beta = r\frac{\sqrt{2}}{2}(\cos\theta+\sin\theta) = r\frac{\sqrt{2}}{2}\left(\frac{u}{\sqrt{u^2+v^2}}+\frac{v}{\sqrt{u^2+v^2}}\right) = \sqrt{u^2+v^2}\,\frac{\sqrt{2}}{2}\frac{u+v}{\sqrt{u^2+v^2}} = \frac{\sqrt{2}}{2}(u+v)$$

$$\therefore \qquad \cos\alpha = \frac{\sqrt{2}}{2}(u-v) \qquad \cos\beta = \frac{\sqrt{2}}{2}(u+v)$$

Step 2: Find an expression for x and y in terms of angle α and β

From step 1, we can write cos α and cos β in terms of x and y

$$\cos\alpha = \frac{\sqrt{2}}{2}\left(x\sqrt{1-\frac{y^2}{2}} - y\sqrt{1-\frac{x^2}{2}}\right) = \frac{x}{\sqrt{2}}\sqrt{1-\frac{y^2}{2}} - \frac{y}{\sqrt{2}}\sqrt{1-\frac{x^2}{2}}$$

$$\cos\beta = \frac{\sqrt{2}}{2}\left(x\sqrt{1-\frac{y^2}{2}} + y\sqrt{1-\frac{x^2}{2}}\right) = \frac{x}{\sqrt{2}}\sqrt{1-\frac{y^2}{2}} + \frac{y}{\sqrt{2}}\sqrt{1-\frac{x^2}{2}}$$

We now introduce another set of intermediate trigonometric variables ϕ and λ. Define:

$$\phi = \cos^{-1}\frac{x}{\sqrt{2}}$$

$$\lambda = \sin^{-1}\frac{y}{\sqrt{2}}$$

Note that since x≤1 and y≤1, it is okay to compute their inverse trigonometric values. This implies

$$\cos\phi = \frac{x}{\sqrt{2}} \qquad \sin\phi = \sqrt{1-\frac{x^2}{2}}$$

and

$$\cos\lambda = \sqrt{1-\frac{y^2}{2}} \qquad \sin\lambda = \frac{y}{\sqrt{2}}$$

So we can write cos α and cos β in terms of ϕ and λ

$$\cos\alpha = \cos\phi\cos\lambda - \sin\phi\sin\lambda = \cos(\phi+\lambda)$$
$$\cos\beta = \cos\phi\cos\lambda + \sin\phi\sin\lambda = \cos(\phi-\lambda)$$

Hence

$$\alpha = \phi + \lambda = \cos^{-1}\frac{x}{\sqrt{2}} + \sin^{-1}\frac{y}{\sqrt{2}}$$
$$\beta = \phi - \lambda = \cos^{-1}\frac{x}{\sqrt{2}} - \sin^{-1}\frac{y}{\sqrt{2}}$$

Taking the sum and difference of these two equations, we get

$$\alpha + \beta = 2\cos^{-1}\frac{x}{\sqrt{2}}$$

$$\alpha - \beta = 2\sin^{-1}\frac{y}{\sqrt{2}}$$

Rearranging the terms, we get

$$x = \sqrt{2}\cos\left(\frac{\alpha+\beta}{2}\right)$$

$$y = \sqrt{2}\sin\left(\frac{\alpha-\beta}{2}\right)$$

Step 3: get an expression for x and y in terms of u and v

We start with the result from step 2 and expand using trigonometric identities for sums and half-angles

$$x = \sqrt{2}\cos\left(\frac{\alpha}{2}+\frac{\beta}{2}\right) = \sqrt{2}\left(\cos\frac{\alpha}{2}\cos\frac{\beta}{2} - \sin\frac{\alpha}{2}\sin\frac{\beta}{2}\right) = \sqrt{2}\left(\sqrt{\frac{(1+\cos\alpha)}{2}}\sqrt{\frac{(1+\cos\beta)}{2}} - \sqrt{\frac{(1-\cos\alpha)}{2}}\sqrt{\frac{(1-\cos\beta)}{2}}\right)$$
$$= \frac{1}{2}\sqrt{2(1+\cos\alpha)(1+\cos\beta)} - \frac{1}{2}\sqrt{2(1-\cos\alpha)(1-\cos\beta)}$$

$$y = \sqrt{2}\sin\left(\frac{\alpha}{2}-\frac{\beta}{2}\right) = \sqrt{2}\left(\sin\frac{\alpha}{2}\cos\frac{\beta}{2} - \cos\frac{\alpha}{2}\sin\frac{\beta}{2}\right) = \sqrt{2}\left(\sqrt{\frac{(1-\cos\alpha)}{2}}\sqrt{\frac{(1+\cos\beta)}{2}} - \sqrt{\frac{(1+\cos\alpha)}{2}}\sqrt{\frac{(1-\cos\beta)}{2}}\right)$$
$$= \frac{1}{2}\sqrt{2(1+\cos\alpha)(1+\cos\beta)} - \frac{1}{2}\sqrt{2(1-\cos\alpha)(1-\cos\beta)}$$

Now, recall from step 1 that

$$\cos\alpha = \frac{\sqrt{2}}{2}(u - v) \qquad \cos\beta = \frac{\sqrt{2}}{2}(u + v)$$

So we can substitute u and v values into cos α and cos β

$$x = \frac{1}{2}\sqrt{2(1+\cos\alpha)(1+\cos\beta)} - \frac{1}{2}\sqrt{2(1-\cos\alpha)(1-\cos\beta)}$$

$$= \frac{1}{2}\sqrt{2\left(1+\frac{\sqrt{2}}{2}(u-v)\right)\left(1+\frac{\sqrt{2}}{2}(u+v)\right)} - \frac{1}{2}\sqrt{2\left(1-\frac{\sqrt{2}}{2}(u-v)\right)\left(1-\frac{\sqrt{2}}{2}(u+v)\right)}$$

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

Likewise,

$$y = \frac{1}{2}\sqrt{2(1-\cos\alpha)(1+\cos\beta)} - \frac{1}{2}\sqrt{2(1+\cos\alpha)(1-\cos\beta)}$$

$$= \frac{1}{2}\sqrt{2\left(1-\frac{\sqrt{2}}{2}(u-v)\right)\left(1+\frac{\sqrt{2}}{2}(u+v)\right)} - \frac{1}{2}\sqrt{2\left(1+\frac{\sqrt{2}}{2}(u-v)\right)\left(1-\frac{\sqrt{2}}{2}(u+v)\right)}$$

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

This completes the derivation.

Algorithm Implemented (C++):

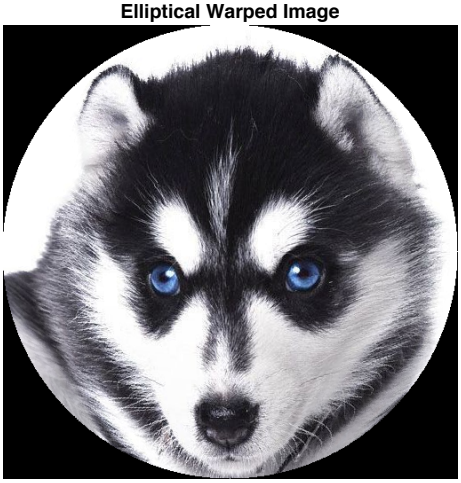
main() Function:

- Read given *puppy.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Allocate memories for output Images using *allocMemory3D()*
- Calculate x and y for every pixel location which is between -1 and 1
- Calculate u and v for every pixel using the forward elliptical warping formula
- Retrace u and v pixel locations to 0 to 512
- Ouput the Forward warping pixel
- Calculate xnew and ynew for every u,v using Inverse Warping formula
- Retrace xnew and ynew to 0 to 512
- Output the Reverse warping pixel
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete, freeMemory3D()*
- Repeat the same procedure for *tiger.raw* image
- Repeat the same procedure for *panda.raw* image

➢ EXPERIMENTAL RESULTS:

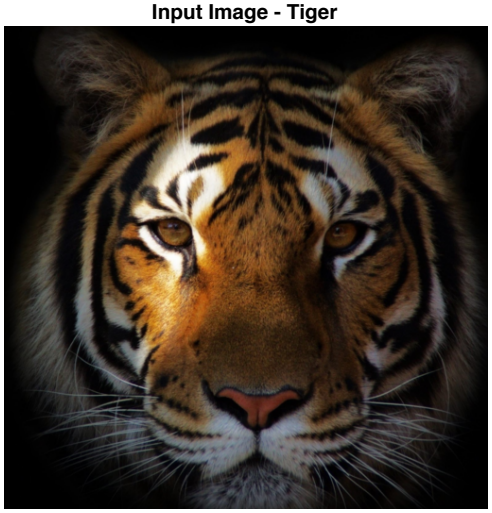
Input Image - Puppy

Elliptical Warped Image



Reversed Image

Input Image - Tiger



Elliptical Warped Image

Reversed Image

Input Image - Panda

Elliptical Warped Image

Reversed Image

➢ DISCUSSION:

- Thus, using Elliptical Image Warping, the given panda, tiger and puppy images were shaped to a disk-shaped image. Also, they were reversed back.
- The reversal looks exactly as same as the Original image. Except for 4 pixels corresponding to the corner of the image. Otherwise, the reversed image is 99.9% similar to the given input image.
- The reversal image should look the same since, in geometrical image warping, the pixel locations are changed and not the actual pixel intensity values itself.
- These change in pixel locations are considered to be one-to-one mapping.
- Since this mapping of pixels are one-to-one mapping, they can be reversed back to original pixel locations from disk-shaped pixel locations.
- Since puppy and panda images are in a white background, when they are warped to disk-shaped images, the background becomes black.
- The given tiger image is already in a black background and tiger face is located in a rounded way. Hence when it is warped to disk-shaped image, there is not much difference in background. There is a small change in foreground and chin areas of the face.

B) HOMOGRAPHIC TRANSFORMATION AND IMAGE STITCHING:

➢ ABSTRACT AND MOTIVATION:

The usage of cameras has become widespread, and it's time to make many exciting experiments and create newer and enthralling experiences. One such marvelous innovation is creating a Panorama. A Panoramic photo stitching is the process of combining multiple photographic images taken by a rotating camera with overlapping fields of view to produce a panorama. This uses the simple concept of homographic image warping of one image based on another image and stitching of images. In this part of experiment, I have created a Panorama based on three given images (left, middle and right). Assumption is that the camera was kept at the same position but rotated from left to right. Details of implementation is discussed below.

➢ APPROACH AND PROCEDURES:

Homographic Image Transformation:

Two images can be related by homography, only if both the images are viewed from the same plane but at different angles. In the given question, for example, left and middle images are taken from the same camera but at different angles. Camera is rotated about its center of projection without any translation.

A 2D image can be represented in homogenous coordinates:

- Homogeneous coordinates – (u, v, w)
- Image coordinates – (u/w, v/w)

These Homogenous coordinates can be used to frame the Homographic matrix for transformation.

Let us consider,

- pair (x,y) belong to middle image(reference image)
- (u,v) belong to left or right image (images to be warped)

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{pmatrix} \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ w \end{pmatrix}$$ where x = x'/w and y = y'/w

The values for a to h can be solved by mapping 4 points from the image to be warped to the reference image. Since we are taking 4 points, we would get 8 equations to solve for the unknowns.
(Source: http://cs.brown.edu/courses/csci1290/2011/results/proj6/ebnelson/)

$$\begin{pmatrix} u_0 & v_0 & 1 & 0 & 0 & 0 & -u_0x_0 & -v_0x_0 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -v_1x_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -v_2x_2 \\ u_3 & v_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -v_3x_3 \\ 0 & 0 & 0 & u_0 & v_0 & 1 & -u_0y_0 & -v_0y_0 \\ 0 & 0 & 0 & u_1 & v_1 & 1 & -u_1y_1 & -v_1y_1 \\ 0 & 0 & 0 & u_2 & v_2 & 1 & -u_2y_2 & -v_2y_2 \\ 0 & 0 & 0 & u_3 & v_3 & 1 & -u_3y_3 & -v_3y_3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Procedure to create panorama using the above mentioned Homographic Transformation:

- Bound the middle image in a bigger image. I have used image size of 1240*2000 and the middle image is placed at a row offset of 300 and column offset of 795
- The control points from middle and left/right images are obtained. They are put in the above equation and MATLAB Linear Equation solving method using pseudo inverse was used to solve for the above big matrix.
- Once the Homographic Matrix H is calculated, we can map points from left image to middle image by forward warping
- Forward Warping creates unnecessary noise and gap in the warped image due to missing points. Hence, it's a common procedure to do inverse warping, to orient the left/right image in the direction of middle image
- The middle image points are obtained. Using inverse warping, they are mapped to the left/right image. The pixel intensity at that point is obtained by Bilinear Interpolation
- The interpolated intensity value is replaced in the bounded middle image
- Once the warped images are obtained, overlapping points in the bounded middle image are averaged to get a smooth transition between images
- The process of stitching happens along with this warping procedure since we are obtaining middle image points by Homographic transformation and bilinear interpolation. No need of extra stitching method except for averaging the overlapping pixel points.

Algorithm Implemented (C++):

main() Function:

- Read given *left.raw, middle.raw and right.raw* image using *fileRead()* function
- Convert 1D images to 3D using *image1Dto3D()* function
- Allocate memories for output Images using *allocMemory3D()*
- Get H and Hinv matrices for left and right warping from Matlab
- Convert 1D H and Hinv matrices to 3D using *convert1DtoMatix()*
- Bound the middle image to a bigger image of given size and offsets
- Write the bounded image to output raw file using *fileWrite()*
- Inverse warp the left image to the bounded image using *inverseWarping()*
- Write the bounded image to output raw file using *fileWrite()*
- Inverse warp the right image to the bounded image using *inverseWarping()*
- Write the bounded image to output raw file using *fileWrite()*
- Re put the middle image into the bounded image by averaging the overlapping pixel locations
- Write the bounded image to output raw file using *fileWrite()*
- Deallocate all memories

Convert1DtoMatrix() Function:

- Given a 1D array of size 9
- Allocate memory for 3*3
- Convert to 3*3 array
- Return 3D array

multiplyMatrixVector() Function:

- Given a 3D matrix and 1D array
- Multiply using Matrix Multiplication formula
- Return 1D array

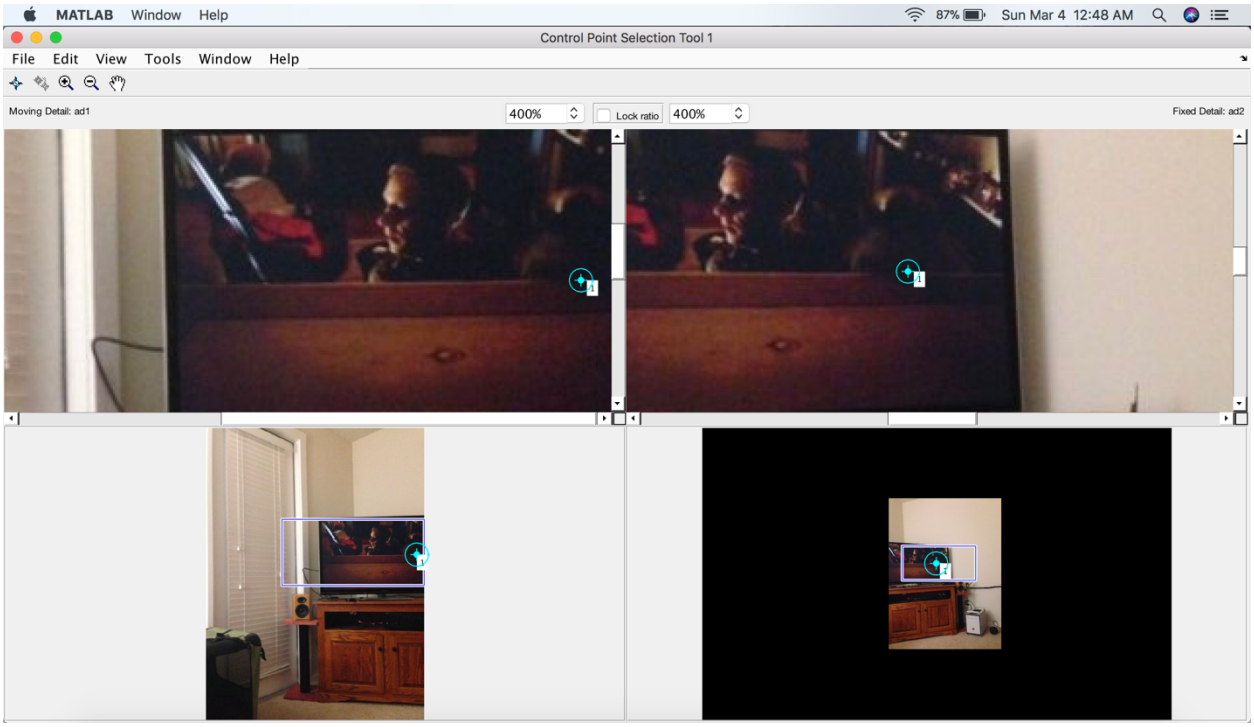
inverseWarping() Function:

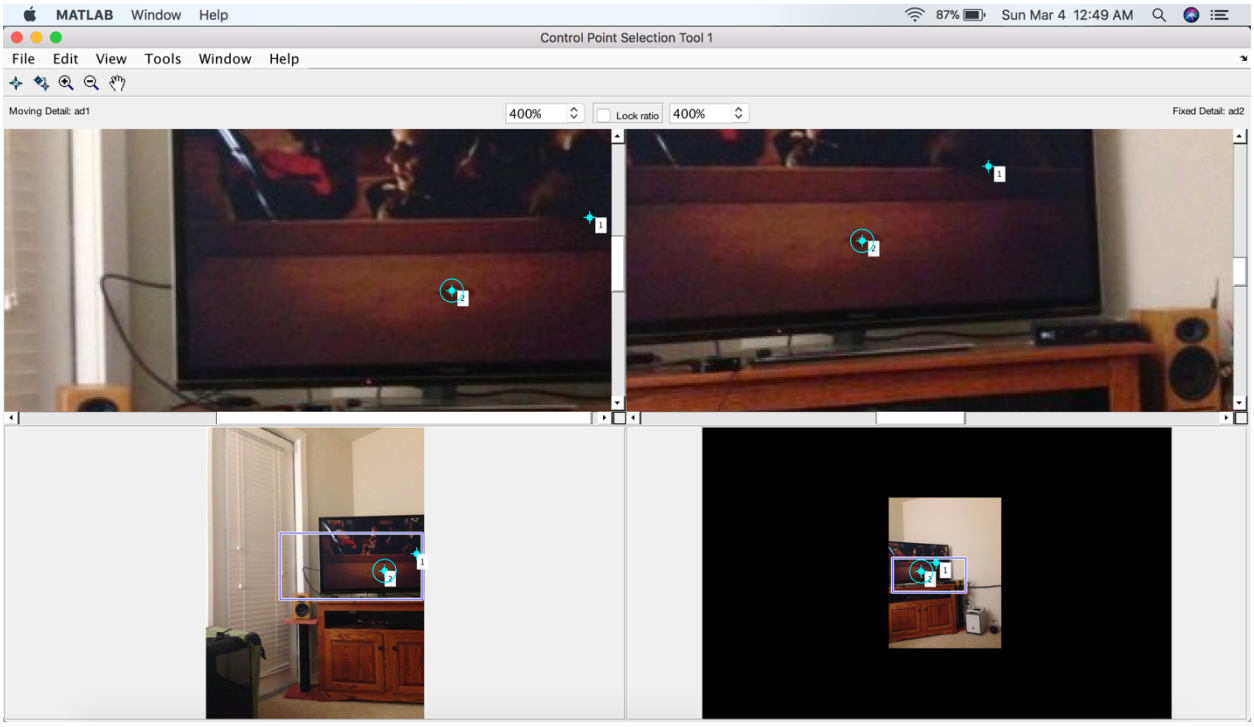
- Traverse through bounded middle image using 2 for loops
- For every x, y pixel location value, get corresponding left/right image coordinates by multiplying with Hinv using *multiplyMatrixVector()* function and dividing by weight value obtained
- Bilinear interpolate to get pixel intensity values for x,y
- Output that value in x,y location of bounded image
- Return bounded middle image

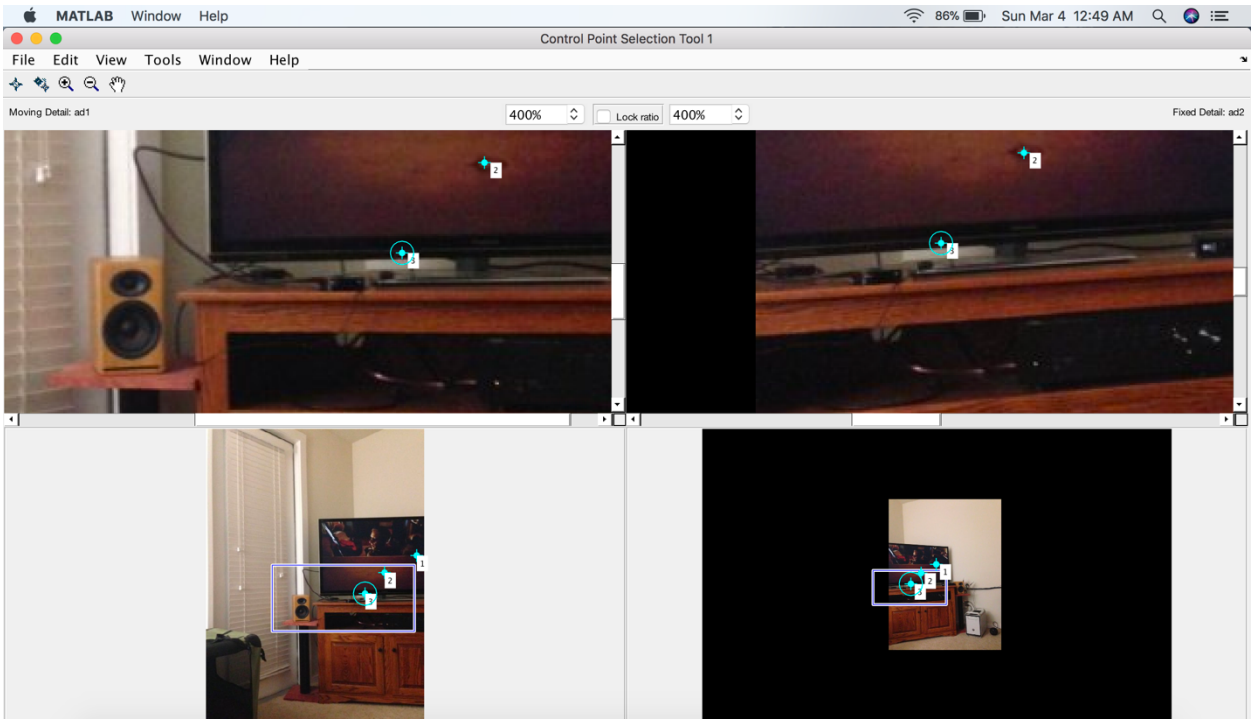
forwardWarping() Function:

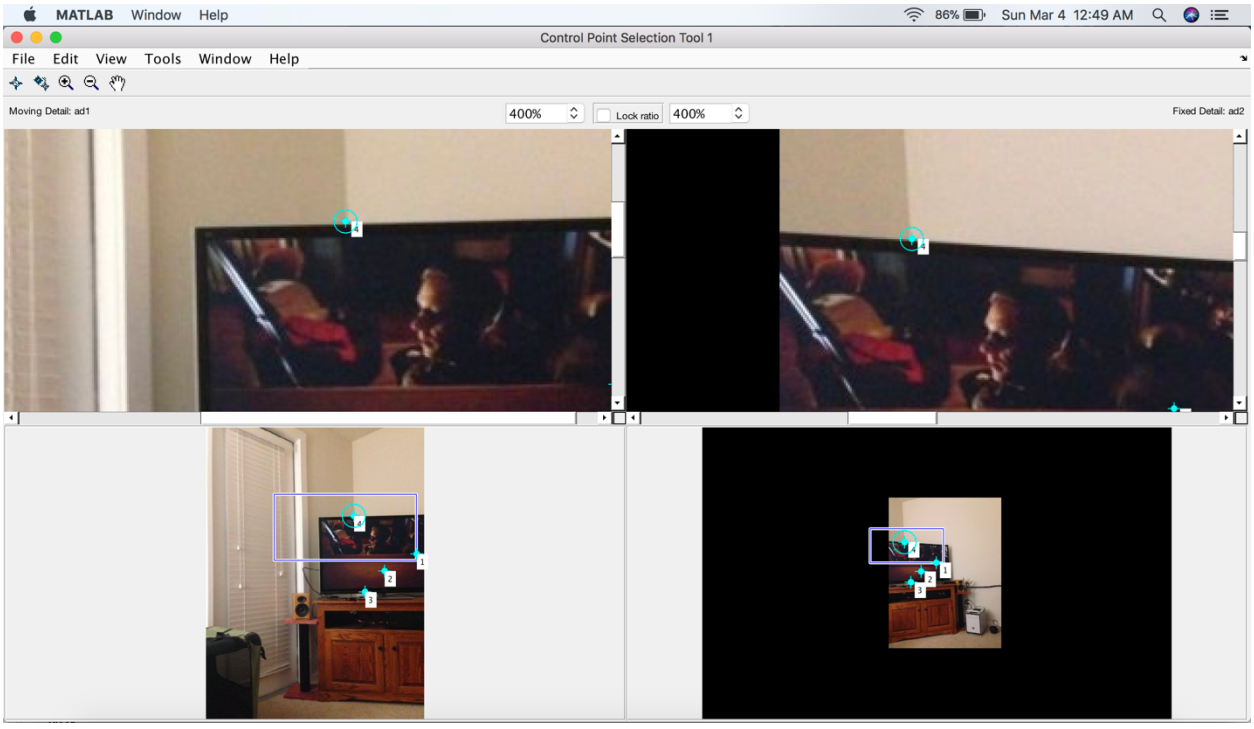
- Get boundary coordinates of left/right image
- For those coordinates, get corresponding middle bounded image coordinates by multiplying with H matrix using *multiplyMatrixVector()* functionand dividing by weight value obtained
- Return the boundary coordinates for inverse warping

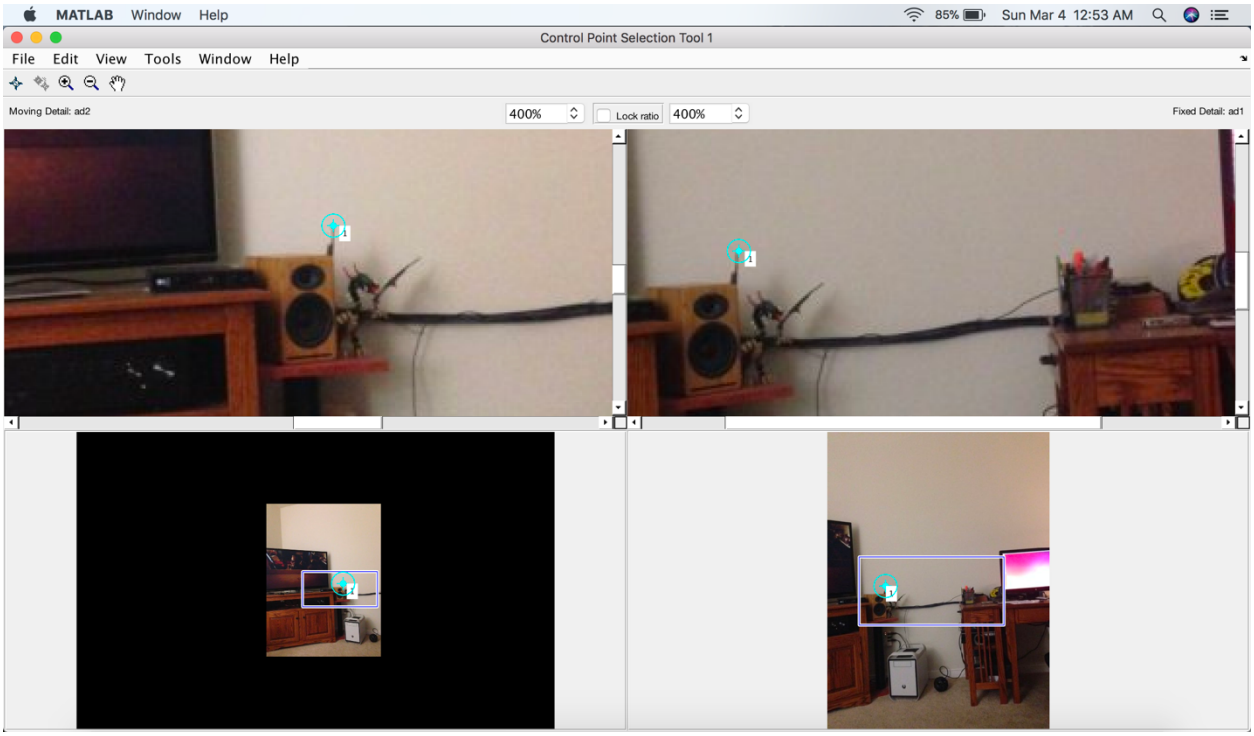
➢ EXPERIMENTAL RESULTS:

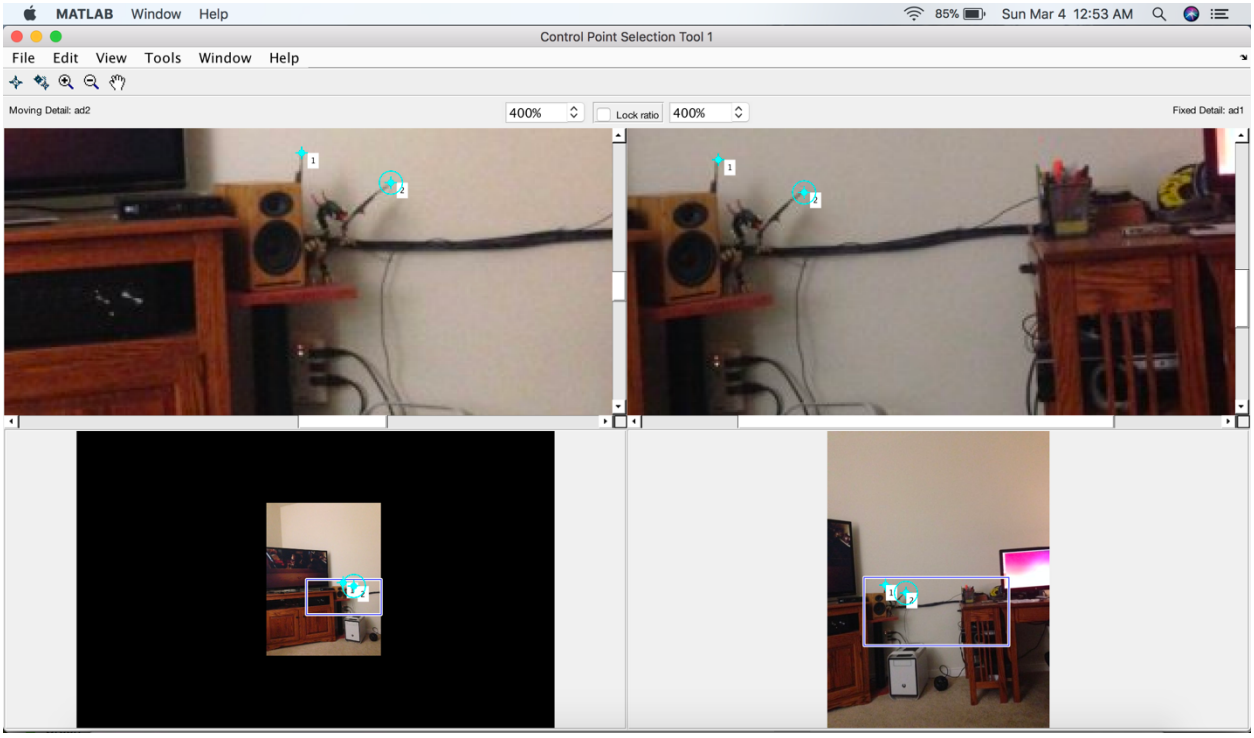


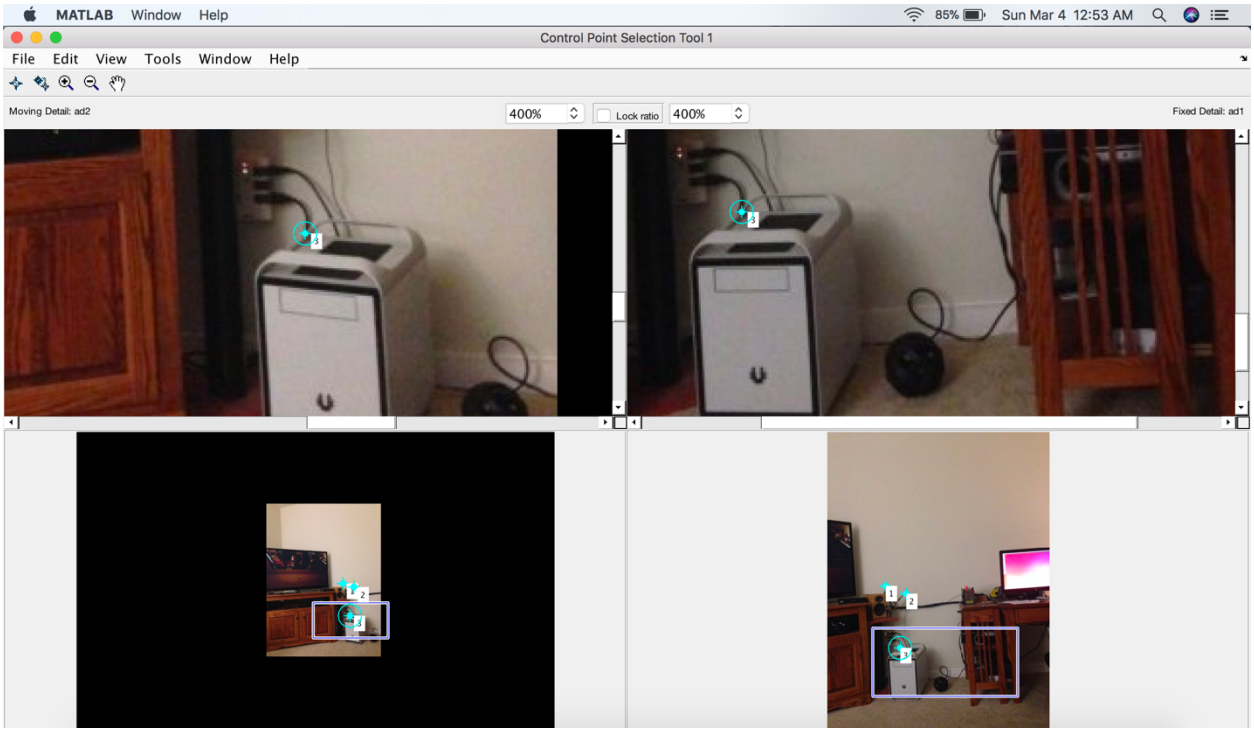


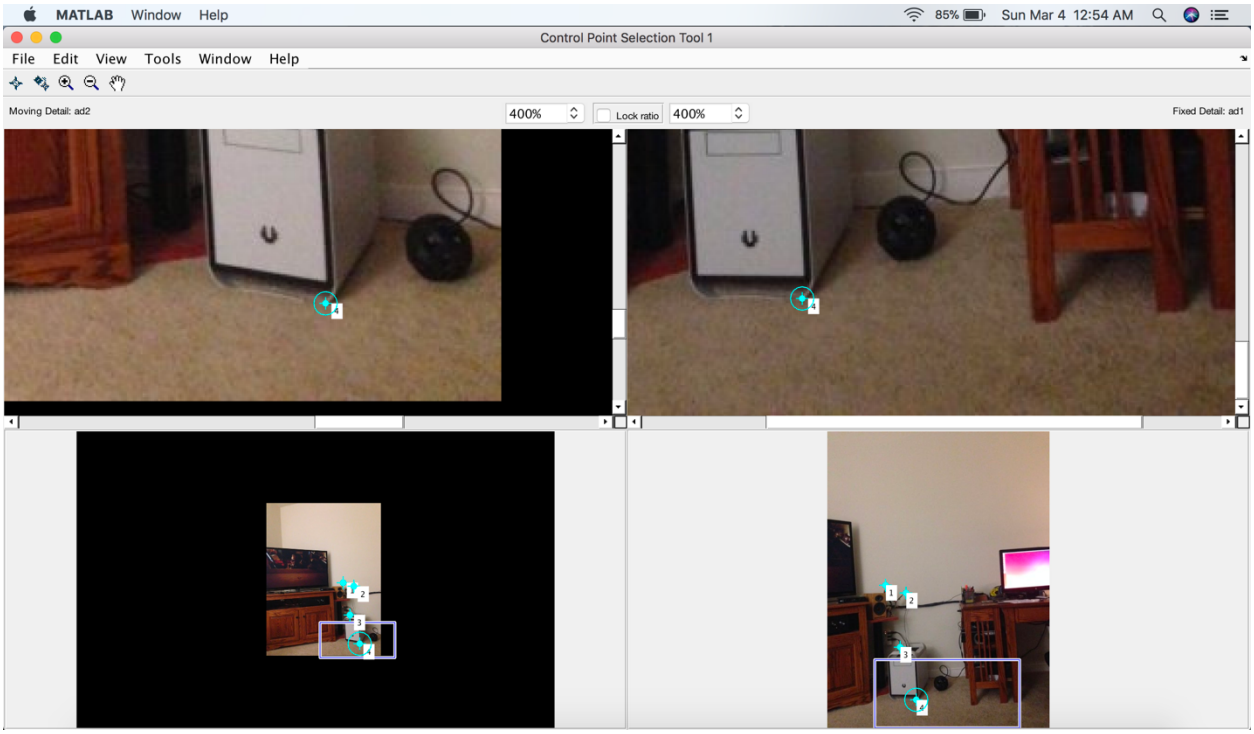

MATLAB Window Help
86% Sun Mar 4 12:49 AM
Control Point Selection Tool 1
File Edit View Tools Window Help
Moving Detail: ad1
400%
Lock ratio
400%
Fixed Detail: ad2


MATLAB Window Help
86% Sun Mar 4 12:49 AM
Control Point Selection Tool 1
File Edit View Tools Window Help
Moving Detail: ad1
400%
Lock ratio
400%
Fixed Detail: ad2


MATLAB Window Help
85% Sun Mar 4 12:53 AM
Control Point Selection Tool 1
File Edit View Tools Window Help
Moving Detail: ad2
400%
Lock ratio
400%
Fixed Detail: ad1

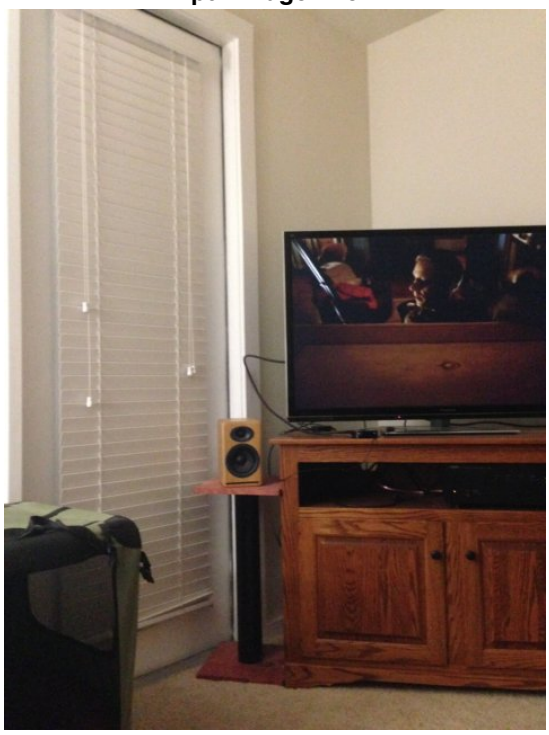

MATLAB Window Help
85% Sun Mar 4 12:53 AM
Control Point Selection Tool 1
File Edit View Tools Window Help
Moving Detail: ad2
400%
Lock ratio
400%
Fixed Detail: ad1



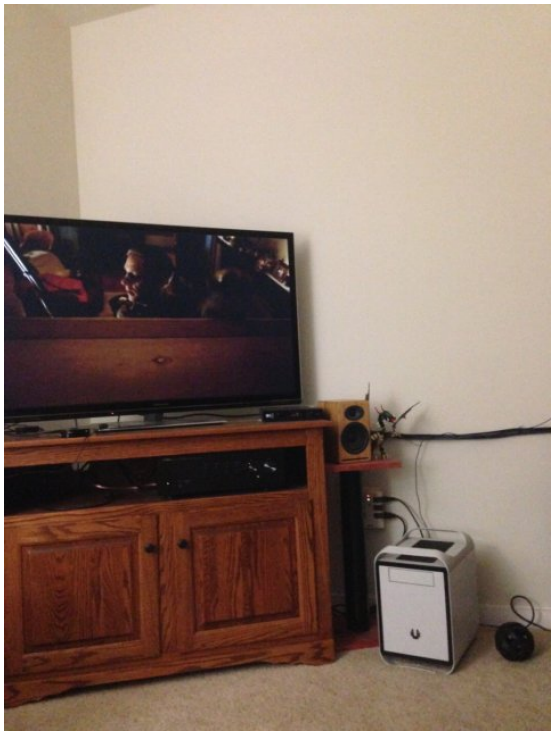


After many trial and error of point selection, these points were chosen and solved for Homographic using MATLAB

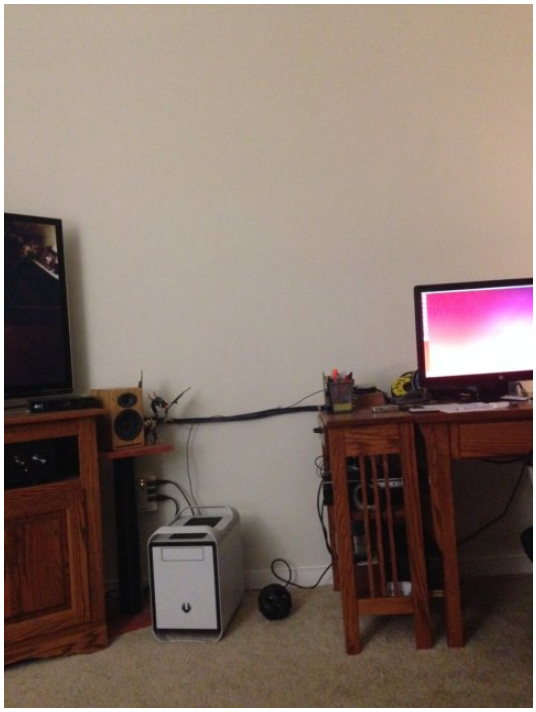
Input Image - Left



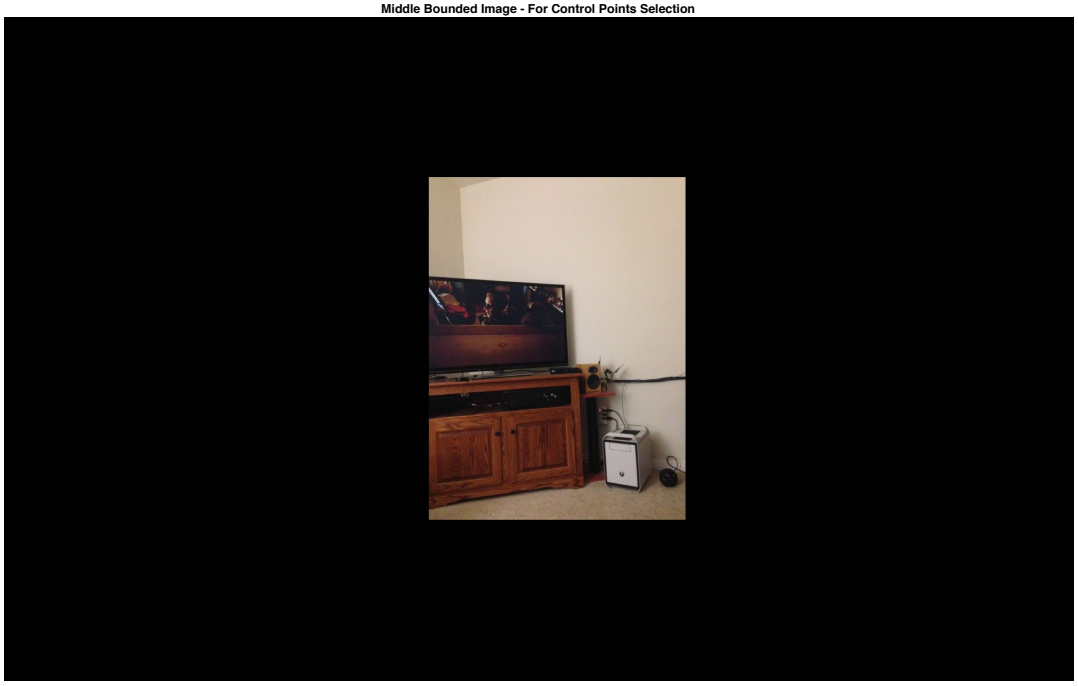
Input Image - Middle



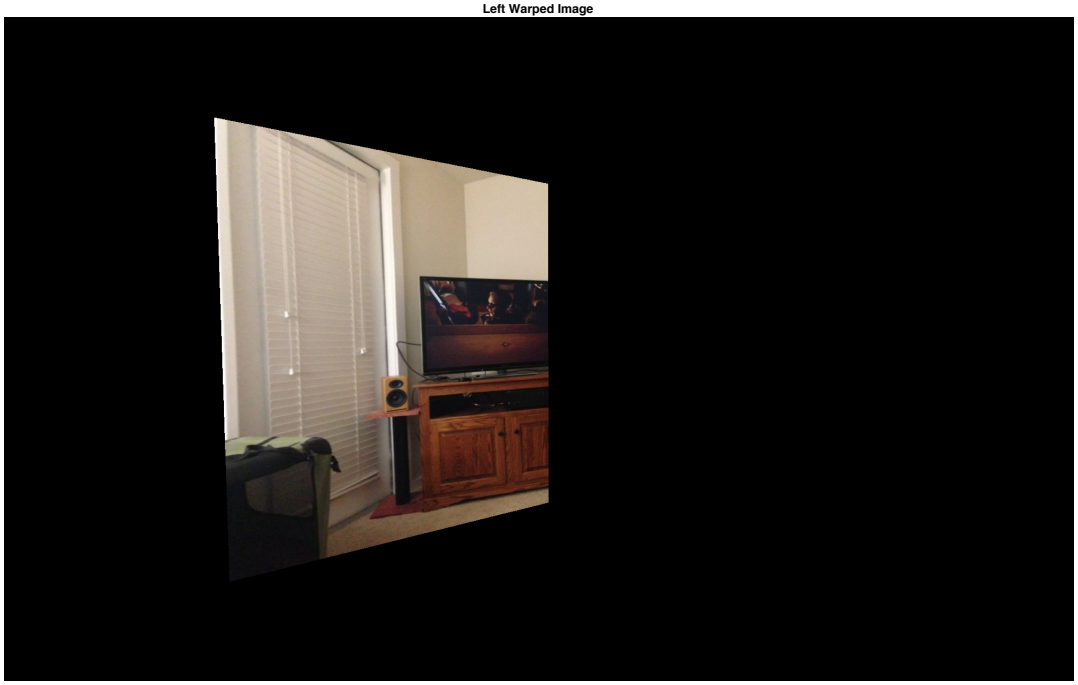
Input Image - Right



Middle Bounded Image - For Control Points Selection

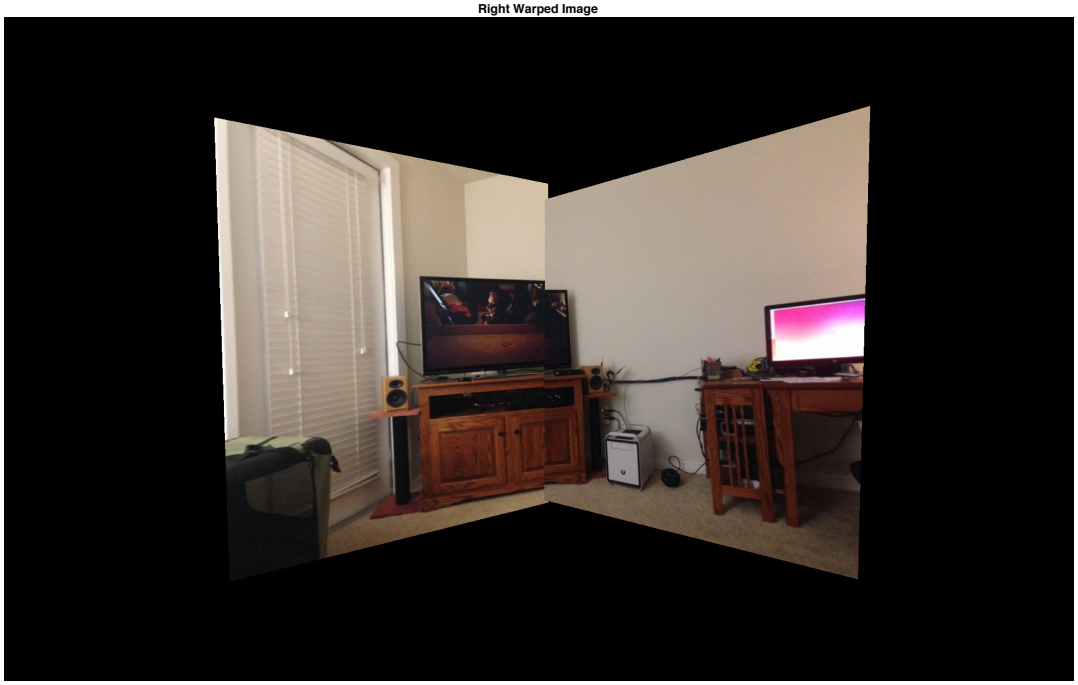


Left Warped Image

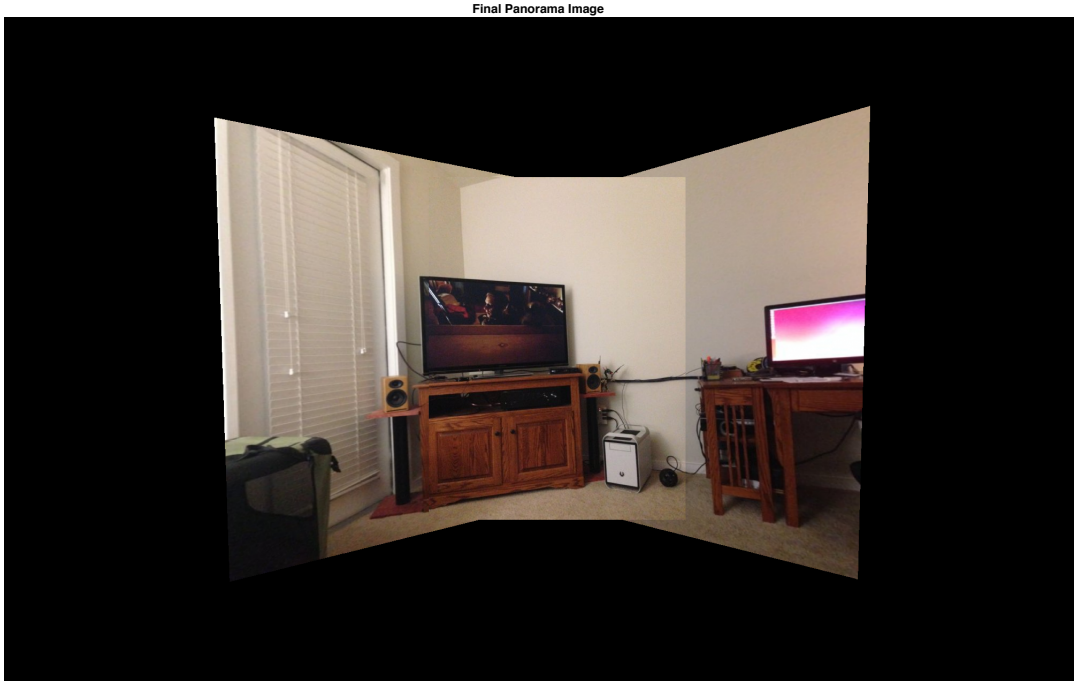


The middle image was bounded on a bigger image to get the control points for Homographic Matrix Calculation.
Left Image warped using inverse Warping

Right Warped Image

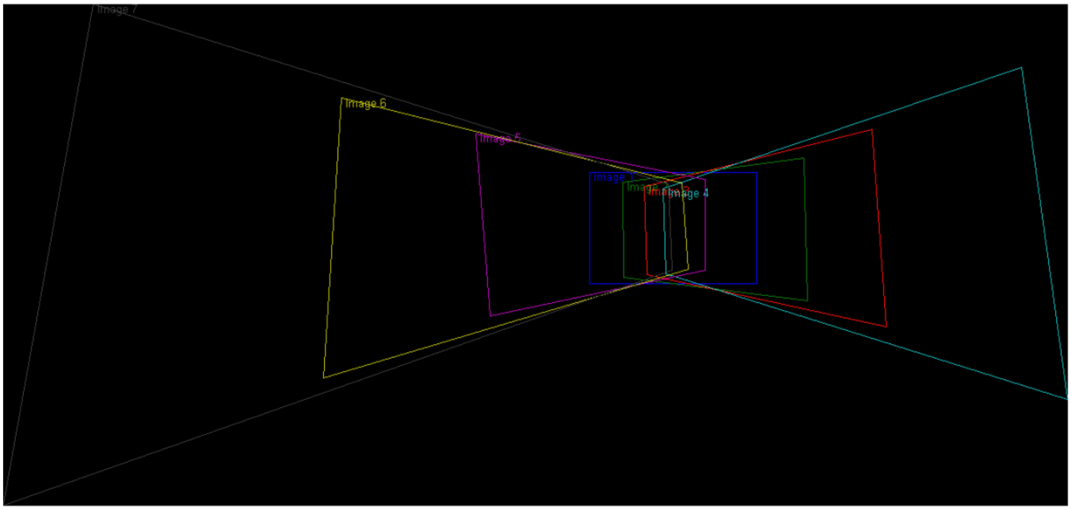


Final Panorama Image



➢ DISCUSSION:

- This was the toughest question in this Homework. I took various points in trial and error basis to get the control points to obtain this panorama output.
- After many experiments, I observed that when I chose 3 points close to each other and 1 point a little far away, there was less distortion in Homographic Warping.
- If given more time, we can take more than 4 points, to get even more perfect and well fit panoramic image.
- The right image is different from middle and left image because of lens distortion. It is not related to the way panorama was created.
- For this experiment, since all the 3 given images were well taken without any disturbance in the rotation of camera, 4 points were enough to get a proper panorama image
- And points selection need not be manual. Thera are many algorithms to get control points. One such algorithm is RANSAC (Random Sample Consensus), which is an iterative method to estimate parameters of a mathematical model.
- Also, if we have many images to get a panaroma, we can warp every left and right images to the middle image as shown below
  (Source: http://vision.gel.ulaval.ca/~jflalonde/cours/4105/h15/tps/results/tp4/ALHAJ1/)



PROBLEM 2
DIGITAL HALF-TONING

C) DITHERING:

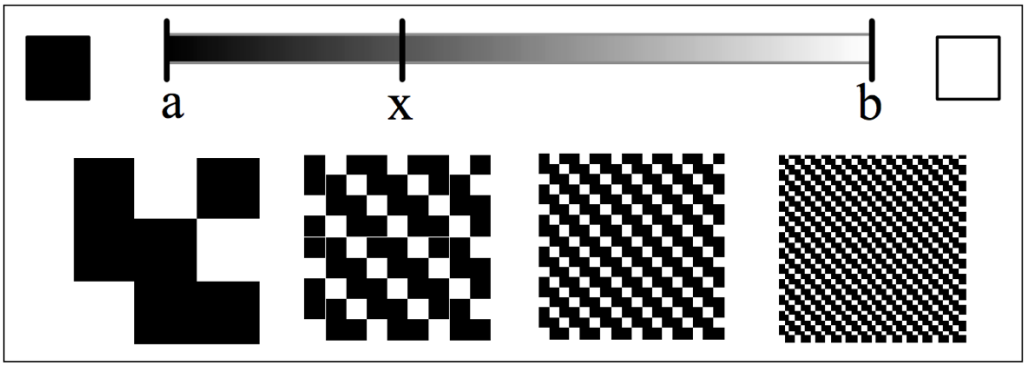
➢ ABSTRACT AND MOTIVATION:

The science of digital halftoning is the process of attaining satisfactory color reduction and image rendering. This is mainly achieved by the dithering technique. Initially, when there were displays which could only display 2 colors – full white or full black, dithering was used to convert the continuous image to image composing of 2 colors to facilitate the process of displaying continuous images. By continuous image, I mean images with more than 2 intensity levels like grayscale image (0-255 intensity levels). Hence digital half toning gives the illusion of continuous image by displaying the image using only 2 binary pixels.

➢ APPROACH AND PROCEDURES:

A simple illustration for better understanding the dithering techniques is given below. The pixel x can be assigned to a or b using different method. In our case, considering binary output, a is 0 and b is 255.

(Source: http://alamos.math.arizona.edu/~rychlik/CourseDir/535/resources/RasterGraphics_slides.pdf)



The process of converting continuous grayscale image to a binary image can be done in many ways. I present 3 such techniques for digital halftoning.

- **Fixed Thresholding:** Done based on a fixed threshold of 127

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

- **Random Thresholding:** Done based on a random threshold between 0 and 255 - generated for every pixel

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

- **Dithering:** Dithering parameters are specified by an Index matrix. This will calculate the likelihood of a dot to be turned ON.

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

In the above given Index matrix, 0 represents the pixel most likely to be turned ON, whereas 3 represents the pixel least likely to be turned ON. A generalized Bayer Index Matrix (using recursion) is given below:

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

This above-mentioned Index Matrix can be used to generate threshold matrix. The threshold Matrix generated using Index Matrix as given below. Here N^2 represents the number of pixels in an image.

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

The final Thresholding of every grayscale image pixel is given by the below mentioned equation.

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

Where, G(i,j) is the dithered matrix and F(I,j) is the normalized pixel intensities.

The above Dithering is repeated for 2*2, 4*4 and 8*8 Index Matrices and the outputs are compared.

- **Dithering – 4 Intensity Levels:** This Dithering is similar to 2-grayscale, except for the fact that, each grayscale intensity levels are mapped to 4 unique intensity levels instead of 2 binary levels. I used the below given equation for 4-level Dithering.

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

Algorithm Implemented (C++):

main() Function:

- Read given *colorchecker.raw* image using *fileRead()* function
- Convert 1D image to 2D using *image1Dto2D()* function
- Allocate memories for output Images using *allocMemory2D()*
- Define index Matrices needed
- Apply Fixed Thresholding using *fixedThresholdin()*
- Apply Random Thresholding using *randomThresholding()*
- Apply Dithering using *dithering2Intensities()* and *dithering4Intensities()*
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()*

fixedThresholding() Function:

- Traverse through the image pixels using 2 for loops
- The threshold fixed is 127
- If the image pixel intensity is less than threshold, output intensity is assigned zero
- Else it is assigned 255
- Return Fixed thresholded output

randomThresholding() Function:

- Traverse through the image pixels using 2 for loops
- Generate a random number between 0 and 255 for every pixel
- If the pixel value is less than generated threshold, output intensity is assigned zero
- Else it is assigned 255
- Return Random thresholded output

formIndexMatrix() Function:

- Converts the given 1D index matrix to 2D
- Return 2D index matrix

Dithering2Intensities() Function:

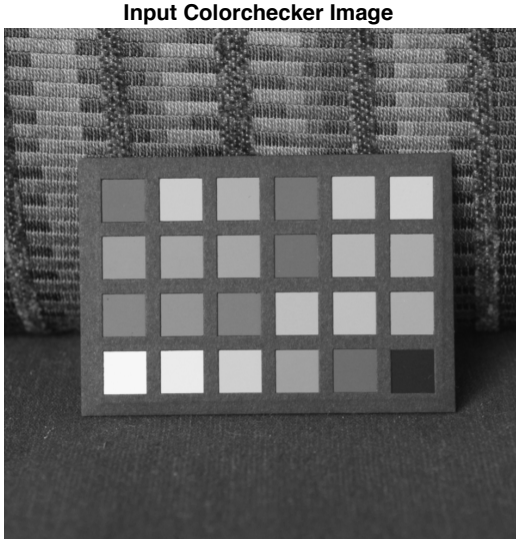
- Traverse through the image pixels using 2 for loops
- Get the threshold matrix value from the Index matrix
- Based on the threshold matrix, assign 0 or 255
- Return 2-level dithered image matrix

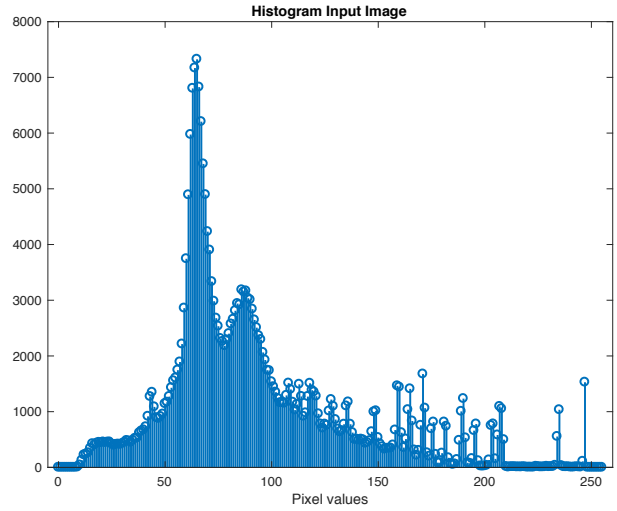
Dithering4Intensities() Function:

- Traverse through the image pixels using 2 for loops
- Get the threshold matrix value from the Index matrix
- Based on the threshold matrix, assign 0, 85, 170 or 255
- Return 4-level dithered image matrix

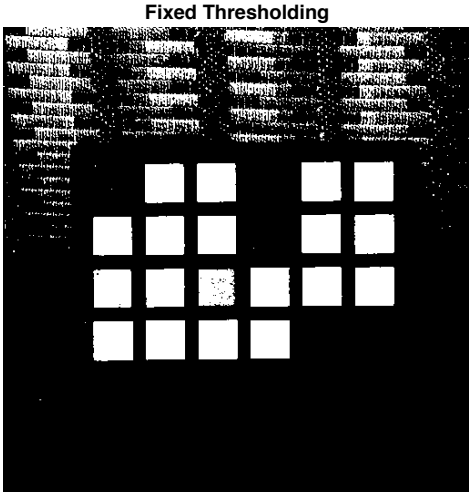
➢ EXPERIMENTAL RESULTS:

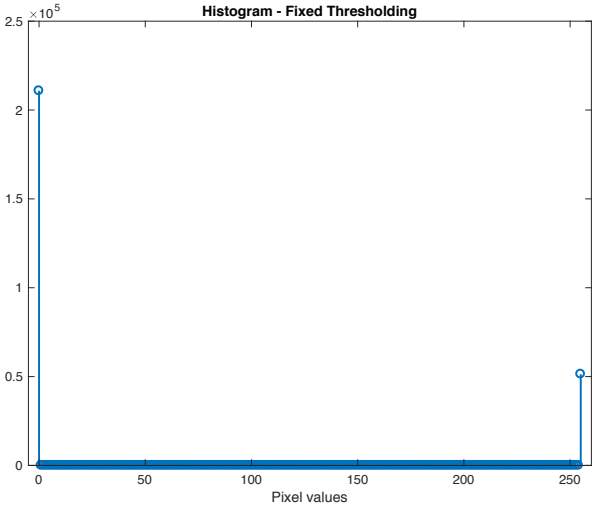
Input Colorchecker Image



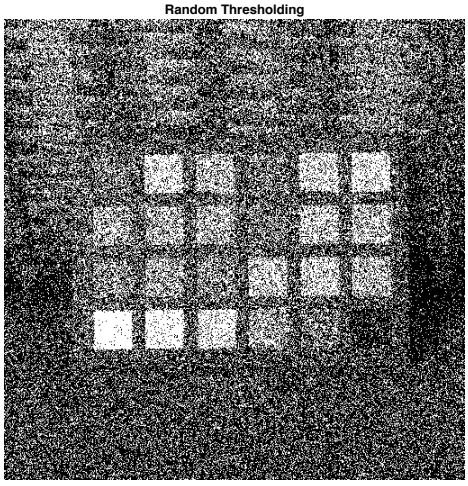


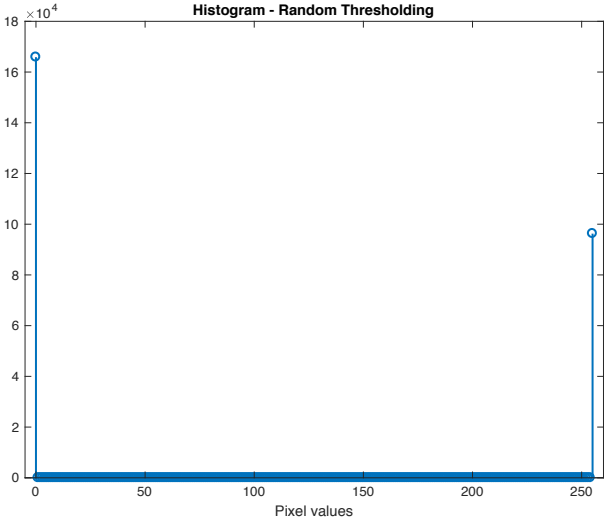
Fixed Thresholding

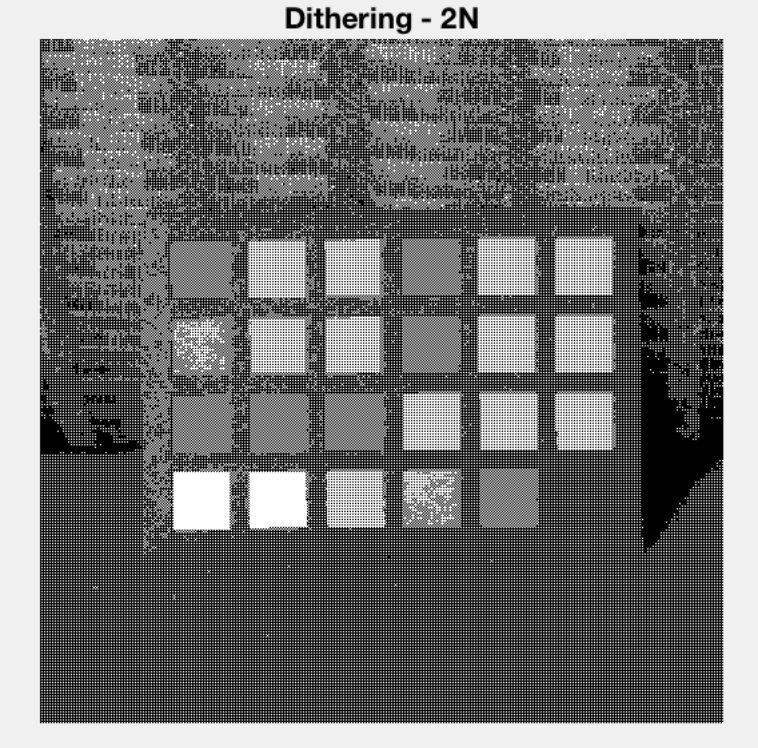


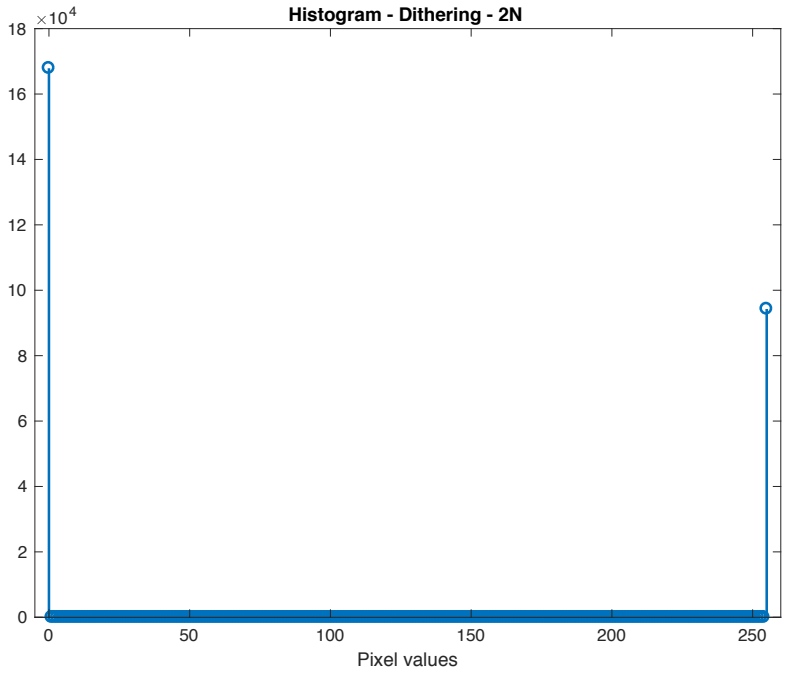


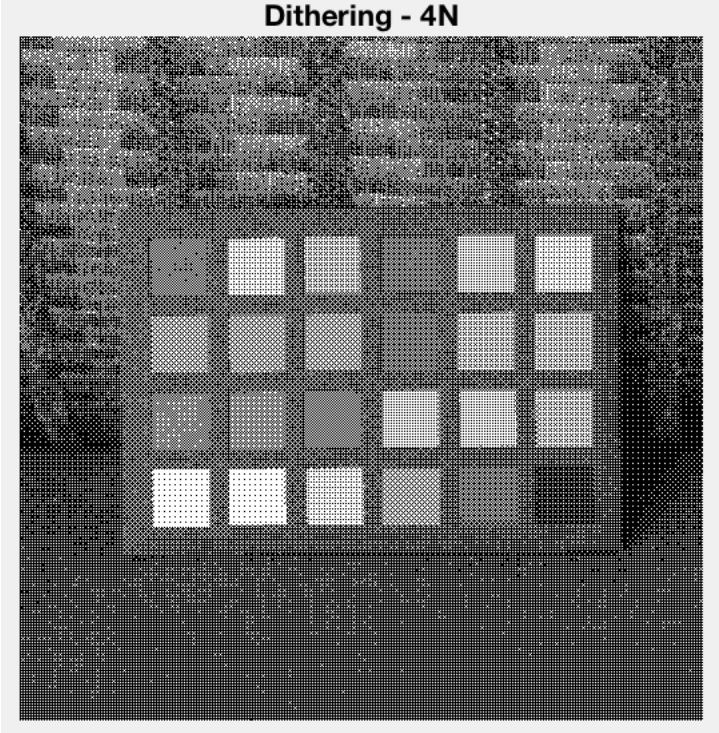
Random Thresholding

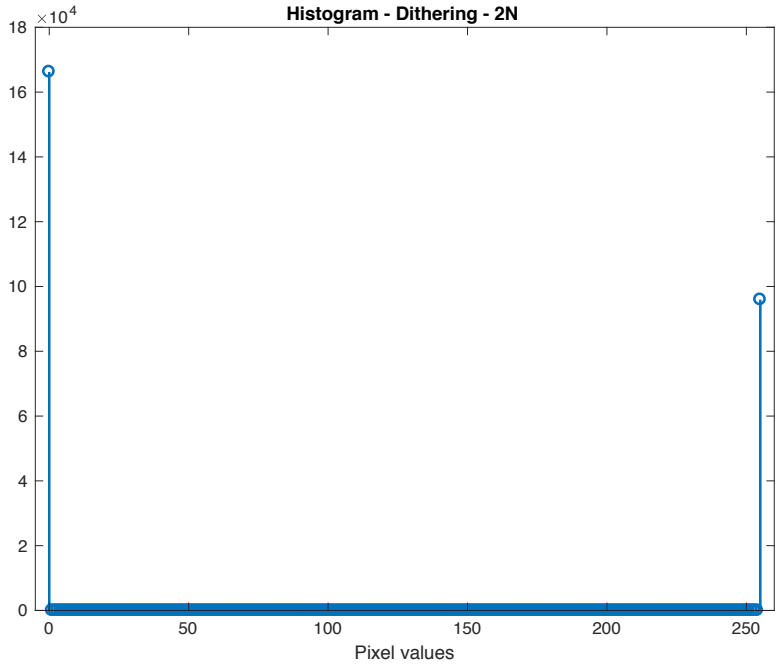


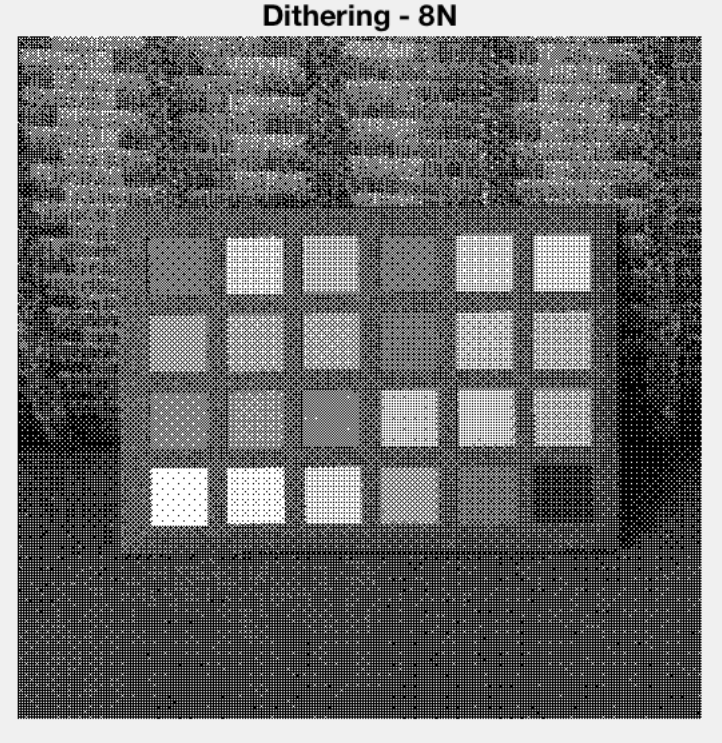


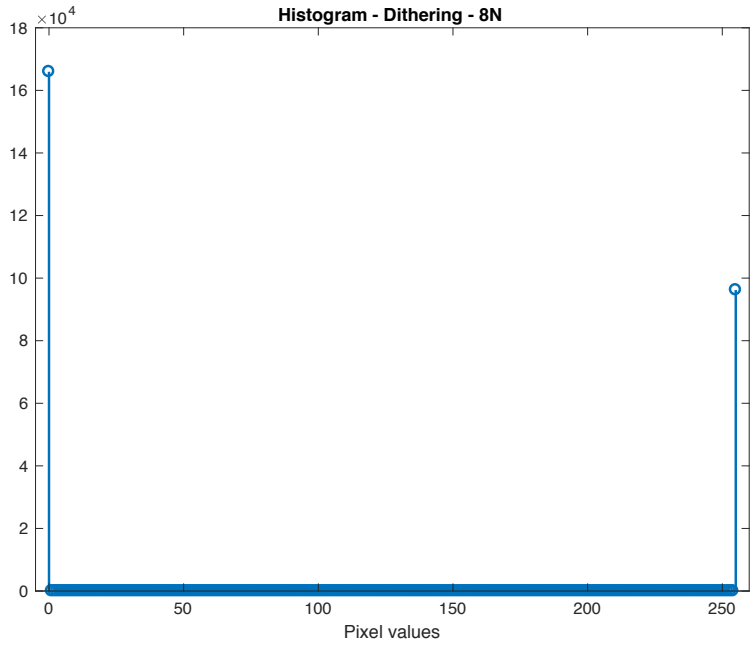

Dithering - 2N

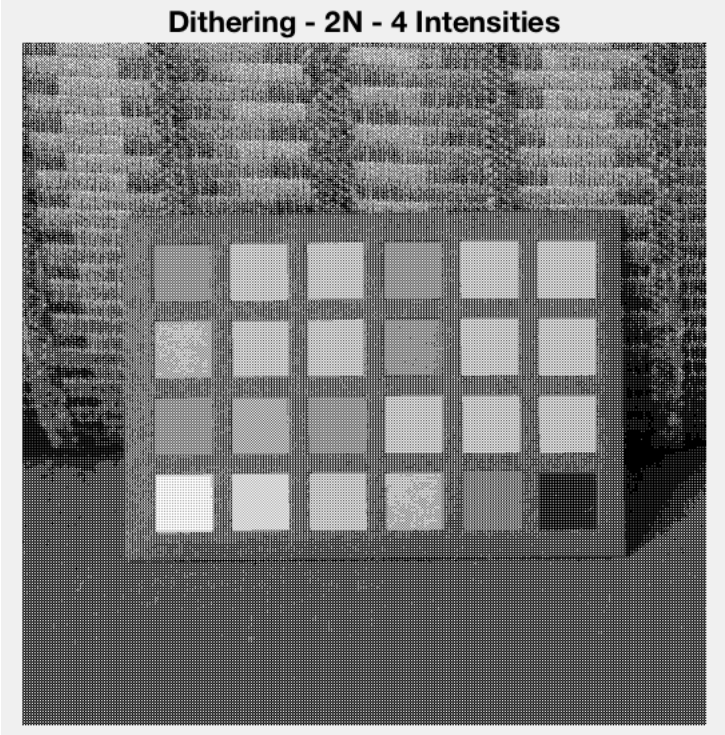

Histogram - Dithering - 2N
×10^4
18
16
14
12
10
8
6
4
2
0
0
50
100
150
200
250
Pixel values

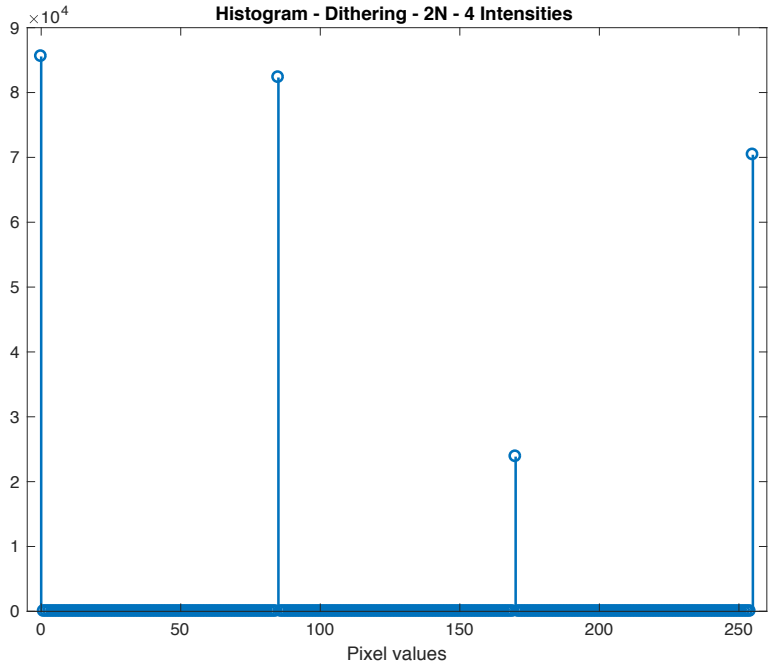

Dithering - 4N

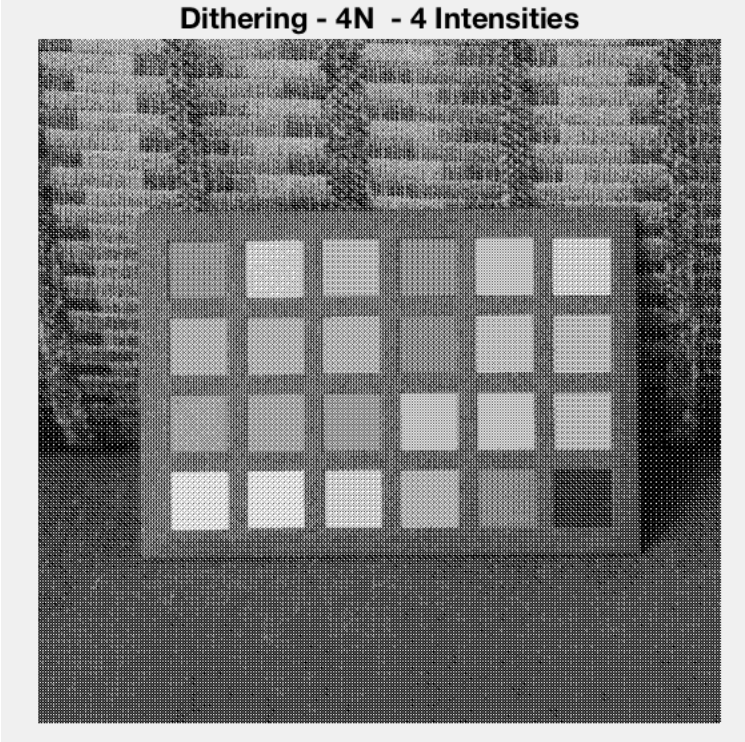

Histogram - Dithering - 2N
×10^4
18
16
14
12
10
8
6
4
2
0
0
50
100
150
200
250
Pixel values

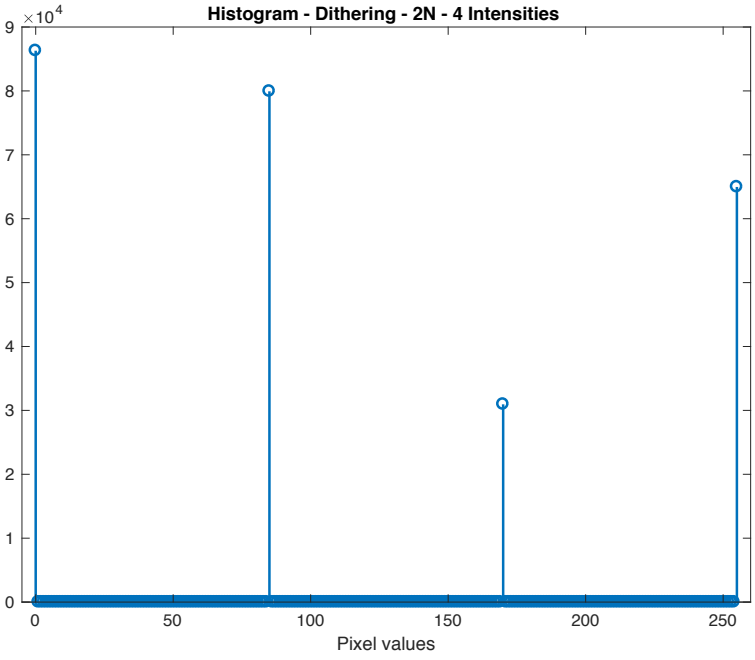

Dithering - 8N

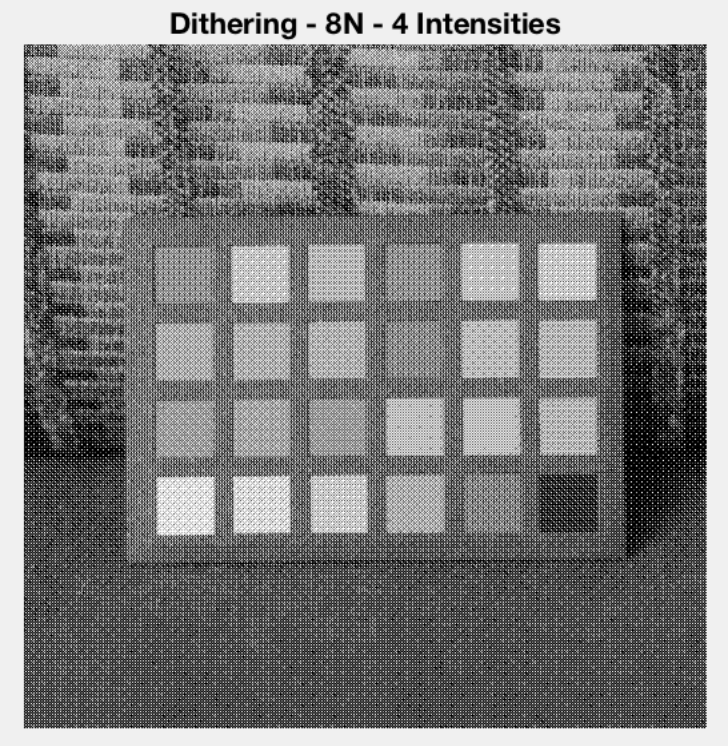

Histogram - Dithering - 8N
×10^4
18
16
14
12
10
8
6
4
2
0
0
50
100
150
200
250
Pixel values

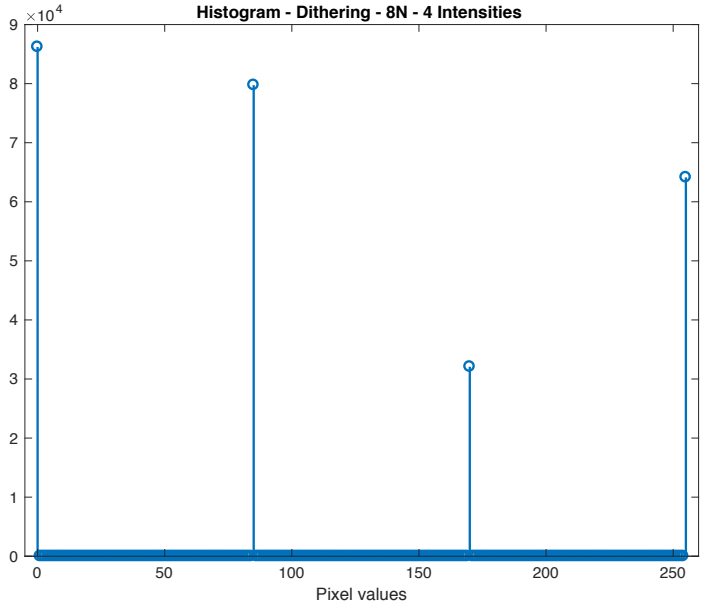

Dithering - 2N - 4 Intensities


Histogram - Dithering - 2N - 4 Intensities
×10^4
9
8
7
6
5
4
3
2
1
0
0
50
100
150
200
250
Pixel values


Dithering - 4N - 4 Intensities


Histogram - Dithering - 2N - 4 Intensities
×10^4
9
8
7
6
5
4
3
2
1
0
0
50
100
150
200
250
Pixel values





➢ DISCUSSION:

- The **Dithering outputs** are very good when compared to Fixed or Random Thresholding. The features and squares of colorchecker image are well distinguished in dithering output
- Similarly, **8*8 Index matrix** Dithering is good when compared with 4*4 or 2*2
- **4 – Intensity level Dithering** is better compared to 2-Intensity level, again because different grayscale squares are viewed better in this method.
- In 4-level Dithering, again **8*8 Index matrix** is better for Digital Halftoning

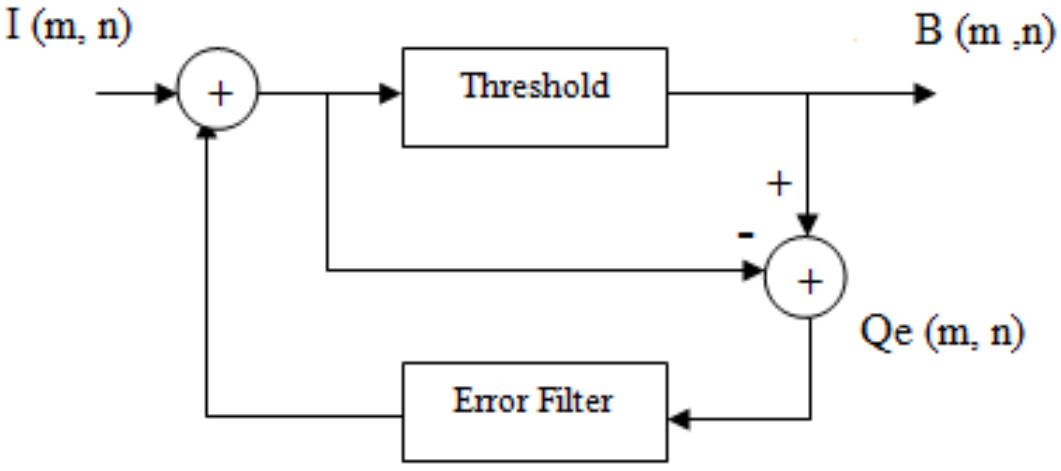
D) ERROR DIFFUSION:

➢ ABSTRACT AND MOTIVATION:

Error Diffusion is a type of Digital Halftoning technique where the error of the present pixel is distributed to the neighboring pixels that have not been traversed still. This kind of Halftoning is called as area diffusion since the effect of the algorithm at one-pixel location affects the pixels at other locations. This is intuitively true because the error is calculated by the algorithm of every pixel and it gets distributed further to all the upcoming pixels. In this part of the question, I have implemented 3 types of error diffusion to perform Digital Halftoning method. At the end of experiment, I compare the results of all the 3 methods. Also, I would be able to explain the differences of Error Diffusion results and the results from normal thresholding and dithering techniques.

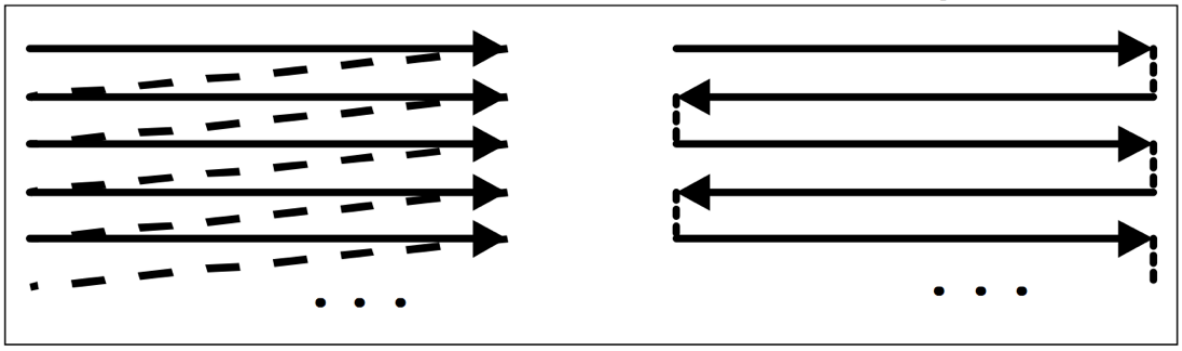
➢ APPROACH AND PROCEDURES:

- The error diffusion technique is given by the following flowchart: The threshold of 127 is chosen for every pixel I(m,n).
- Based on the threshold, the image pixel becomes either 0 or 255.
- The error is then calculated based on the output value and then diffused on the upcoming pixels based on different error diffusion filter discussed below
- (Source: https://www.researchgate.net/figure/Floyd-Steinbergs-error-diffusion-algorithm_fig1_45796483)



The method of error diffusion can be traversed via the image as usual or in the serpentine method. Artificial stripes in the halftoned image can be largely reduced by performing error diffusion in the serpentine manner. The demonstration is shown below: (Left is the normal traversing method and Right is the serpentine method)

(Source: http://alamos.math.arizona.edu/~rychlik/CourseDir/535/resources/RasterGraphics_slides.pdf)



- **Floyd's Error Diffusion (With and Without Serpentine Scanning):**
  The Error Diffusion Matrix is given below:

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

- **Error Diffusion proposed by Jarvis, Judice, and Ninke (JJN) (With and Without Serpentine Scanning):**
  The Error Diffusion Matrix is given below:

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

- **Error diffusion proposed by Stucki (With and Without Serpentine Scanning):**
  The Error Diffusion Matrix is given below:

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

Algorithm Implemented (C++):
main() Function:

- Read given *colorchecker.raw* image using *fileRead()* function
- Convert 1D image to 2D using *image1Dto2D()* function
- Allocate memories for output Images using *allocMemory2D()*
- Define the error diffusion matrices for all three methods using the above Matrices
- Halftone the images using *errorDiffusion()* and *errorDiffusionSerpentine()*
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()*

errorDiffusionSerpentine() Function:

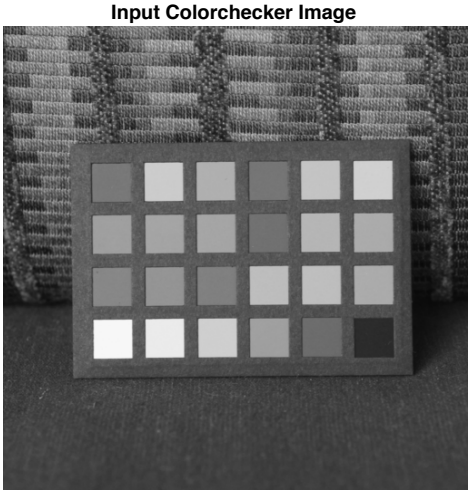
- Traverse through the image using 2 for loops
- If the row index mod 2 is zero, traverse from left to right in the column
- If the row index mod 2 is non-zero, traverse from right to left in the column
- Decide 0 or 255 pixel for the output using threshold 127
- Calculate the error as output – input pixel
- Diffuse errors to the future pixels using the corresponding matrices
- Return halftoned image matrix

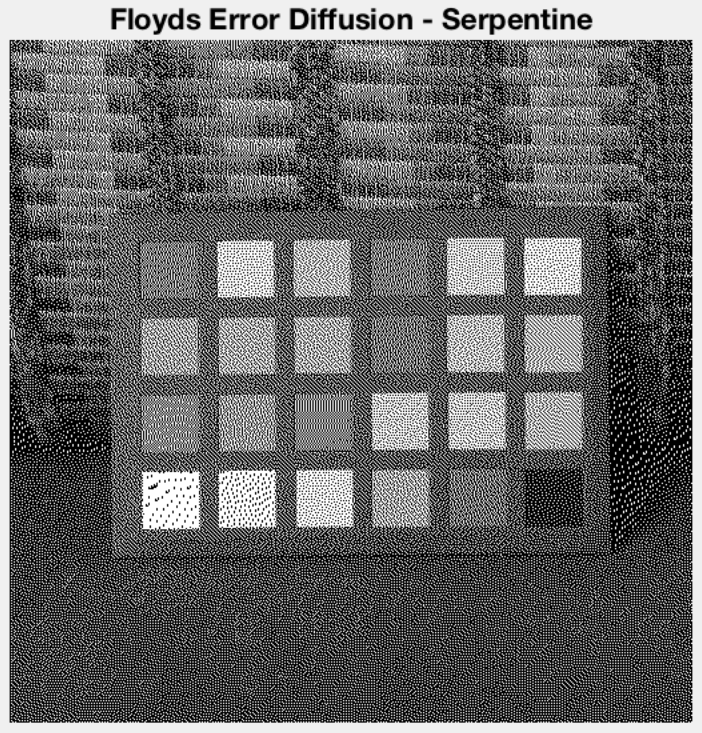
errorDiffusion() Function:

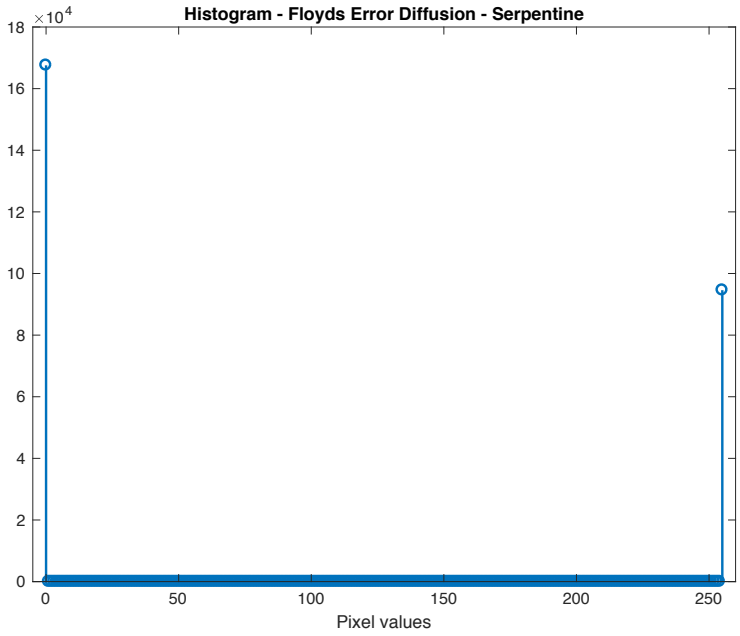
- Traverse through the image using 2 for loops
- Traverse always from left to right for all the rows
- Decide 0 or 255 pixel for the output using threshold 127
- Calculate the error as output – input pixel
- Diffuse errors to the future pixels using the corresponding matrices
- Return halftoned image matrix

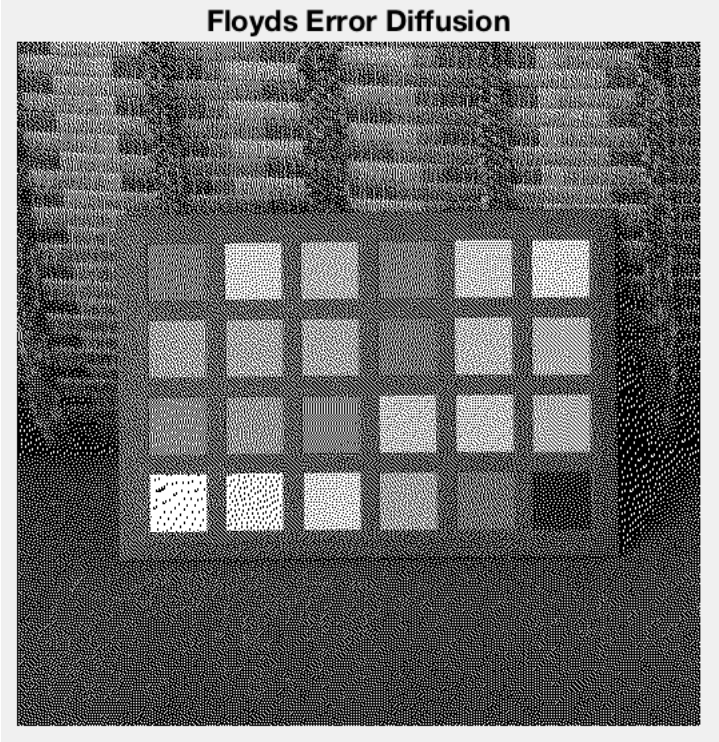
➢ EXPERIMENTAL RESULTS:

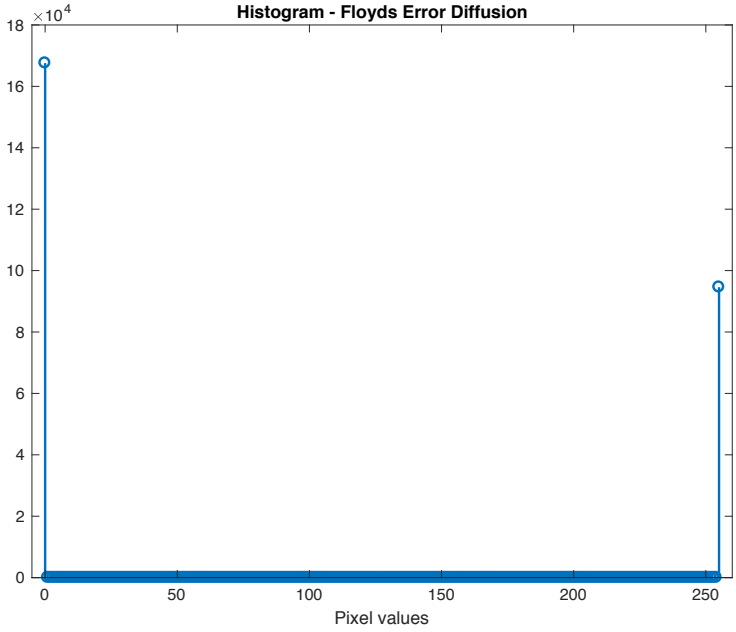
Input Colorchecker Image

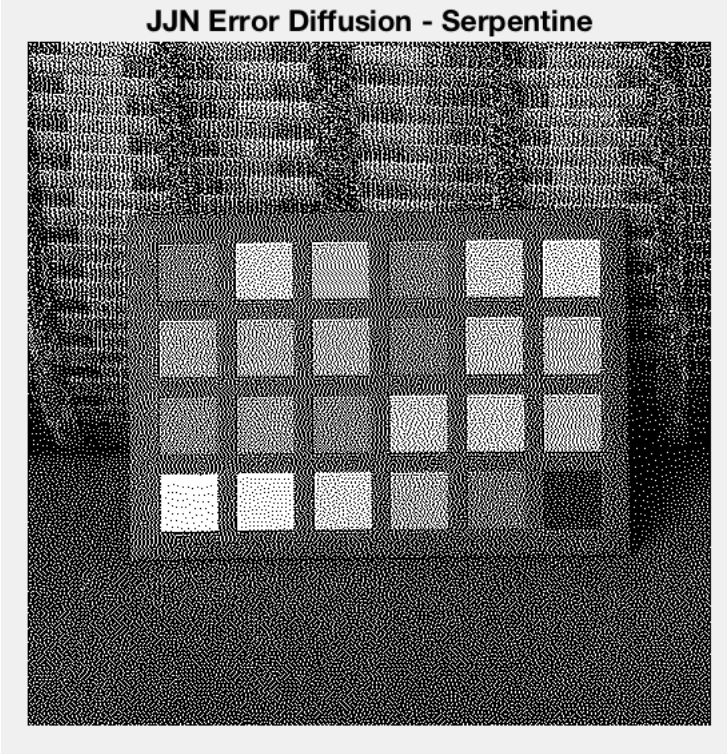


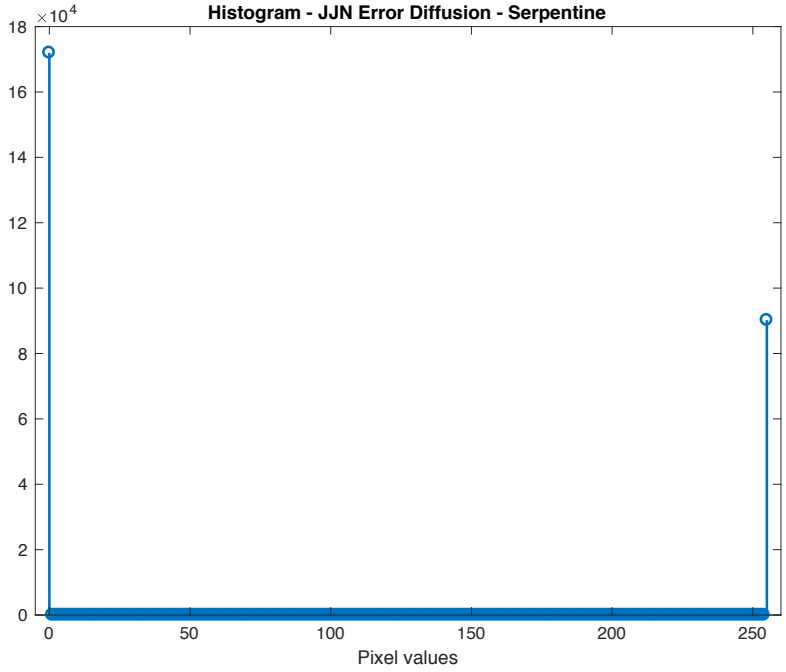

Floyds Error Diffusion - Serpentine

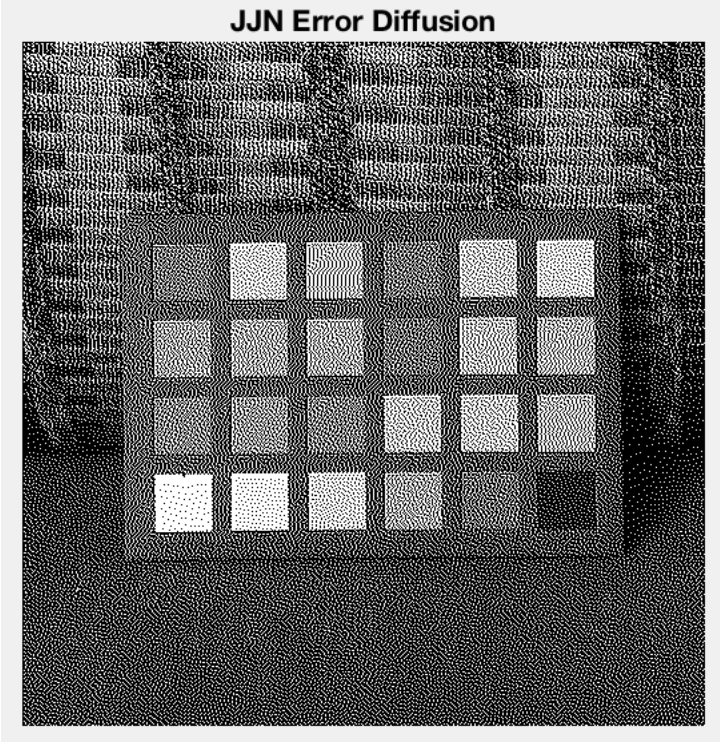

Histogram - Floyds Error Diffusion - Serpentine
×10^4
18
16
14
12
10
8
6
4
2
0
0
50
100
150
200
250
Pixel values

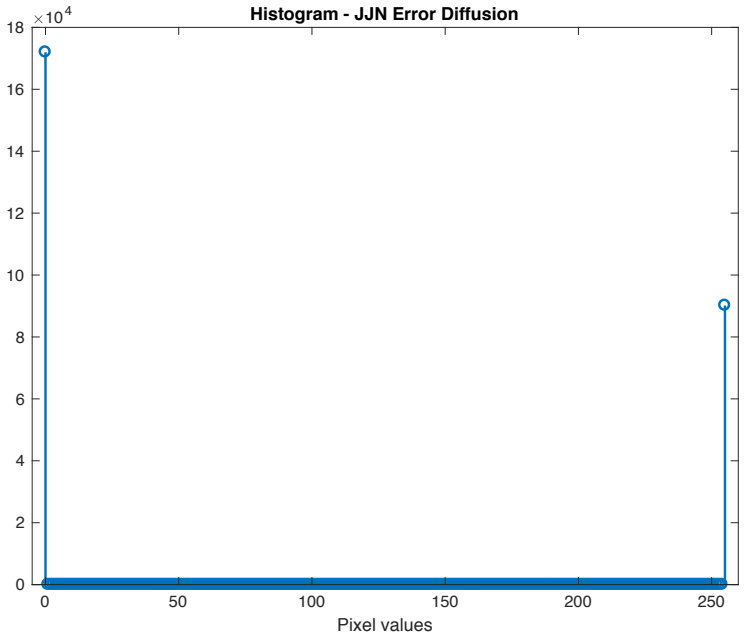

Floyds Error Diffusion

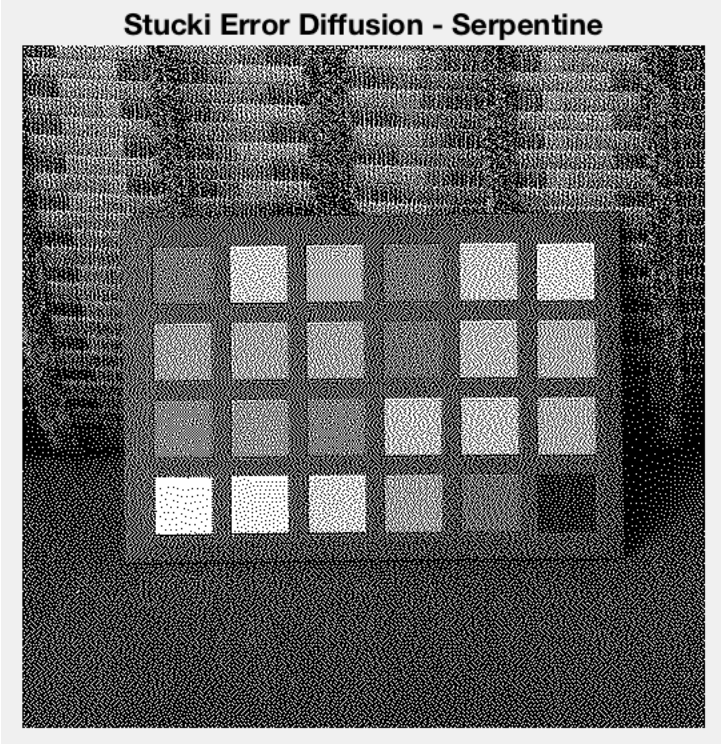

Histogram - Floyds Error Diffusion
×10^4
18
16
14
12
10
8
6
4
2
0
0
50
100
150
200
250
Pixel values

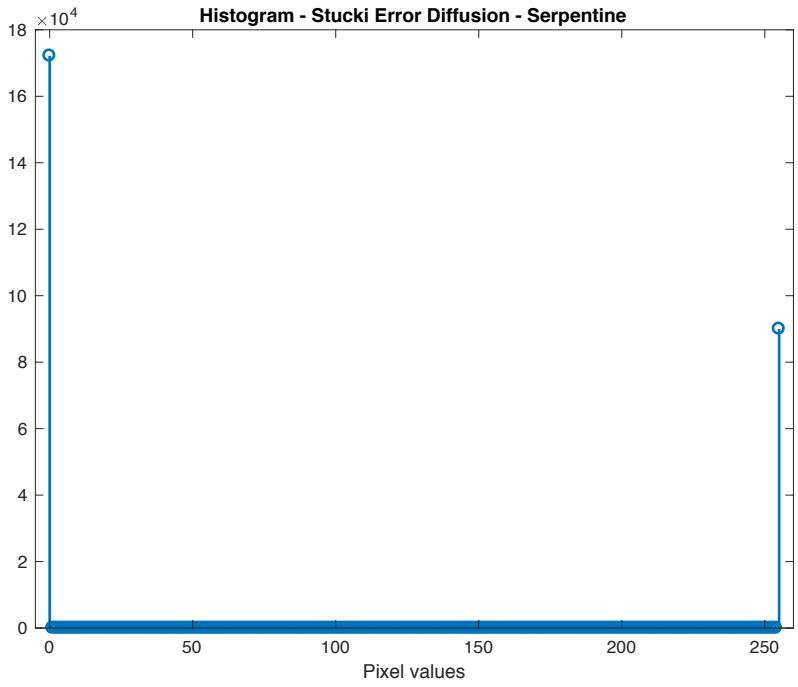

JJN Error Diffusion - Serpentine

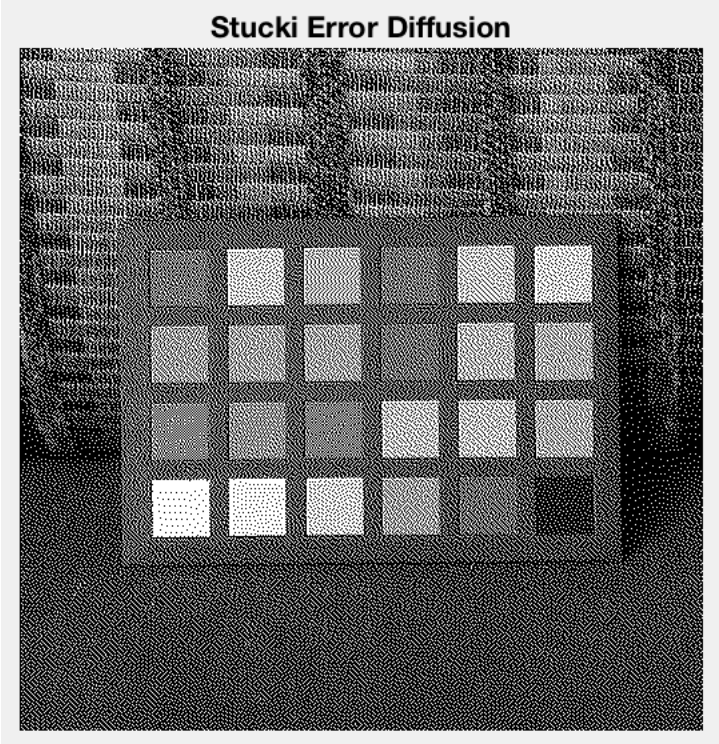

Histogram - JJN Error Diffusion - Serpentine
×10^4
18
16
14
12
10
8
6
4
2
0
0
50
100
150
200
250
Pixel values

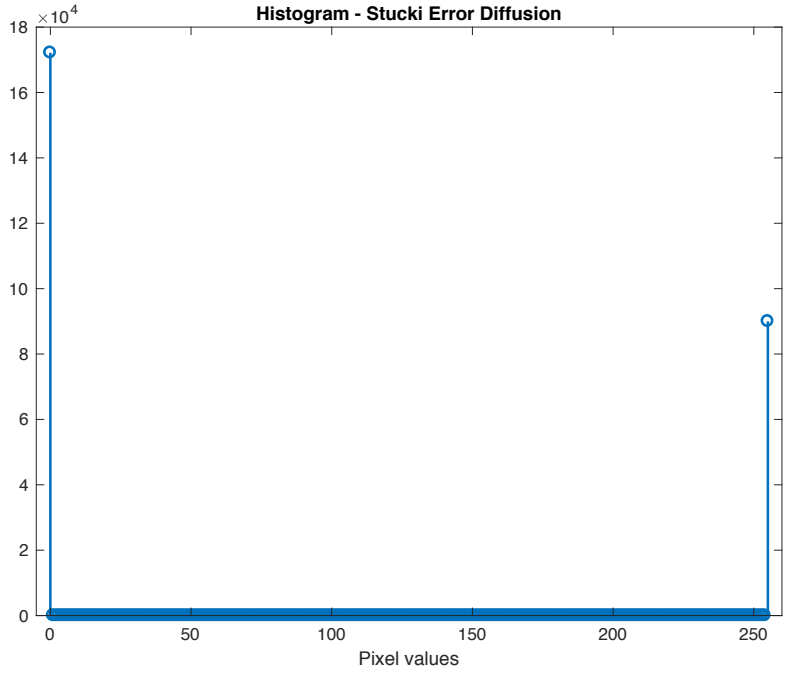

JJN Error Diffusion


Histogram - JJN Error Diffusion
×10^4
18
16
14
12
10
8
6
4
2
0
0
50
100
150
200
250
Pixel values


Stucki Error Diffusion - Serpentine


Histogram - Stucki Error Diffusion - Serpentine
×10^4
18
16
14
12
10
8
6
4
2
0
0
50
100
150
200
250
Pixel values


Stucki Error Diffusion


Histogram - Stucki Error Diffusion
×10^4
18
16
14
12
10
8
6
4
2
0
0
50
100
150
200
250
Pixel values

➢ DISCUSSION:

Thus, I have implemented all the 3 algorithms in both Serpentine and Non-Serpentine ways:

- Floyd–Steinberg dithering only diffuses the error to neighboring pixels. This results in very fine-grained dithering.
- JJN dithering diffuses the error also to pixels one step further away. The dithering looks coarser. The runtime is also slow.
- Stucki dithering is slightly faster. Its output looks clean and sharp.

There are various other methods for Halftoning like Burkes, Sierra Dithering, Two-Row Dithering, Sierra Lite and Atkinson. These methods prove to be faster, simple and clean. A comparison of the implemented algorithms and algorithms (which can improve) are presented below.

(Source: https://brucebcampbell.files.wordpress.com/2013/04/dithering-methods.pdf)

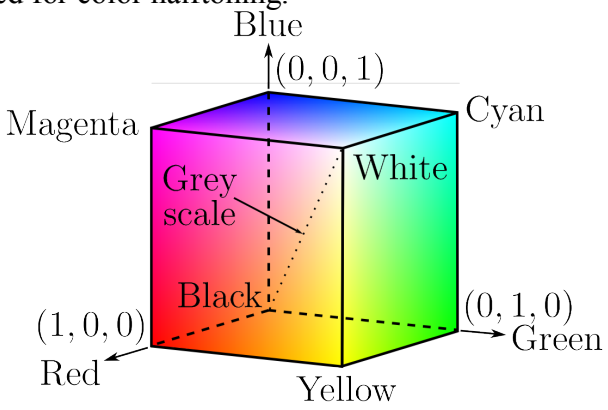
E) COLOR HALFTONING:

➢ APPROACH AND PROCEDURES:

In the previous part of the question I demonstrated what how error diffusion technique of diffusing the errors to the future pixels can be achieved for Halftoning. In this part of the question, I would explain how Error Diffusion method can be used for Color Halftoning. A simple Separable Error Diffusion method and a novel MBVQ-based Error Diffusion method are discussed below.

➢ APPROACH AND PROCEDURES:

This problem is a generalized version of Error Diffusion on grayscale images. Now, each channel in the image can be considered separately and Error Diffusion (using any method) can be performed to get color halftoned images. More generally Floyd's-Steinberg's serpentine error diffusion method is performed for color halftoning.

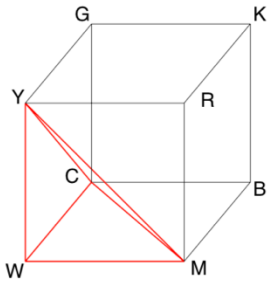


- **Separable Error Diffusion:**
  The method is to find the closest of the 8-vertices in the cube given above for every image pixel.
  - Convert RGB image to CMY image
  - Channelize the image into C, M and Y channels
  - Perform Floyd's error diffusion technique on separate C, M and Y channels
  - Combine C, M and Y channels and convert it back to RGB

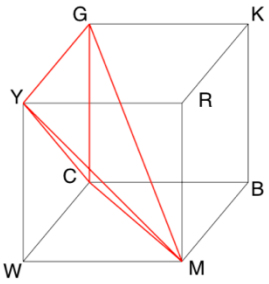
- **MBVQ-Based Error Diffusion:**
  The main difference between the Separable Error Diffusion and MBVQ-based error diffusion is that, for every image pixel, we find the closed vertex in the chosen Tetrahedron (4-vertices) instead of the usual 8-vertices cube. The tetrahedron or quadruple for every pixel is chosen based on the RGB value of the pixel. The algorithm is given below:
  - Determine the MBVQ quadrant (Tetrahedron) for a pixel based on RGB(x,y)
  - Find the vertex V of the tetrahedron closest to CMY(x,y) + e(x,y) where e(x,y) is the error that was diffused from previous seen pixel
  - Output the value of V at the location of (x,y)

- The new quantization error is calculated using CMY(x,y) + e(x,y) – V
- The error is then distributed to the future pixels using the Floyd's error diffusion method.

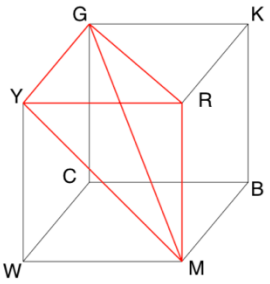
The Different Tetrahedrons (Quadruples) and the algorithm is shown below:
(Source: https://pdfs.semanticscholar.org/c67f/37a2ab36bab46bb632b65dc8dc3866f7c80e.pdf)



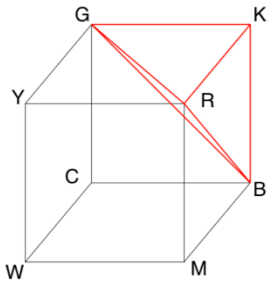
CMYW



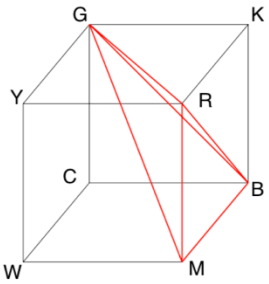
MYGC



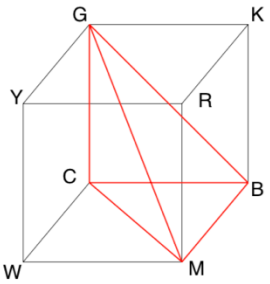
RGMY



KRGB



RGBM



CMGB

```
{
    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;
}
```

Algorithm Implemented (C++): (Separable Error Diffusion)
main() Function:

- Read given *flower.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Allocate memories for output Images using *allocMemory3D()*
- Initialize Floyd's error diffusion matrix
- Get C, M and Y values using R,G and B values of the flower image
- Halftone the C, M and Y *errorDiffusionSerpentine()* and the diffusion matrix
- Combine C, M and Y channels into 3D matrix
- Retrace it back to RGB channel
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory3D()*

errorDiffusionSerpentine() Function:

- Traverse through the image using 2 for loops
- If the row index mod 2 is zero, traverse from left to right in the column
- If the row index mod 2 is non-zero, traverse from right to left in the column
- Decide 0 or 255 pixel for the output using threshold 127
- Calculate the error as output – input pixel
- Diffuse errors to the future pixels using the corresponding matrices
- Return halftoned image matrix

Algorithm Implemented (C++): (MBVQ-based Error Diffusion)
main() Function:

- Read given *flower.raw* image using *fileRead()* function
- Convert 1D image to 3D using *image1Dto3D()* function
- Allocate memories for output Images using *allocMemory3D()*
- Initialize Floyd's error diffusion matrix
- Initialize MBVQ Quadruple vectors
- Initialize C, M, Y, R, G, B, K, W coordinates
- Halftone the C, M and Y *errorDiffusionSerpentine()* and the diffusion matrix
- Combine C, M and Y channels into 3D matrix
- Retrace it back to RGB channel
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()*

errorDiffusionSerpentine() Function:

- Traverse through the image using 2 for loops
- If the row index mod 2 is zero, traverse from left to right in the column
- If the row index mod 2 is non-zero, traverse from right to left in the column
- Get R, G and B values for every pixel location
- Get the quadruple using *getMBVQString()*
- Get closest vertex belonging to MBVQ vertex using *getMinDistVertex()* using CMY value
- Calculate the error as output – input pixel (All in CMY channels)

- Diffuse errors to the future pixels using the corresponding matrices
- Return halftoned image matrix

getMBVQString() Function:

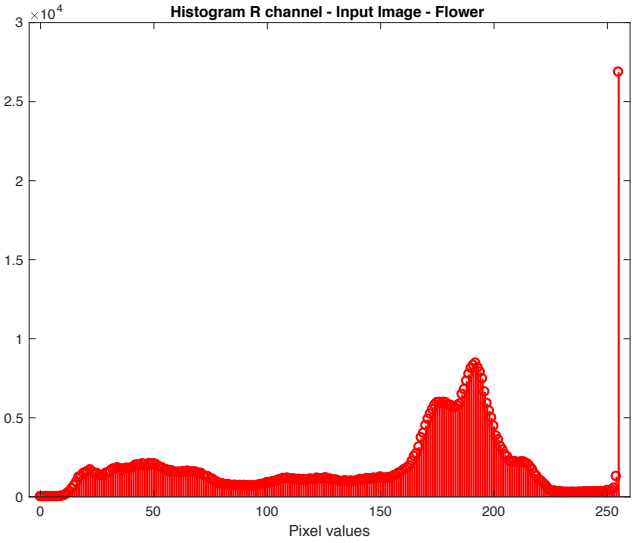
- Get R, G and B values
- Using the given algorithm, decide which quadrupole the pixel belongs to
- Return the decided string

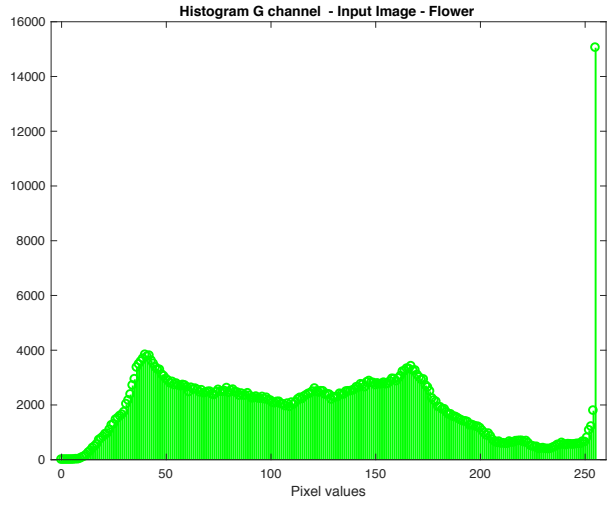
getMinDistVertex() Function:

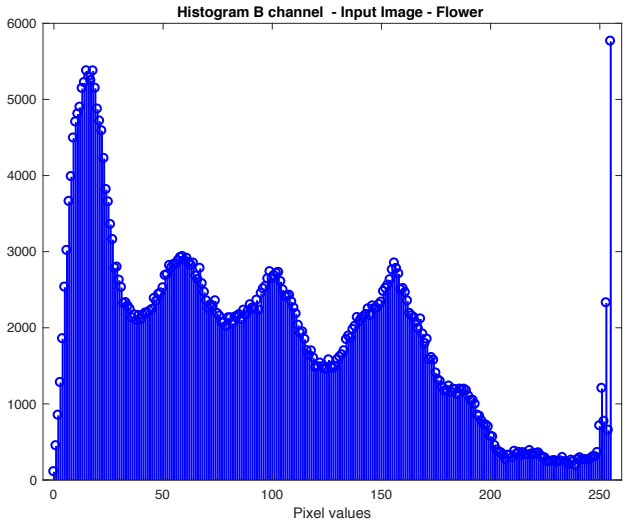
- Get C, M and Y values
- Normalize the values
- Find the Euclidian distance of the pixel from four vertices of the tetrahedron
- Select the vertex which corresponds to minimum distance
- Return vertex V

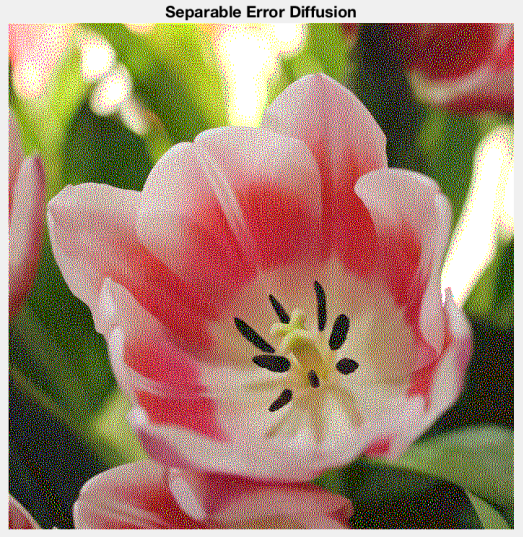
➢ EXPERIMENTAL RESULTS:

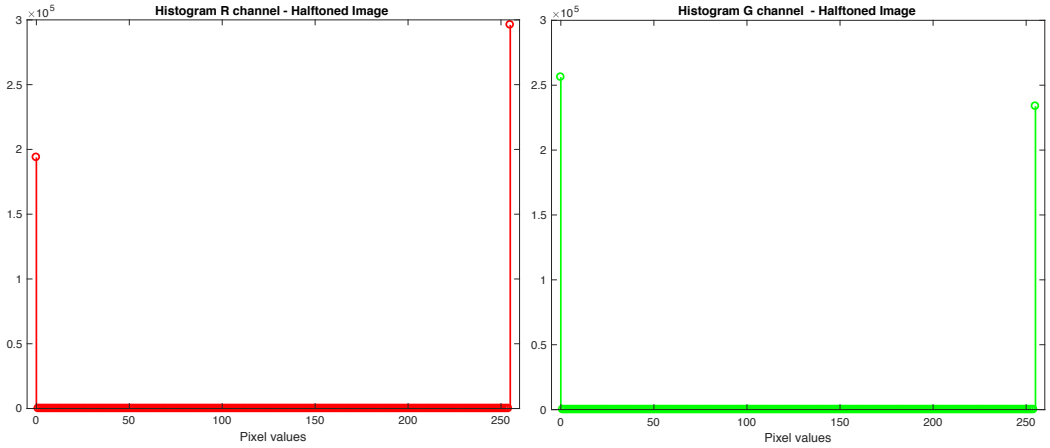
Input Image - Flower

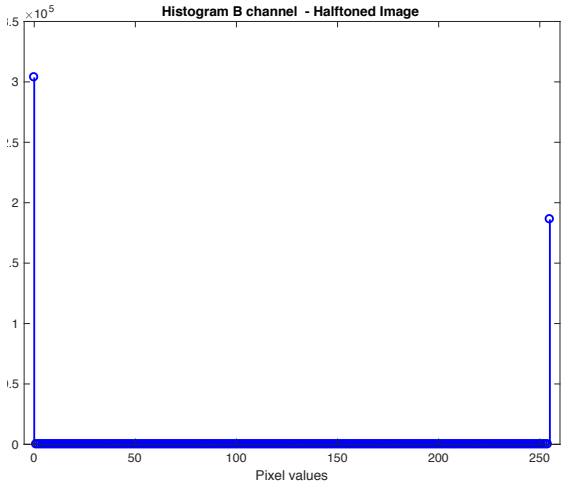

Histogram R channel - Input Image - Flower
×10^4
3
2.5
2
1.5
1
0.5
0
0
50
100
150
200
250
Pixel values

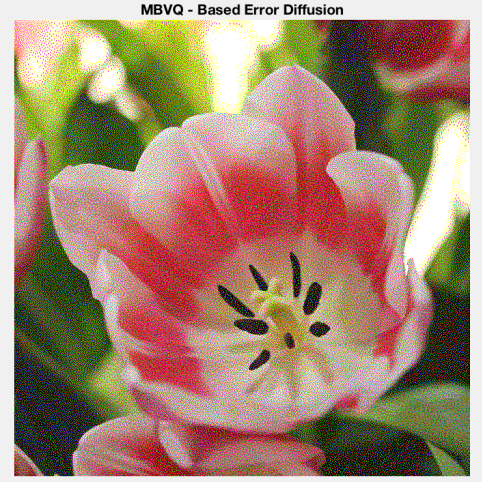

Histogram G channel - Input Image - Flower
16000
14000
12000
10000
8000
6000
4000
2000
0
0
50
100
150
200
250
Pixel values

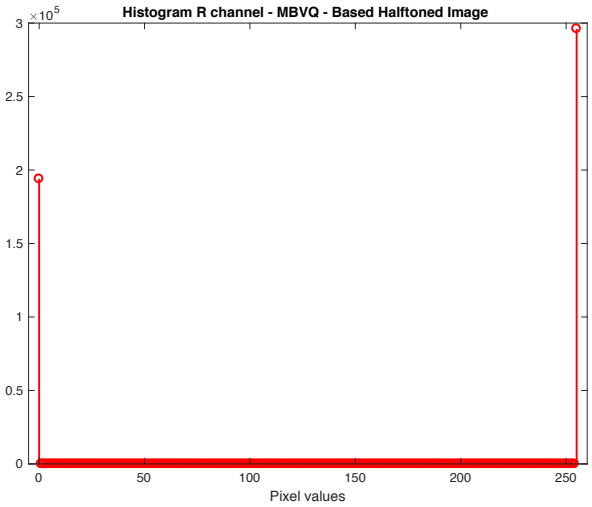

Histogram B channel - Input Image - Flower
6000
5000
4000
3000
2000
1000
0
0
50
100
150
200
250
Pixel values

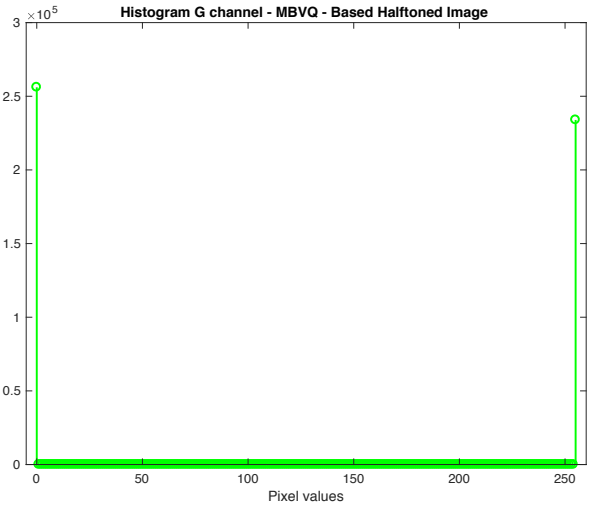

Separable Error Diffusion

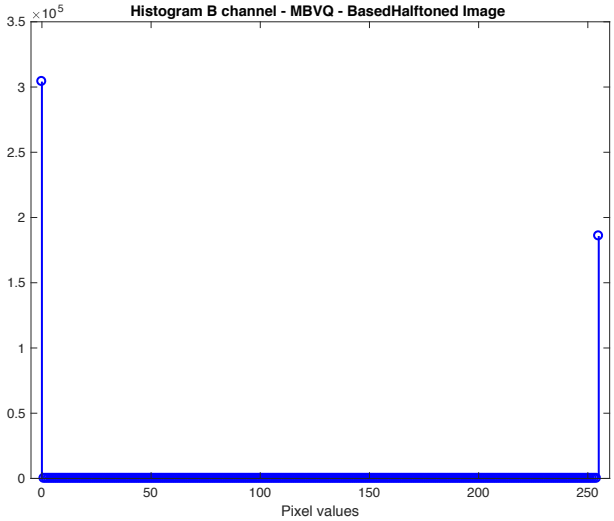

Histogram R channel - Halftoned Image
Histogram G channel - Halftoned Image
Pixel values
Pixel values


Histogram B channel - Halftoned Image
Pixel values


MBVQ - Based Error Diffusion


Histogram R channel - MBVQ - Based Halftoned Image
×10^5
3
2.5
2
1.5
1
0.5
0
0
50
100
150
200
250
Pixel values


Histogram G channel - MBVQ - Based Halftoned Image
×10^5
3
2.5
2
1.5
1
0.5
0
0
50
100
150
200
250
Pixel values


Histogram B channel - MBVQ - BasedHalftoned Image
×10^5
3.5
3
2.5
2
1.5
1
0.5
0
0
50
100
150
200
250
Pixel values

➢ DISCUSSION:

I implemented 2 different methods of Color Halftoning with Error Diffusion.

- The main difference between Separable Error Diffusion and MBVQ Based Error Diffusion is that, Separable Error Diffusion finds the closest vertex of all the 8 vertices in the cube.
- Whereas, in MBVQ based Error Diffusion, the closest vertex in a tetrahedron (out of 4 vertices) is found as the output pixel
- The Error diffusion mechanism is same in both the methods and both of them use CMY channels for Color Halftoning
- From the results above, though at an outer look both the images look similar, MBVQ-based color halftoned image looks best in terms of preserving color and brightness
- As the name suggests, MBVQ based Error Diffusion method reduces Halftone noise. Also selects output vertex in a manner that there is minimal brightness variation
- The high frequency pattern in the flower petals are well preserved in the MBVQ-based Error Diffusion method than basic Separable Error Diffusion method

(All the problem 2 outputs look a little different when copy pasted in MS Word (Report) Please look into the submitted raw files if the outputs don't look good in the report! Appearances are deceiving!)

PROBLEM 3
MORPHOLOGICAL PROCESSING

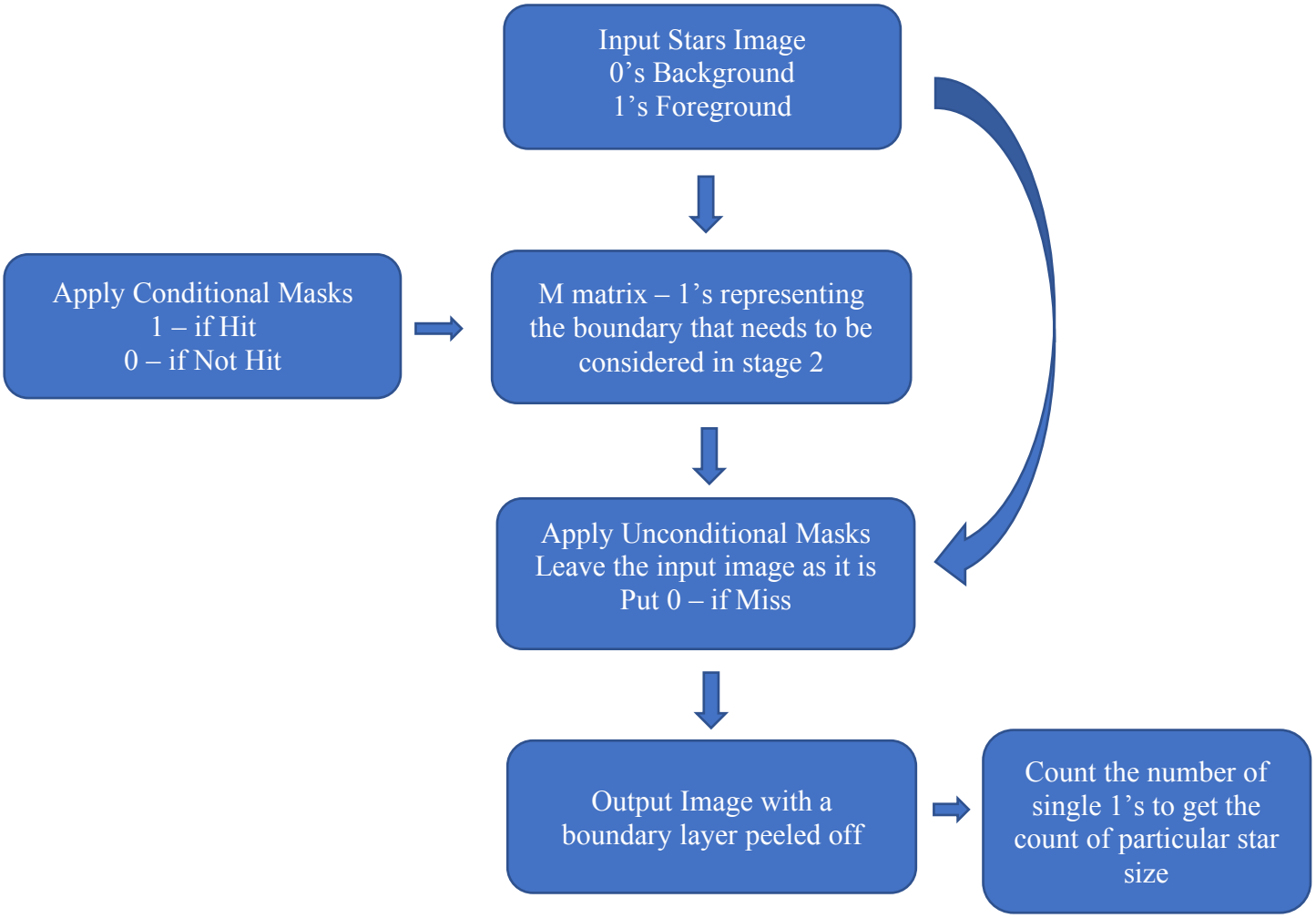
F) SHRINKING:

➢ ABSTRACT AND MOTIVATION:

Morphological Image Processing is a collection of certain non-linear operations related to the shape, structure or morphology of features in an image. All operations related to morphology are done only in relation with the order of the pixels and not the pixel intensity itself. Also, morphological operations are usually done considering '0' as the background and '1' as the foreground. In this problem, I would be implementing Shrinking, which is a simple morphological operation which is very "Extreme" too. I mention this Extreme because, it reduces a solid image to a single point. Using this morphological operation, I would be counting the number of stars in the given image.

➢ APPROACH AND PROCEDURES:

I used the Shrinking – Conditional and Unconditional pattern masks to shrink every star in the given image to a single point. A single point means, a single '1' to represent every star. When I apply Shrinking Conditional Masks, the output would be boundary layers that needed to be peeled off in the second stage. The second stage is where the boundaries are peeled off and a star with a reduced boundary is given as the output.



The Conditional Mask Patterns are given below:
(Source: Digital Image Processing by William K. Pratt – pg 433)

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

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

The Unconditional Mask Patterns are given below:
(Source: Digital Image Processing by William K. Pratt – pg 435)

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																		

Algorithm Implemented (C++):

main() Function:

- Read given *stars.raw* image using *fileRead()* function
- Convert 1D image to 2D using *image1Dto2D()* function
- Allocate memory for Binary image, 1st stage M image and final output using *allocMemory2D()*
- Convert the grayscale image using fixed thresholding. The threshold chosen was 127.
- Get first stage intermediate matrix using *returnFirstStage()* function
- Get second stage output using returnSecondStage() function
- Iteratively call the first stage and second stage to get the desired output
- The desired output is when there is no change to the image and it saturates
- Count the number of stars in every iteration.
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()*

returnFirstStage() Function:

- Initialize the intermediate matrix with all zeros
- Mask size is fixed as 3
- Traverse through the image using 2 for loops
- Get every single pixel if it is 0 or 1
- If it's 0 – do nothing
- If it's 1 – calculate the bond value and compare the 3*3 neighbor of that pixel with all the bond's corresponding Conditional Masks
- If it's a hit, put 1 in the Intermediate matrix. If 0 put zero
- Return the intermediate matrix

returnSecondStage() Function:

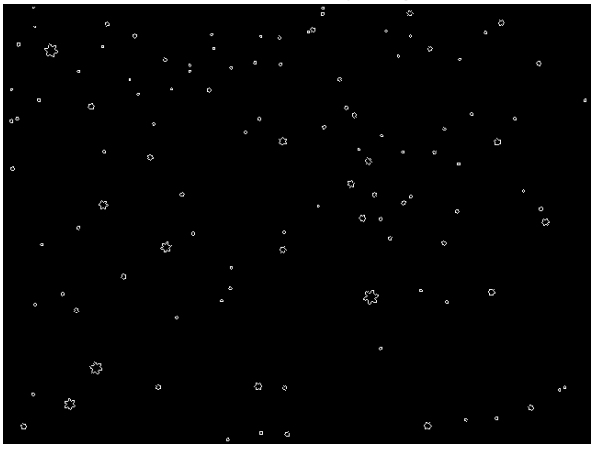
- Get the input matrix along with Intermediate Matrix
- Mask size is fixed as 3
- Traverse through the image using 2 for loops
- Get every single pixel if it is 0 or 1
- If it's 0 – copy input to output
- If it's 1 – compare the 3*3 neighbor of that pixel with all the Conditional Masks
- If it's a hit – copy input to output
- If it's a miss – put 0 in the output pixel value
- Return the output image

compareWithMask() Function:

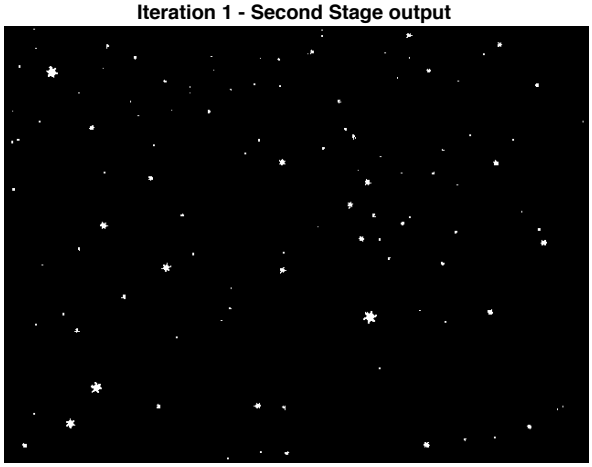
- Check every element of the 3*3 neighbor with every element of the mask
- If one element doesn't match, break
- Iterate through all the masks
- If one mask completely matches, break
- Return hit/miss output

➢ EXPERIMENTAL RESULTS:

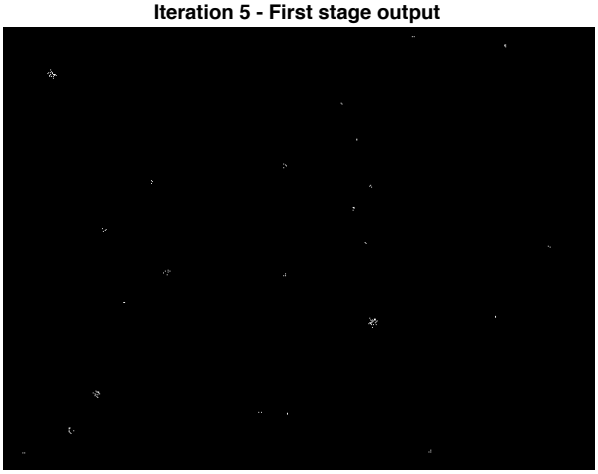
Iteration 1- First stage output



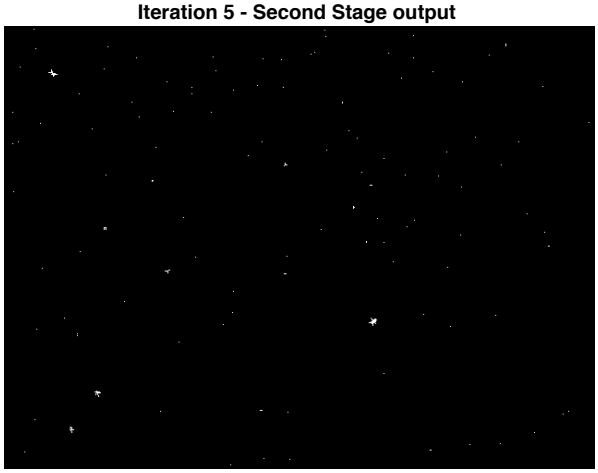
Iteration 1 - Second Stage output



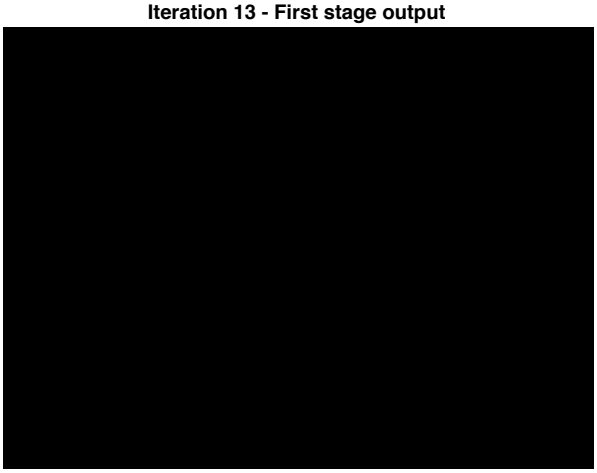
Iteration 5 - First stage output



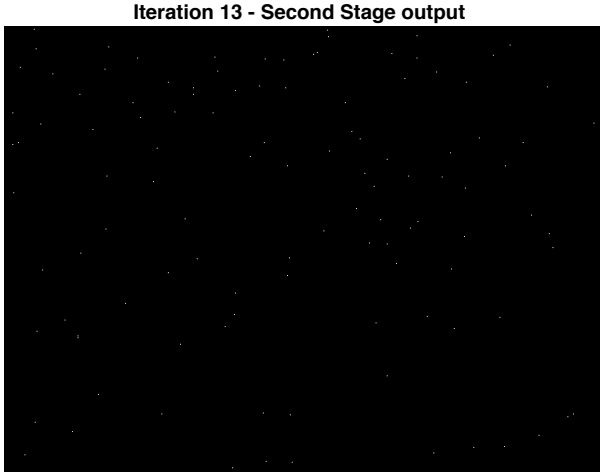
Iteration 5 - Second Stage output



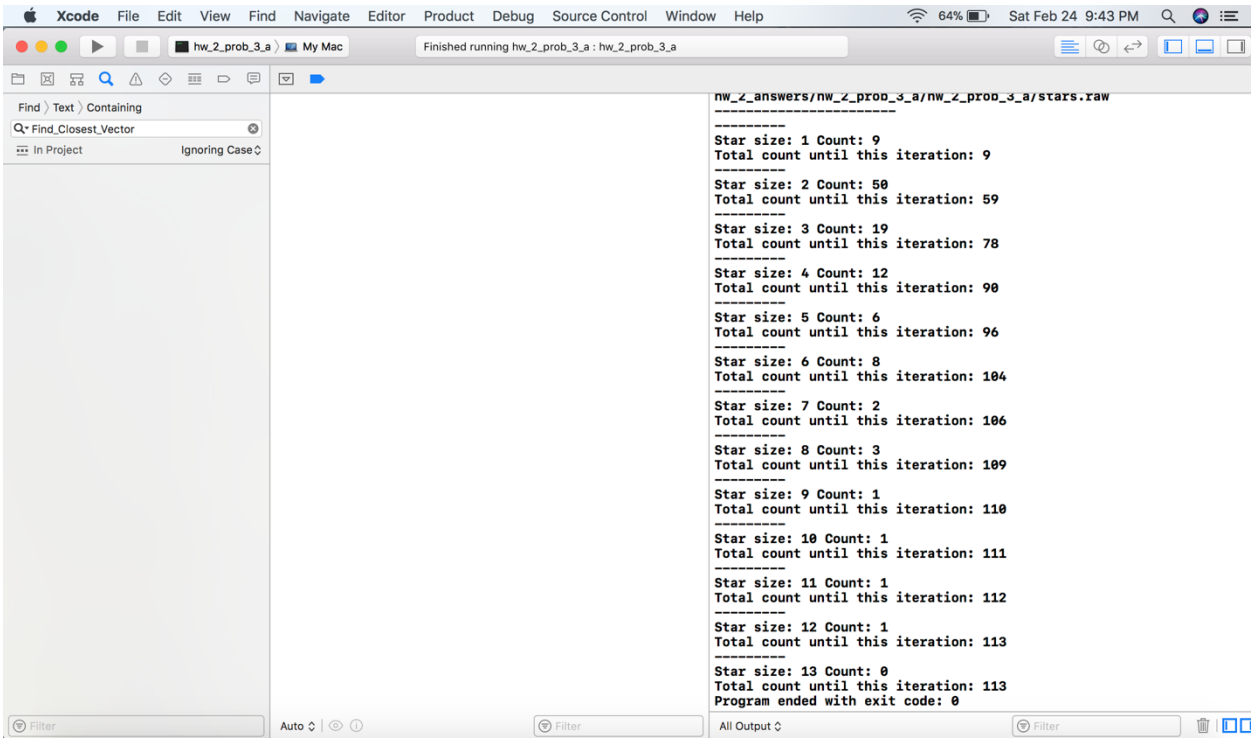
Iteration 13 - First stage output



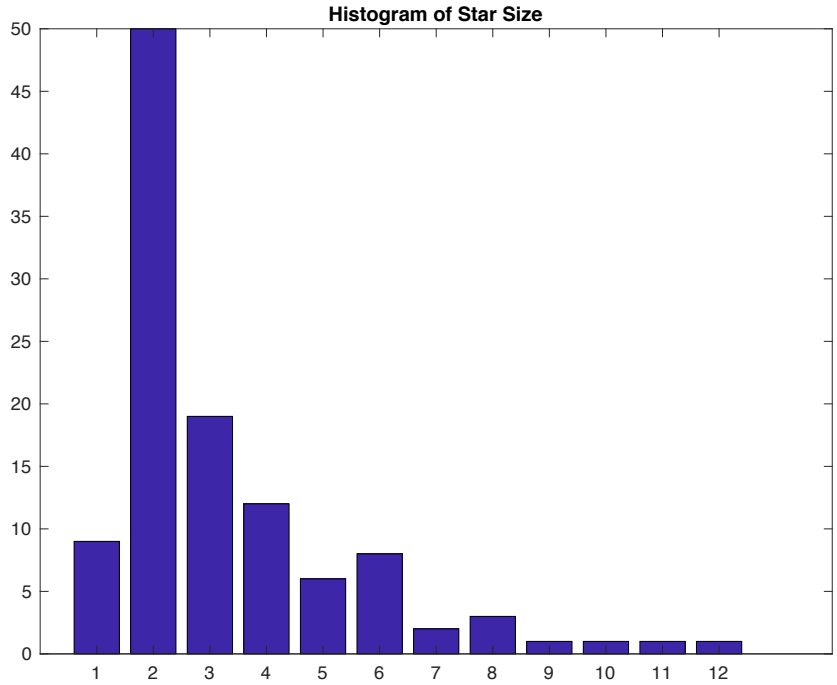
Iteration 13 - Second Stage output

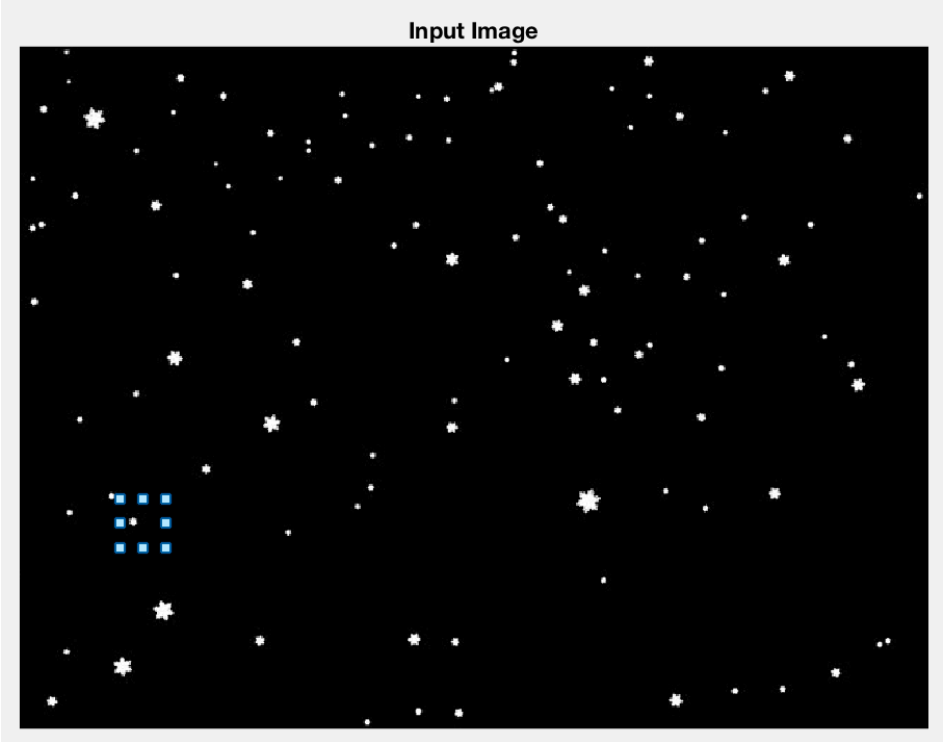


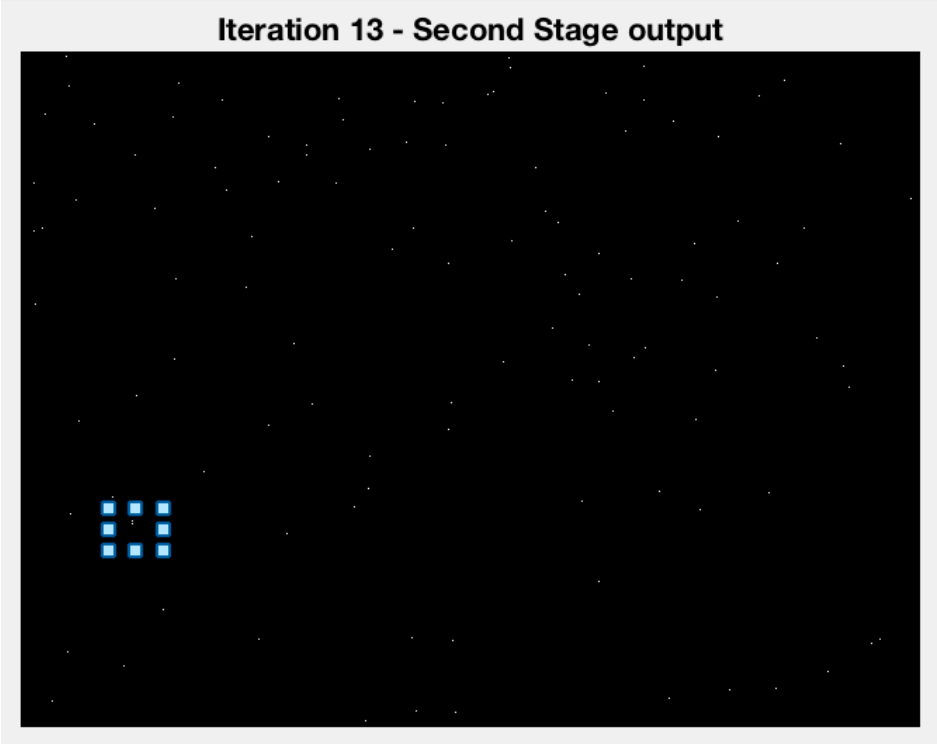
The first stage outputs give the boundary of the stars to be peeled off. Second stage output is the peeled off one. Star size reduces by 1 at the end of every iteration.



The star size is defined as the number of iterations it needs to get reduced to 1. The above output is the star size and count of that. The actual number of stars in the input image are 112. But I am getting 113 count because, one star is getting split into two with this method.







If not for this breaking of stars, I would get **112 Stars** as count!

➢ **DISCUSSION:**

- Thus, Shrinking Morphological operation was done using the above-mentioned procedures. One application of Shrinking is that they are used in counting desired objects, often combined with other set of morphological operations
- And I counted **112 stars** with this operation. The Conditional and Unconditional masks were helpful in deciding the layers to be peeled off
- Step by step the star size got reduced beautifully
- Different star size and count of each of them were also reported with the screenshot of the output (Star size being defined as the number of iterations to reduce to a single point).
- Iterations were stopped when the star count remained the same and the first stage output was NULL

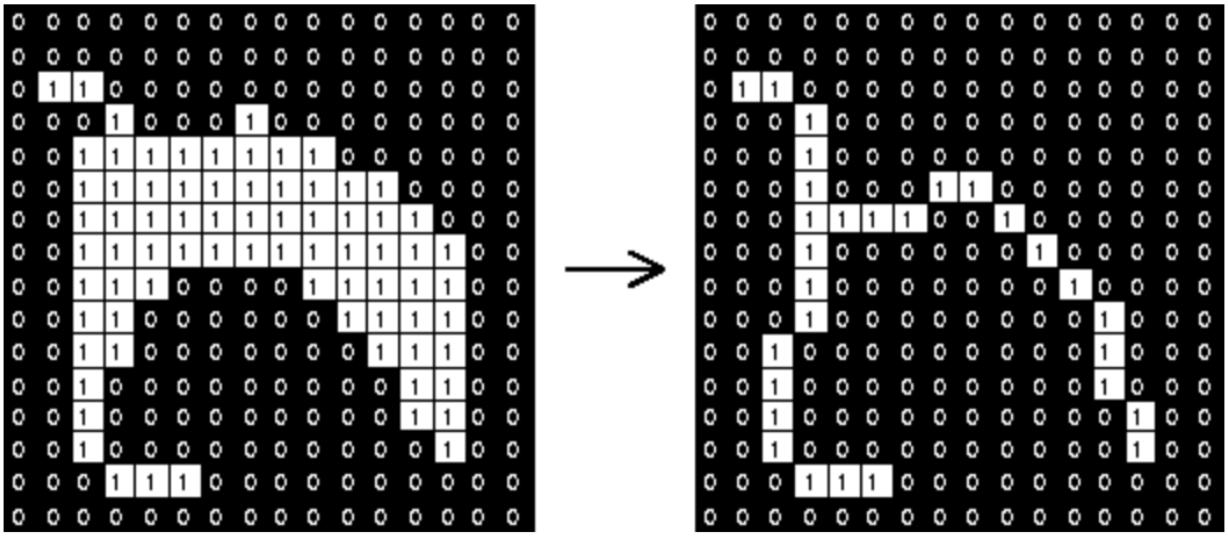
G) THINNING:

➢ **ABSTRACT AND MOTIVATION:**

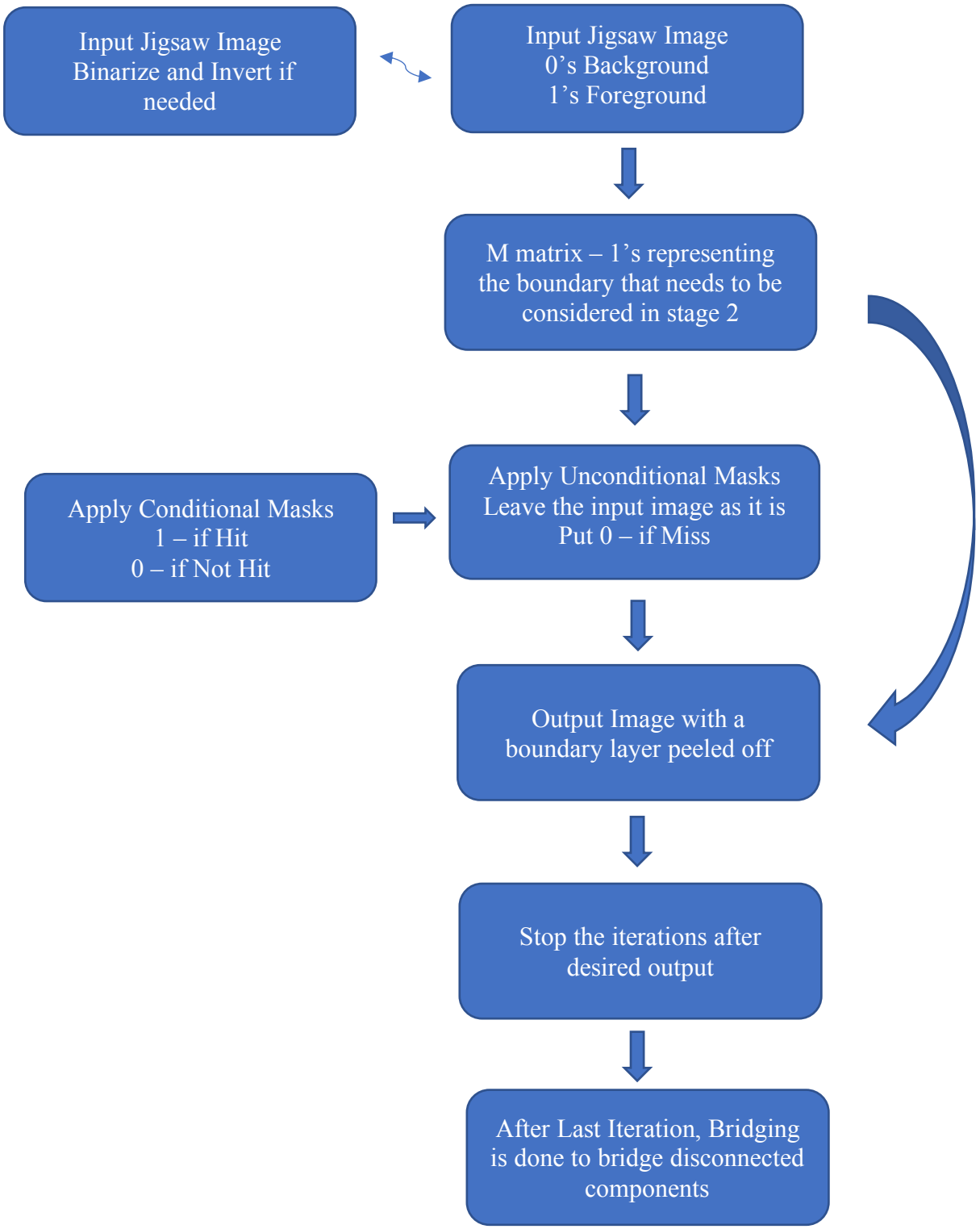
I implemented Shrinking operation in the previous question. In this part of the question, I would implement Thinning – one another morphological operation. This is an operation which is not so extreme like Shrinking, but also does not give the skeleton of the image. The given input Jigsaw image is considered for Thinning operation. Similar to previous morphological operation, here also 0's are considered as the background and 1's are considered as foreground.

➢ **APPROACH AND PROCEDURES:**

The Thinning operation, is related to hit and miss transform based on Conditional and Unconditional masks that are given in the table below. The structure elements are passed through the image pixels and a thin representation of the underlying Foreground image is visualized. An example of Thinning on a simple binary image is shown below.
(Source: https://homepages.inf.ed.ac.uk/rbf/HIPR2/thin.htm)



A Flow chart of the algorithm I followed is given below. The main difference between Shrinking and Thinning are the **masks**. Appropriate masks are identified for Thinning (Note that they vary for every morphological operation). Also, an extra block of **Bridging** was performed. This was done to connect small disconnected components in the Thinning process.



The Conditional Mask Patterns are given below:
(Source: Digital Image Processing by William K. Pratt – pg 433)

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

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

The Unconditional Mask Patterns are given below:
(Source: Digital Image Processing by William K. Pratt – pg 435)

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												

More Details on Bridging:

At the conclusion of last iteration, One step of Bridging was performed to avoid the unnecessary disjoints in the output.

(Source: Digital Image Processing by William K. Pratt – pg 426)

Bridge. Create a black pixel if creation results in connectivity of previously unconnected neighboring black pixels.

$$G(j,k) = X \cup [P_1 \cup P_2 \cup \cdots \cup P_6] \tag{14.2-4a}$$

where

$$P_1 = \bar{X}_2 \cap \bar{X}_6 \cap [X_3 \cup X_4 \cup X_5] \cap [X_0 \cup X_1 \cup X_7] \cap \bar{P}_Q \tag{14.2-4b}$$

$$P_2 = \bar{X}_0 \cap \bar{X}_4 \cap [X_1 \cup X_2 \cup X_3] \cap [X_5 \cup X_6 \cup X_7] \cap \bar{P}_Q \tag{14.2-4c}$$

$$P_3 = \bar{X}_0 \cap \bar{X}_6 \cap X_7 \cap [X_2 \cup X_3 \cup X_4] \tag{14.2-4d}$$

$$P_4 = \bar{X}_0 \cap \bar{X}_2 \cap X_1 \cap [X_4 \cup X_5 \cup X_6] \tag{14.2-4e}$$

$$P_5 = \bar{X}_2 \cap \bar{X}_4 \cap X_3 \cap [X_0 \cup X_6 \cup X_7] \tag{14.2-4f}$$

$$P_6 = \bar{X}_4 \cap \bar{X}_6 \cap X_5 \cap [X_0 \cup X_1 \cup X_2] \tag{14.2-4g}$$

and

$$P_Q = L_1 \cup L_2 \cup L_3 \cup L_4 \tag{14.2-4h}$$

$$L_1 = \bar{X} \cap \bar{X}_0 \cap X_1 \cap \bar{X}_2 \cap X_3 \cap \bar{X}_4 \cap \bar{X}_5 \cap \bar{X}_6 \cap \bar{X}_7 \tag{14.2-4i}$$

$$L_2 = \bar{X} \cap \bar{X}_0 \cap \bar{X}_1 \cap \bar{X}_2 \cap X_3 \cap \bar{X}_4 \cap X_5 \cap \bar{X}_6 \cap \bar{X}_7 \tag{14.2-4j}$$

$$L_3 = \bar{X} \cap \bar{X}_0 \cap \bar{X}_1 \cap \bar{X}_2 \cap \bar{X}_3 \cap \bar{X}_4 \cap X_5 \cap \bar{X}_6 \cap X_7 \tag{14.2-4k}$$

$$L_4 = \bar{X} \cap \bar{X}_0 \cap X_1 \cap \bar{X}_2 \cap \bar{X}_3 \cap \bar{X}_4 \cap \bar{X}_5 \cap \bar{X}_6 \cap X_7 \tag{14.2-4l}$$

Algorithm Implemented (C++):

main() Function:

- Read given *Jigsaw_1.raw* image using *fileRead()* function
- Convert 1D image to 2D using *image1Dto2D()* function
- Allocate memory for Binary image, 1st stage M image and final output using *allocMemory2D()*
- Convert the grayscale image to Binary using fixed thresholding. A threshold of 127 was chosen for this Binary Conversion
- Invert the image to have 0's in background and 1 in foreground
- Get first stage intermediate matrix using *returnFirstStage()* function
- Get second stage output using *returnSecondStage()* function
- Iteratively call the first stage and second stage to get the desired output
- The desired output is when there is no change to the image and it saturates
- Bridging is performed to join the disjoint lines in output using *returnBridgedImage()*
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()*

returnFirstStage() Function:

- Initialize the intermediate matrix with all zeros
- Mask size is fixed as 3
- Traverse through the image using 2 for loops
- Get every single pixel if it is 0 or 1
- If it's 0 – do nothing
- If it's 1 – calculate the bond value and compare the 3*3 neighbor of that pixel with all the bond's corresponding Conditional Masks
- If it's a hit, put 1 in the Intermediate matrix. If 0 put zero
- Return the intermediate matrix

returnSecondStage() Function:

- Get the input matrix along with Intermediate Matrix
- Mask size is fixed as 3
- Traverse through the image using 2 for loops
- Get every single pixel if it is 0 or 1
- If it's 0 – copy input to output
- If it's 1 – compare the 3*3 neighbor of that pixel with all the Conditional Masks
- If it's a hit – copy input to output
- If it's a miss – put 0 in the output pixel value
- Return the output image

compareWithMask() Function:

- Check every element of the 3*3 neighbor with every element of the mask
- If one element doesn't match, break
- Iterate through all the masks

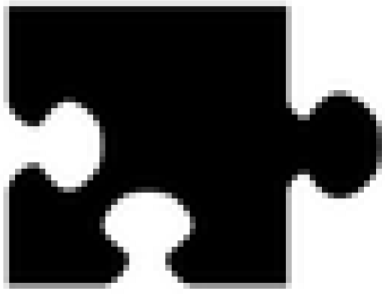
- If one mask completely matches, break
- Return hit/miss output

returnBridgedImage() Function:

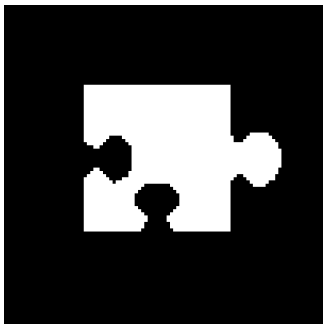
- Traverse through the final iteration image using 2 for loops
- Get X, X0, X1, X2, X3, X4, X5, X6 and X7
- X is the center pixel and all others are surrounding neighbor pixels
- Calculate L1, L2, L3, L4, P1, P2, P3, P4, P5 and P6 using the above mentioned Logical Operations
- Calculate the pixel intensity value using the above variables
- Return the Bridged Image

➢ EXPERIMENTAL RESULTS:

Input Image - Jigsaw $_1$

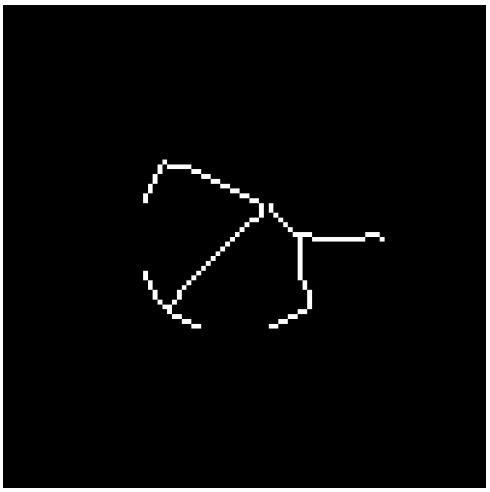


Jigsaw$_1$ - Binary Image

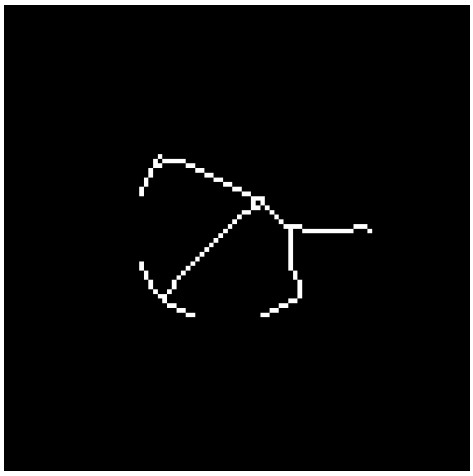


The first Image is the given Grayscale Image. Second one is the Binary and Inverted Image. Inversion here means inverting the background and foreground pixels, so as to have foreground as the object of interest.

Jigsaw$_1$ - Thinning Without Bridging



Jigsaw$_1$ - Thinning with Bridging



Jigsaw$_1$ - Thinning With Bridging

➢ DISCUSSION:

- The Outputs of Thinning operations are given above.
- First one is without Bridging. We can see that there are 2 connected components and is broken in the middle.
- The second one is with Bridging. Now all the points are connected and there are no disjoints.
- The third image is with 0's as Foreground and 1's as background. (White background and dark foreground). Just to be on the same track as the input image
- Thus Thinning was performed on a given Jigsaw image and various outputs were seen above
- Thinning is often compared with many other sets of morphological operations. This would help us understand the structure of the object of interest better.
- Example application is automatic detection of handwriting. Thinning operation might give a better understanding of the written/scribled letter

H) SKELETONIZING:

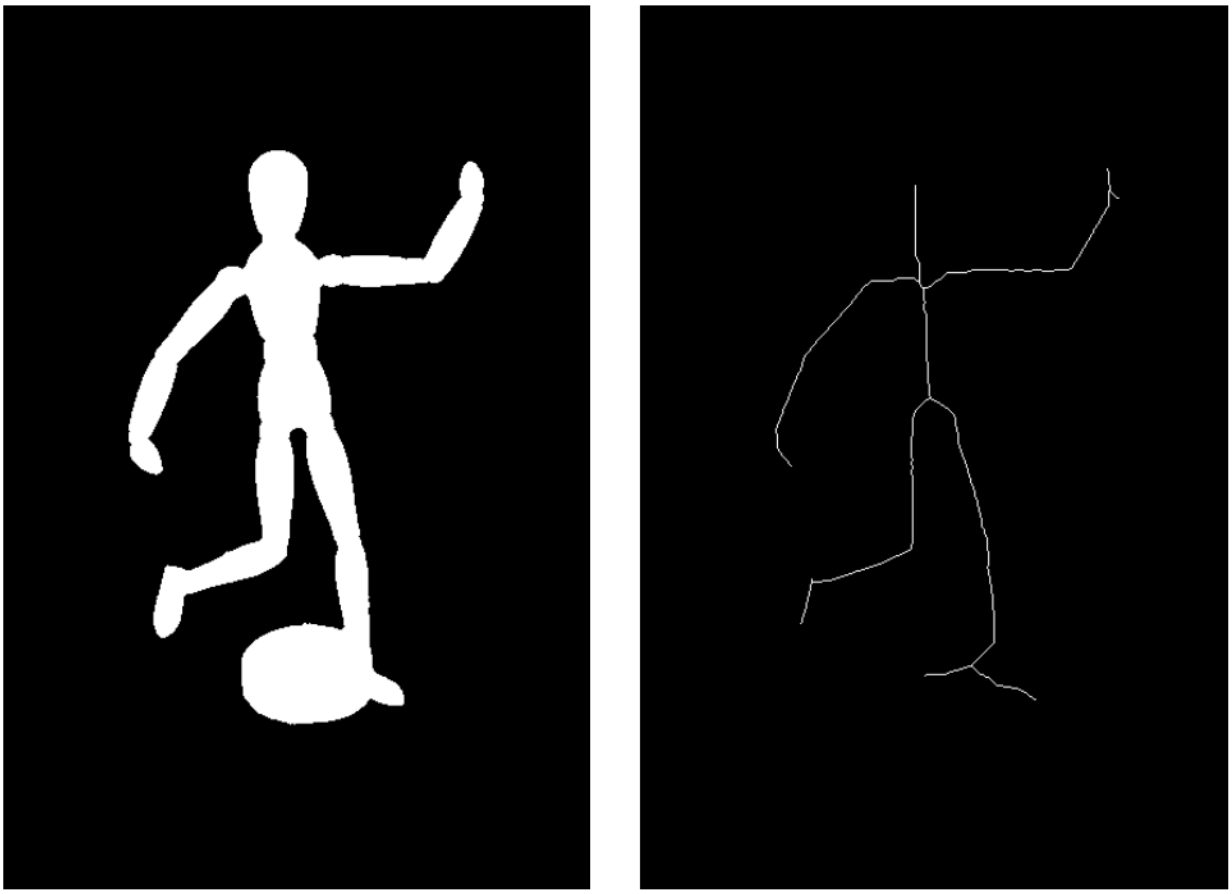
➢ ABSTRACT AND MOTIVATION:

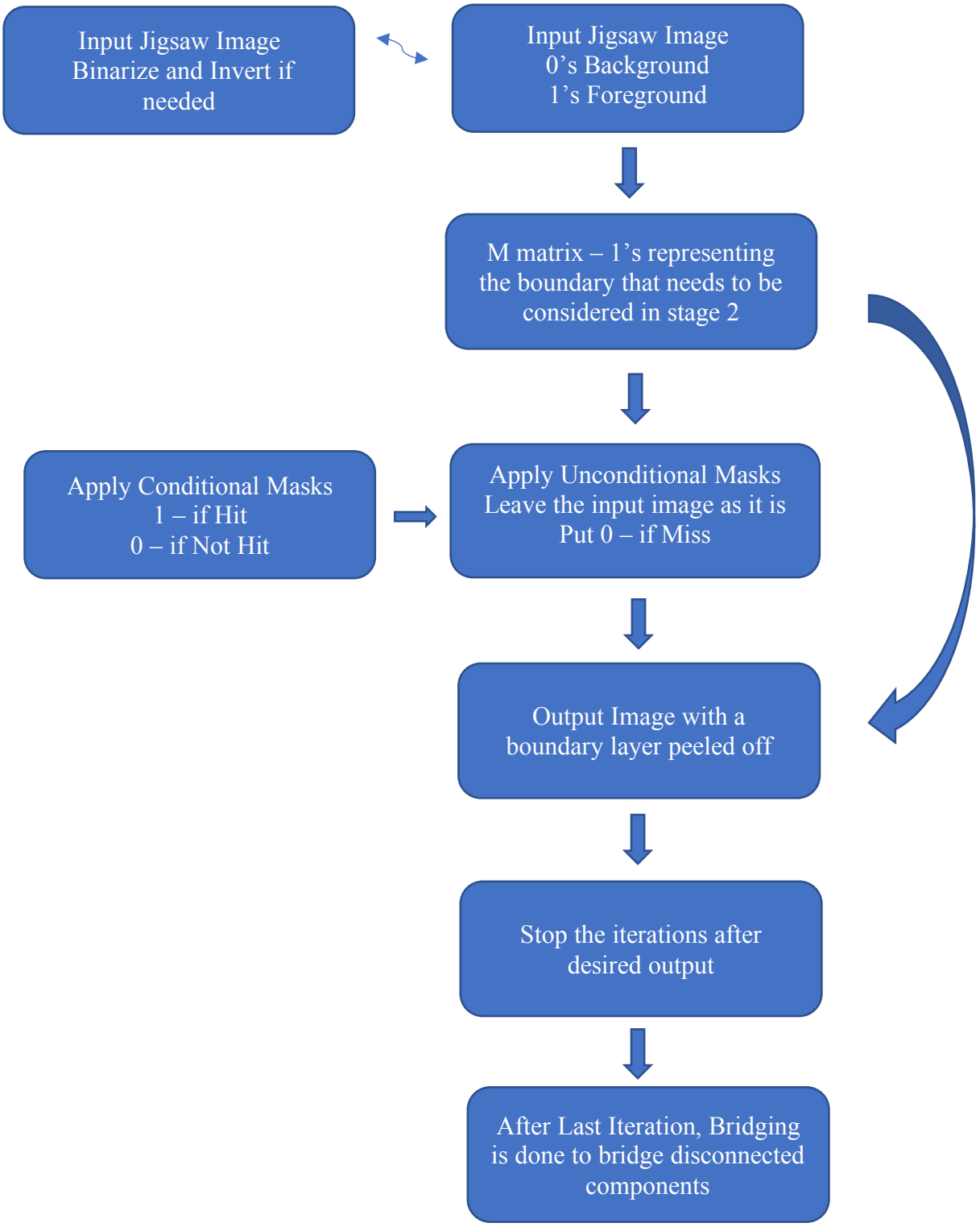
Skeletonizing is a process of reducing foreground regions in a binary image to a skeletal region which preserves the extent of connectivity and disregards most of the other foreground pixels. When compared to Shrinking and Thinning, Skeletonizing preserves most of the information and is the least extreme in extracting morphological information. In this question, I have performed Skeletonizing on the given Jigsaw image. Similar to previous morphological operation, here also 0's are considered as the background and 1's are considered as foreground.

➢ APPROACH AND PROCEDURES:

The Skeletonizing operation, is related to hit and miss transform based on Conditional and Unconditional masks that are given in the tables below. The structure elements are passed through the image pixels and a skeleton representation of the underlying Foreground image is visualized. A cool example of Skeletonizing on a simple binary image is shown below.

(Source: http://www.coe.utah.edu/~cs4640/slides/Lecture11.pdf)




Input Jigsaw Image
Binarize and Invert if needed
Input Jigsaw Image
0's Background
1's Foreground
M matrix – 1's representing the boundary that needs to be considered in stage 2
Apply Conditional Masks
1 – if Hit
0 – if Not Hit
Apply Unconditional Masks
Leave the input image as it is
Put 0 – if Miss
Output Image with a boundary layer peeled off
Stop the iterations after desired output
After Last Iteration, Bridging is done to bridge disconnected components

The Conditional Mask Patterns are given below:
(Source: Digital Image Processing by William K. Pratt – pg 433)

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

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

The Unconditional Mask Patterns are given below:
(Source: Digital Image Processing by William K. Pratt – pg 435)

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

- Please Note that the Conditional Mask Patterns for Shrinking, Thinning and Skeletonizing are given in one single table and hence is the same in all the above sub questions.
- The Unconditional Mask Patterns are same for Shrinking and Thinning. Whereas it is different for Skeletonizing.
- Since Shrinking and Thinning are very extreme in extracting morphological information, they have similar Unconditional Mask Patterns.
- But Skeletonizing is not so extreme where as it preserves most of the information in a binary image to give the skeleton of the image structure, it has a different Unconditional Mask Patterns.

More Details on Bridging:

At the conclusion of last iteration, one step of Bridging was performed to avoid the unnecessary disjoints in the output.

(Source: Digital Image Processing by William K. Pratt – pg 426)

Bridge. Create a black pixel if creation results in connectivity of previously unconnected neighboring black pixels.

$$G(j,k) = X \cup [P_1 \cup P_2 \cup \cdots \cup P_6] \tag{14.2-4a}$$

where

$$P_1 = \bar{X}_2 \cap \bar{X}_6 \cap [X_3 \cup X_4 \cup X_5] \cap [X_0 \cup X_1 \cup X_7] \cap \bar{P}_Q \tag{14.2-4b}$$

$$P_2 = \bar{X}_0 \cap \bar{X}_4 \cap [X_1 \cup X_2 \cup X_3] \cap [X_5 \cup X_6 \cup X_7] \cap \bar{P}_Q \tag{14.2-4c}$$

$$P_3 = \bar{X}_0 \cap \bar{X}_6 \cap X_7 \cap [X_2 \cup X_3 \cup X_4] \tag{14.2-4d}$$

$$P_4 = \bar{X}_0 \cap \bar{X}_2 \cap X_1 \cap [X_4 \cup X_5 \cup X_6] \tag{14.2-4e}$$

$$P_5 = \bar{X}_2 \cap \bar{X}_4 \cap X_3 \cap [X_0 \cup X_6 \cup X_7] \tag{14.2-4f}$$

$$P_6 = \bar{X}_4 \cap \bar{X}_6 \cap X_5 \cap [X_0 \cup X_1 \cup X_2] \tag{14.2-4g}$$

and

$$P_Q = L_1 \cup L_2 \cup L_3 \cup L_4 \tag{14.2-4h}$$

$$L_1 = \bar{X} \cap \bar{X}_0 \cap X_1 \cap \bar{X}_2 \cap X_3 \cap \bar{X}_4 \cap \bar{X}_5 \cap \bar{X}_6 \cap \bar{X}_7 \tag{14.2-4i}$$

$$L_2 = \bar{X} \cap \bar{X}_0 \cap \bar{X}_1 \cap \bar{X}_2 \cap X_3 \cap \bar{X}_4 \cap X_5 \cap \bar{X}_6 \cap \bar{X}_7 \tag{14.2-4j}$$

$$L_3 = \bar{X} \cap \bar{X}_0 \cap \bar{X}_1 \cap \bar{X}_2 \cap \bar{X}_3 \cap \bar{X}_4 \cap X_5 \cap \bar{X}_6 \cap X_7 \tag{14.2-4k}$$

$$L_4 = \bar{X} \cap \bar{X}_0 \cap X_1 \cap \bar{X}_2 \cap \bar{X}_3 \cap \bar{X}_4 \cap \bar{X}_5 \cap \bar{X}_6 \cap X_7 \tag{14.2-4l}$$

Algorithm Implemented (C++):

main() Function:

- Read given *Jigsaw_2.raw* image using *fileRead()* function
- Convert 1D image to 2D using *image1Dto2D()* function
- Allocate memory for Binary image, 1st stage M image and final output using *allocMemory2D()*
- Convert the grayscale image to Binary using fixed thresholding. A threshold of 127 was chosen for this Binary Conversion
- Invert the image to have 0's in background and 1 in foreground
- Get first stage intermediate matrix using *returnFirstStage()* function
- Get second stage output using *returnSecondStage()* function
- Iteratively call the first stage and second stage to get the desired output
- The desired output is when there is no change to the image and it saturates
- Bridging is performed to join the disjoint lines in output using *returnBridgedImage()*
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()*

returnFirstStage() Function:

- Initialize the intermediate matrix with all zeros
- Mask size is fixed as 3
- Traverse through the image using 2 for loops
- Get every single pixel if it is 0 or 1
- If it's 0 – do nothing
- If it's 1 – calculate the bond value and compare the 3*3 neighbor of that pixel with all the bond's corresponding Conditional Masks
- If it's a hit, put 1 in the Intermediate matrix. If 0 put zero
- Return the intermediate matrix

returnSecondStage() Function:

- Get the input matrix along with Intermediate Matrix
- Mask size is fixed as 3
- Traverse through the image using 2 for loops
- Get every single pixel if it is 0 or 1
- If it's 0 – copy input to output
- If it's 1 – compare the 3*3 neighbor of that pixel with all the Conditional Masks
- If it's a hit – copy input to output
- If it's a miss – put 0 in the output pixel value
- Return the output image

compareWithMask() Function:

- Check every element of the 3*3 neighbor with every element of the mask
- If one element doesn't match, break
- Iterate through all the masks

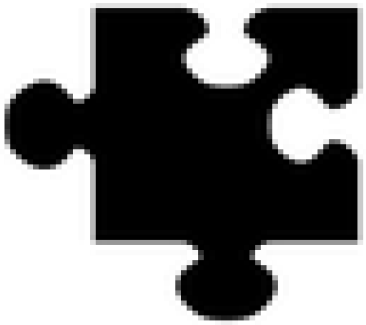
- If one mask completely matches, break
- Return hit/miss output

returnBridgedImage() Function:

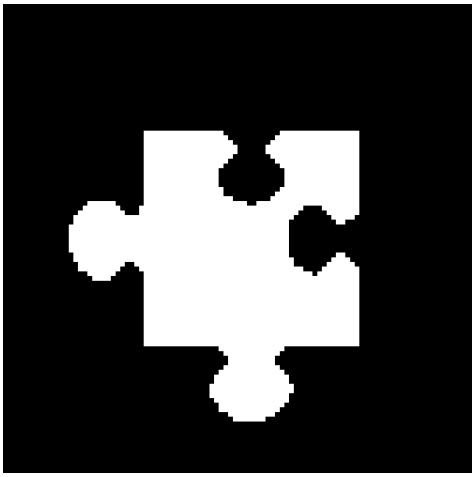
- Traverse through the final iteration image using 2 for loops
- Get X, X0, X1, X2, X3, X4, X5, X6 and X7
- X is the center pixel and all others are surrounding neighbor pixels
- Calculate L1, L2, L3, L4, P1, P2, P3, P4, P5 and P6 using the above mentioned Logical Operations
- Calculate the pixel intensity value using the above variables
- Return the Bridged Image

➢ EXPERIMENTAL RESULTS:

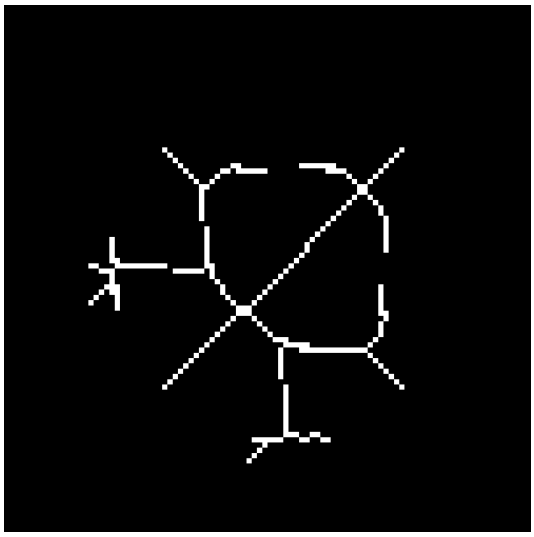
Input Image - Jigsaw $_2$



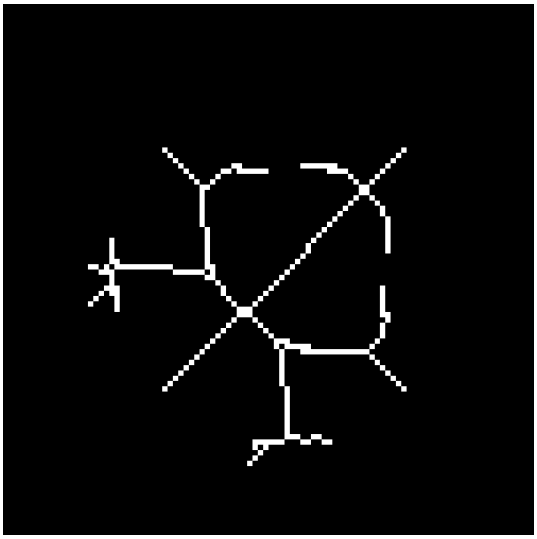
Jigsaw$_2$ - Binary Inverted Image



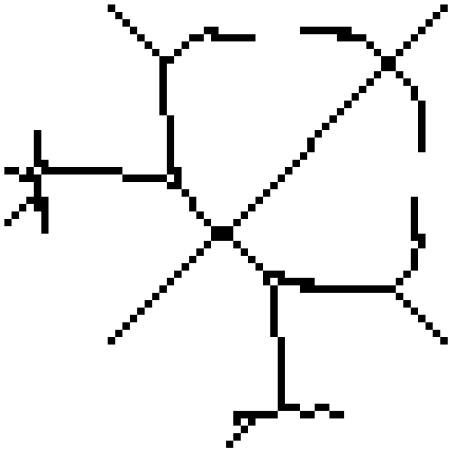
Jigsaw$_2$ - Skeletonized without Bridging



Jigsaw$_2$ - Skeletonized with Bridging



Jigsaw$_2$ - Skeletonized with Bridging



- **DISCUSSION:**

- The results above show that, when compared to Shrinking or Thinning, Skeletonizing gives more information about the structure/skeleton of a binary image.
- Skeletonizing is more sensitive to noise, since it preserves most information when compared to other morphological operation we saw above.
- Hence, Skeletonizing can be used when an application needs a simple compact information of the skeleton of the image.

➢ COUNTING GAME: (NUMBER OF PIECES IN THE BOARD IMAGE)

➢ ABSTRACT AND MOTIVATION:

This portion of the question is the most interesting one of the homework. This question is the application of Morphological Image Processing. We would see how the above learnt concepts of Shrinking, Thinning and Skeletonizing along with other morphological processing like Erosion and Dilation can be combined to automate counting game. In part 1 of this question, I have performed Erosion, Dilation and Shrinking to count the number of Jigsaws in the given board image.

➢ APPROACH AND PROCEDURES:

We saw Shrinking as the extreme process of morphological operation to reduce a solid foreground in an image to a single point. So Initially, I approached this problem as Shrinking, so as to reduce each Jigsaw image to a single point. Hence, I thought it would be like Stars counting to count the number of single image pixels after Shrinking.

But, the challenge was reducing each Jigsaw to a single point. Since the Jigsaws had holes and protrusions, just applying Shrinking algorithm reduced each Jigsaw to more than one pixels. So just Shrinking all the Jigsaws was not enough.

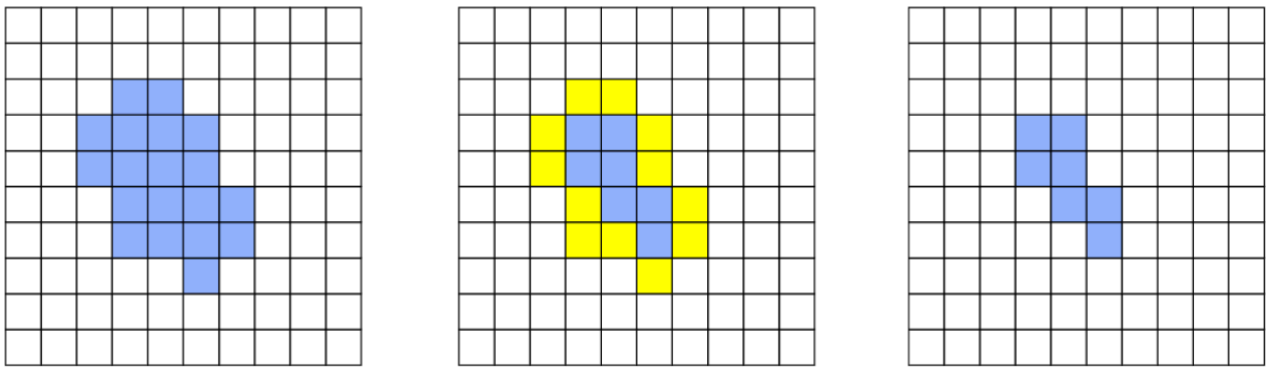
I had to first erode and dilate each Jigsaw image so that the, they are not reducing to more than one point when Shrinking is applied.

Once this was done, every Jigsaw got reduced to a single image pixel, and I counted the number of single foreground shrunk pixels to get the number of Jigsaws in the board. Please note that this process would not give number of unique objects but will give just the number of objects in the image.

The process is explained in detail below and outputs with Erosion and Dilation are discussed. The outputs of Shrinking without them are also discussed below, to give an intuition on why they were necessary

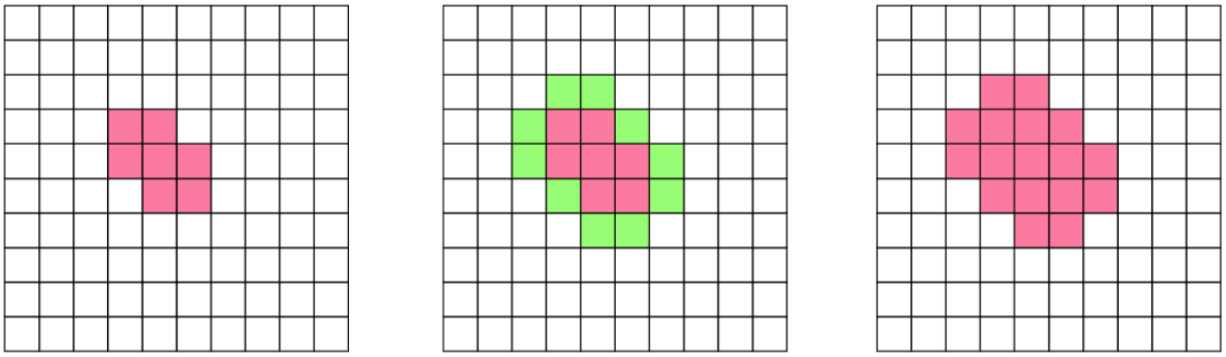
Erosion:

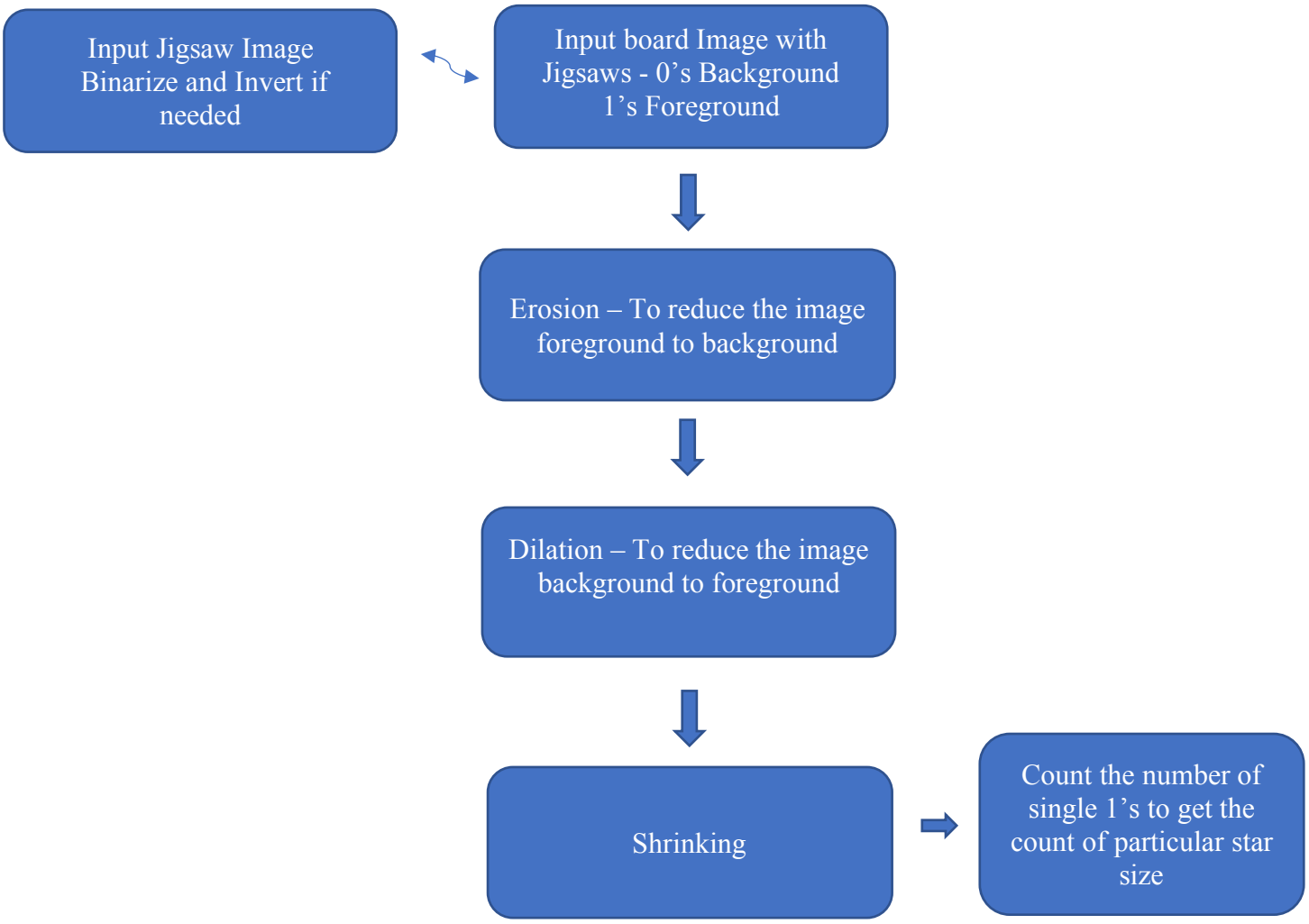
Erosion is the process of changing the foreground pixel to background if it has a background pixel in its 8-neighbour. An example is shown below.
(Source: http://www.coe.utah.edu/~cs4640/slides/Lecture11.pdf)



Dilation:

Dilation is the process of changing the background pixel to foreground if it has a foreground pixel in its 8-neighbour. An example is shown below.
(Source: http://www.coe.utah.edu/~cs4640/slides/Lecture11.pdf)





Algorithm Implemented (C++):

main() Function:

- Read given *board.raw* image using *fileRead()* function
- Convert 1D image to 2D using *image1Dto2D()* function
- Allocate memory for Binary image, 1st stage M image and final output using *allocMemory2D()*
- Convert the grayscale image to Binary using fixed thresholding. A threshold of 127 was chosen for this Binary Conversion
- Invert the image to have 0's in background and 1 in foreground
- Erode the image using *erodeOrDilate()* function with "erode" as the parameter
- Dilate the image using *erodeOrDilate()* function with "dilate" as the parameter
- Get first stage intermediate matrix using *returnFirstStage()* function
- Get second stage output using *returnSecondStage()* function
- Iteratively call the first stage and second stage to get the desired output
- The desired output is when there is no change to the image and it saturates
- Count the number of single image points
- Write them to output raw files using *fileWrite()*
- Deallocate all the memories using *delete*, *freeMemory2D()*

erodeOrDilate() Function:

- Check erosion or dilation has to be performed using the arguments passed
- Traverse through the image using 2 for loops
- Get the 8-connected components of every pixel in the image
- Perform erosion/dilation needed
- Return eroded/dilated image array

returnFirstStage() Function:

- Initialize the intermediate matrix with all zeros
- Mask size is fixed as 3
- Traverse through the image using 2 for loops
- Get every single pixel if it is 0 or 1
- If it's 0 – do nothing
- If it's 1 – calculate the bond value and compare the 3*3 neighbor of that pixel with all the bond's corresponding Conditional Masks
- If it's a hit, put 1 in the Intermediate matrix. If 0 put zero
- Return the intermediate matrix

returnSecondStage() Function:

- Get the input matrix along with Intermediate Matrix
- Mask size is fixed as 3
- Traverse through the image using 2 for loops
- Get every single pixel if it is 0 or 1

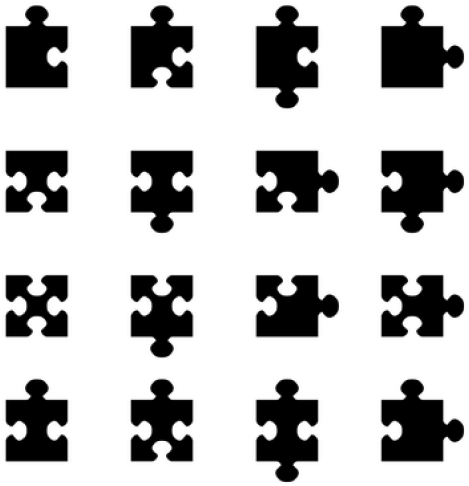
- If it's 0 – copy input to output
- If it's 1 – compare the 3*3 neighbor of that pixel with all the Conditional Masks
- If it's a hit – copy input to output
- If it's a miss – put 0 in the output pixel value
- Return the output image

compareWithMask() Function:

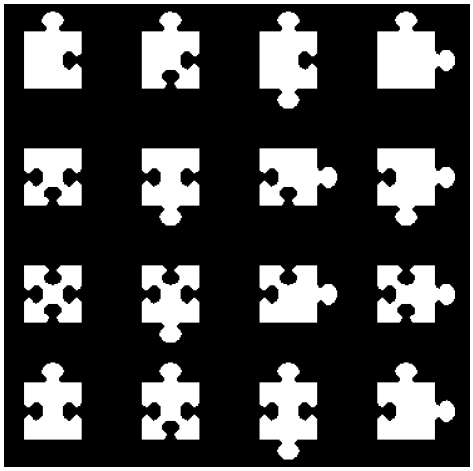
- Check every element of the 3*3 neighbor with every element of the mask
- If one element doesn't match, break
- Iterate through all the masks
- If one mask completely matches, break
- Return hit/miss output

➢ EXPERIMENTAL RESULTS:

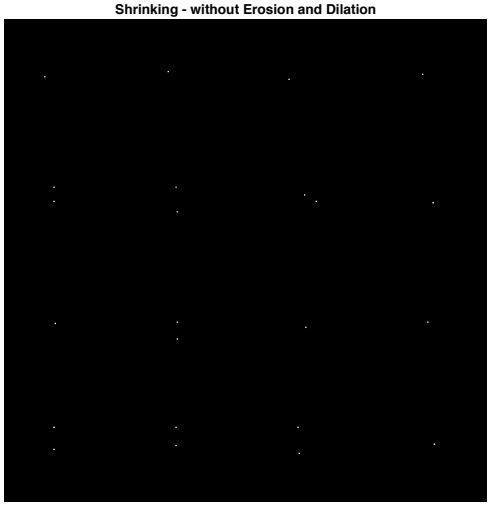
Input Board Image



Input Board Image - Binarized and Inverted

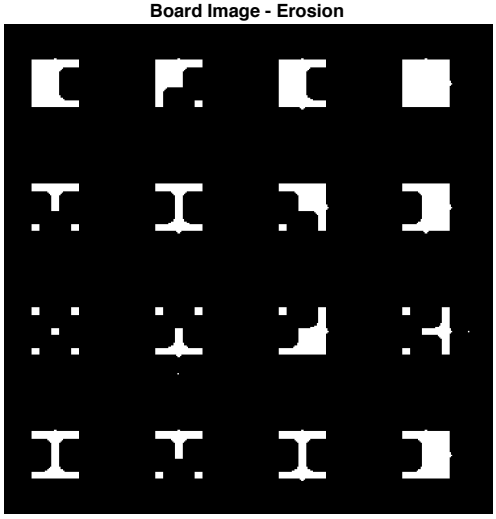


Shrinking - without Erosion and Dilation

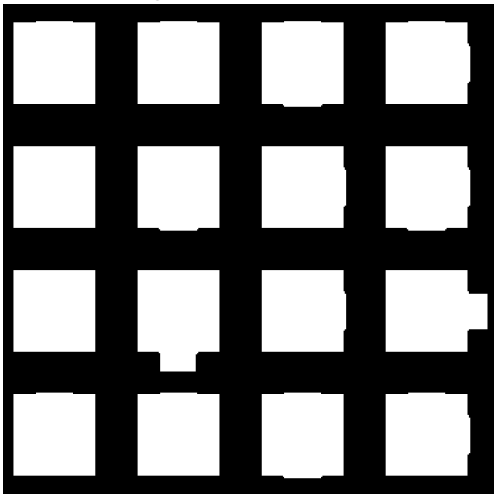


Notice that the Shrinking doesn't reduce the Jigsaws to a single point
Hence counting cannot be done with just Shrinking!

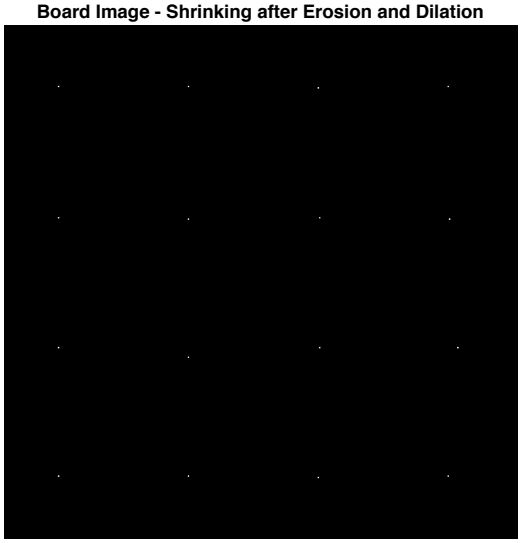
Board Image - Erosion

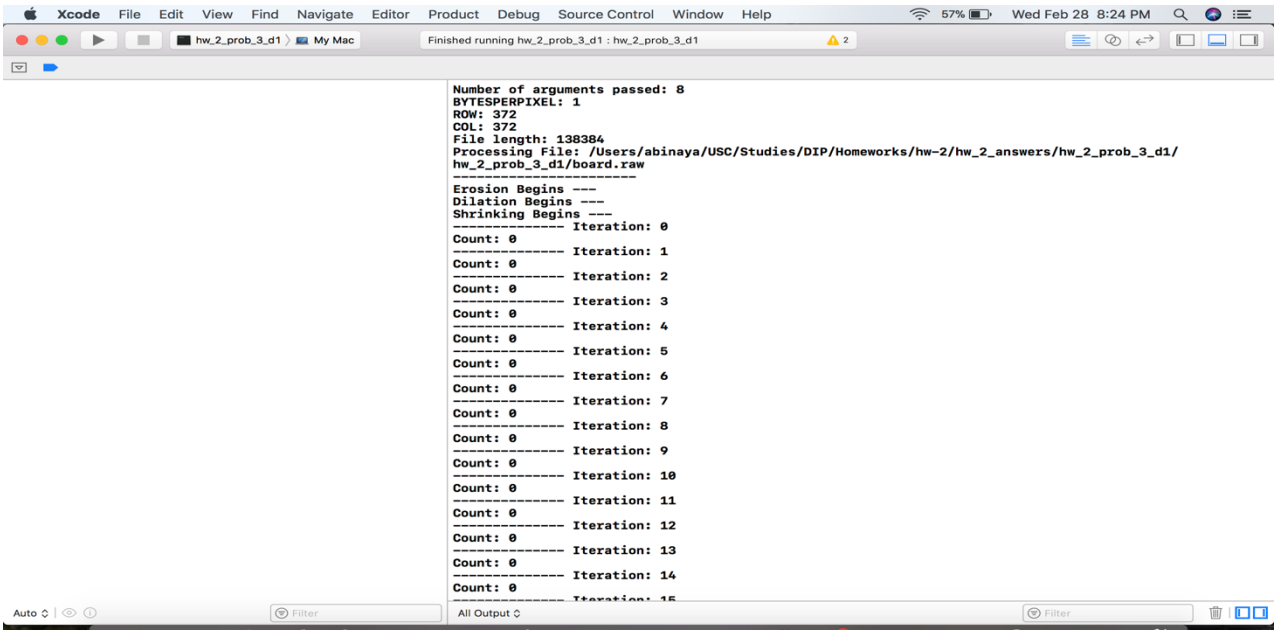


Board Image - after Erosion and Dilation

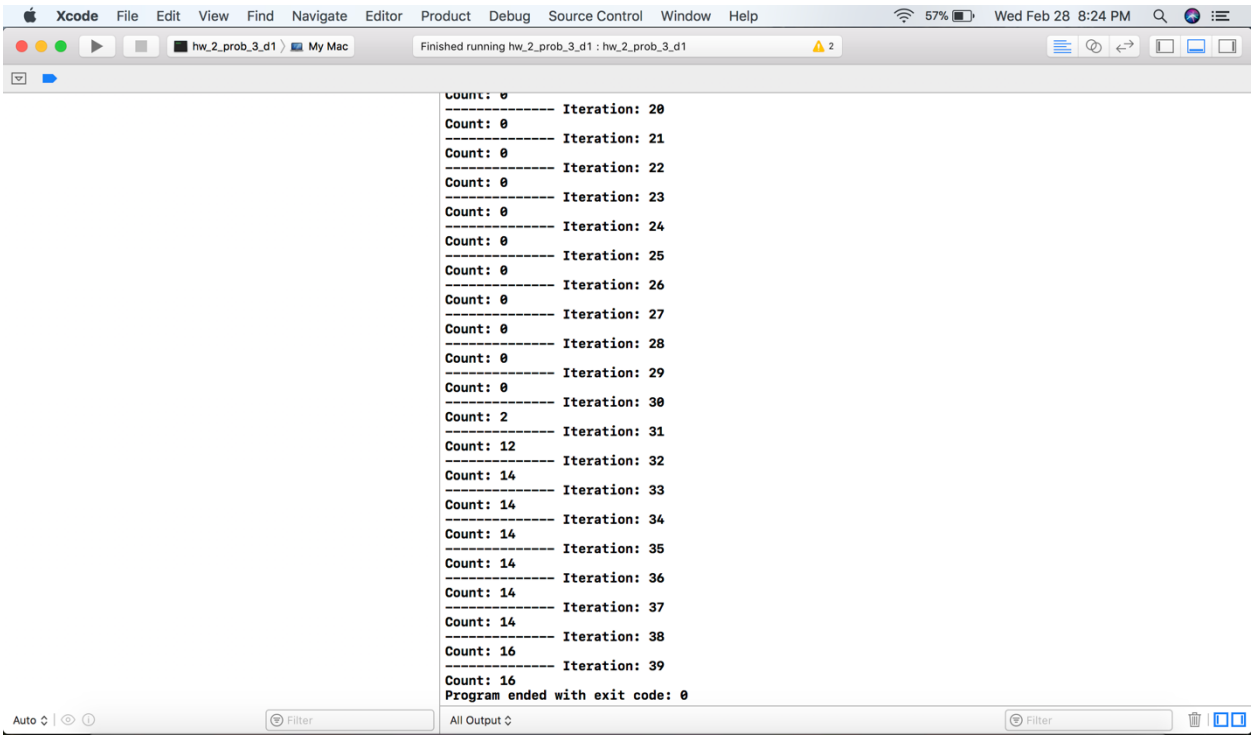


Board Image - Shrinking after Erosion and Dilation





Screenshot of Erosion, Dilation and Shrinking (First few iterations)



Screenshot of Shrinking (Last few iterations)

No of Jigsaws in the given board image = 16

➢ DISCUSSION:

Thus, I demonstrated saw how Erosion and Dilation can be used before Shrinking to count the number of items in the given board image. All Jigsaw elements were shrunk to a single foreground pixel. And after many iterations it happened. It nearly took 40 iterations of shrinking (First stage and Second stage) to reduce them to single foreground points. Though this might not be an efficient method to count the number of Jigsaws in the image, this method showed an elegant application of morphological operations!

➢ COUNTING GAME: (NUMBER OF UNIQUE PIECES IN THE BOARD IMAGE)

➢ ABSTRACT AND MOTIVATION:

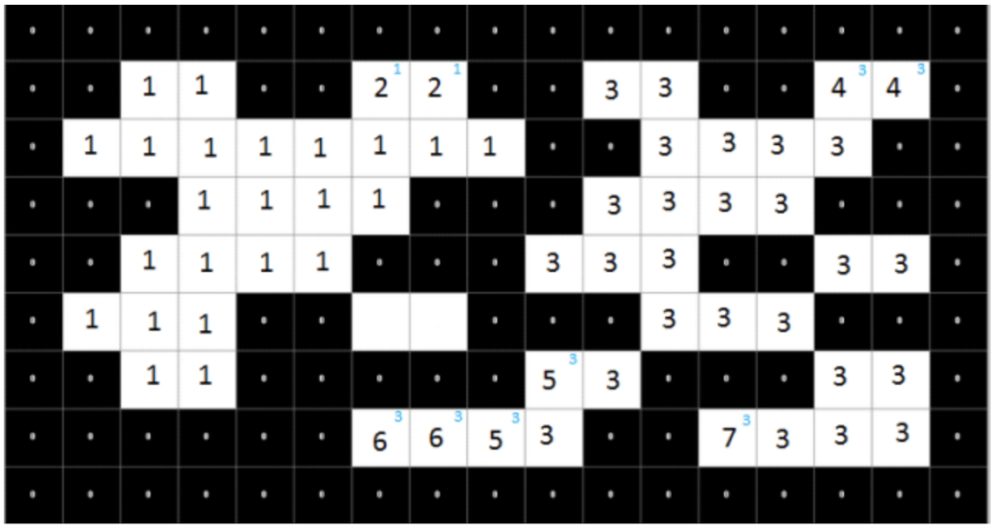
This is another interesting portion of the question to count the number of unique objects in the given board image. I developed an elegant algorithm to keep track of number of protrusions and holes in every side of the Jigsaw. Similar pieces will have protrusions and holes in the similar way is the concept behind this developed algorithm.

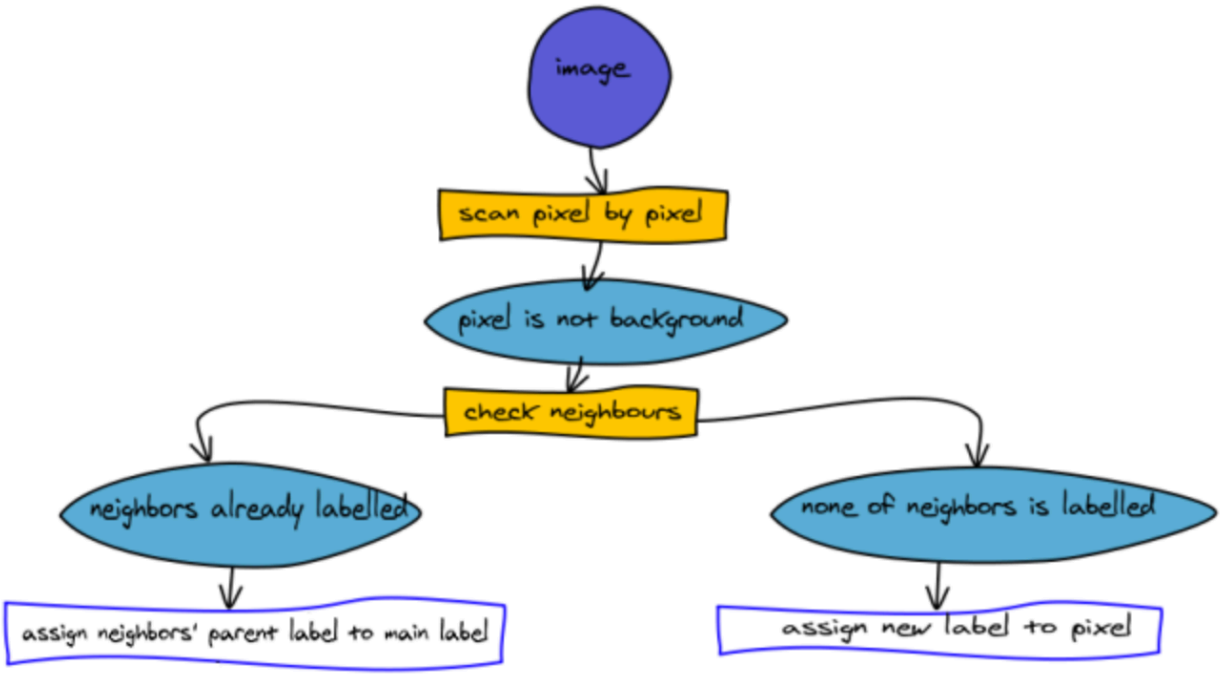
➢ APPROACH AND PROCEDURES:

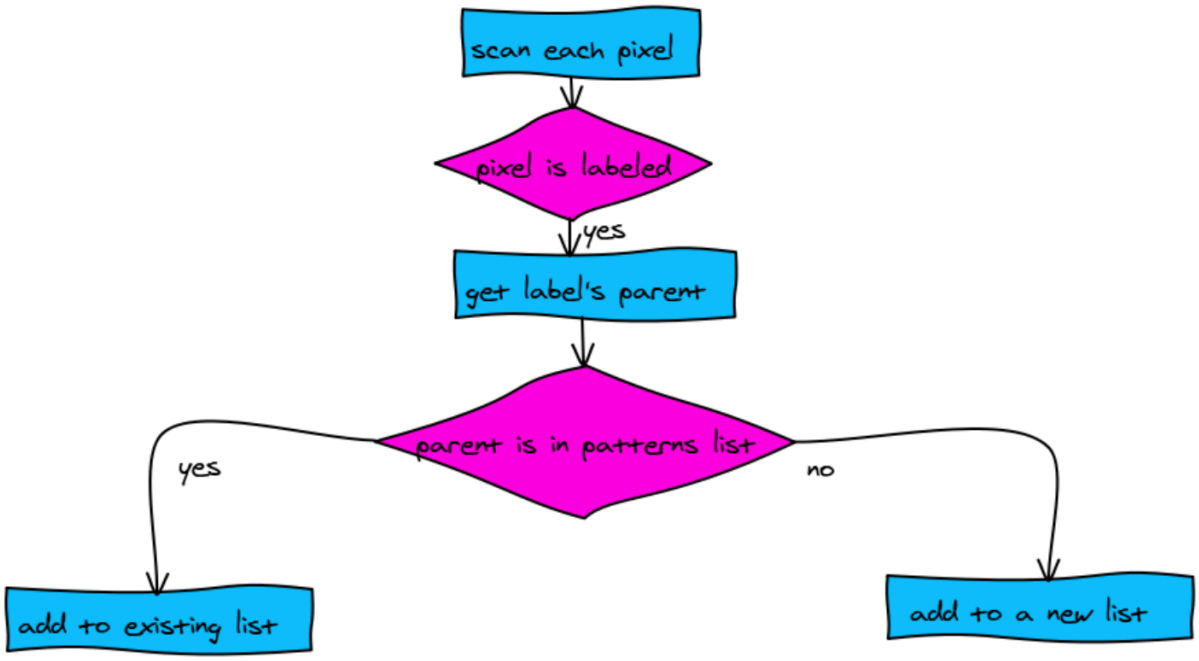
I first used Connected Component Analysis for the labelling all the objects in the given image. At the end of Connected Component Analysis, each Jigsaw would be given a unique label. Then, I followed the below given block diagram to count the number of unique objects.

Connected Component Analysis:

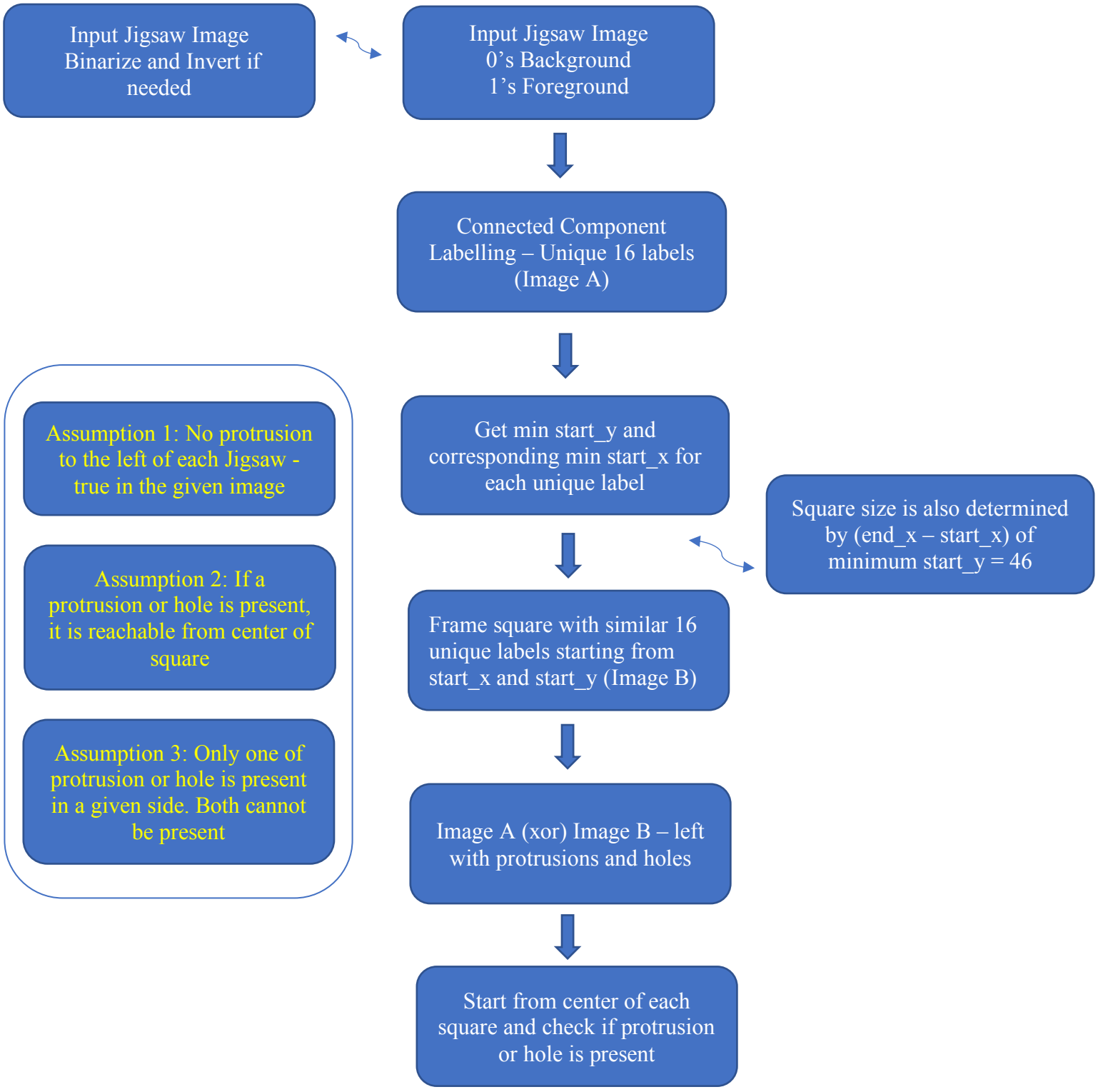
Connected Component Analysis scans the entire image and group the pixels into components according to pixel connectivity.
(Source: https://www.codeproject.com/Articles/336915/Connected-Component-Labeling-Algorithm)



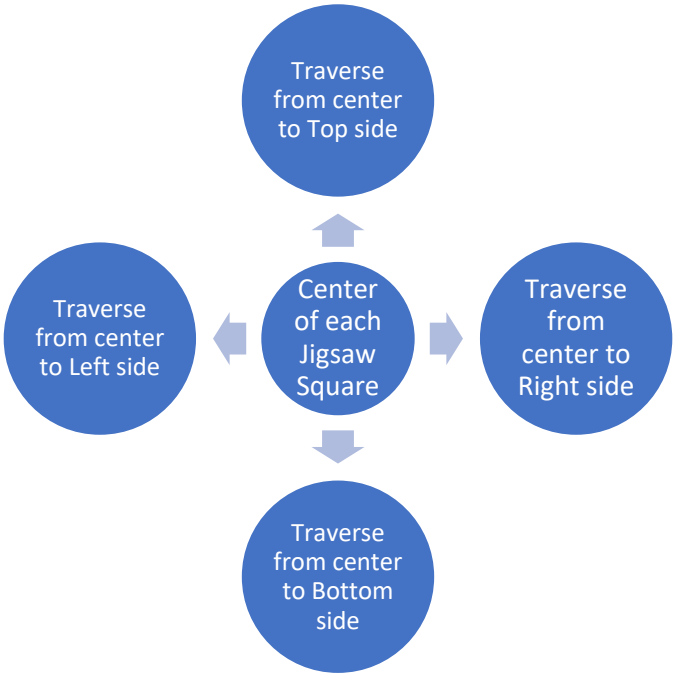




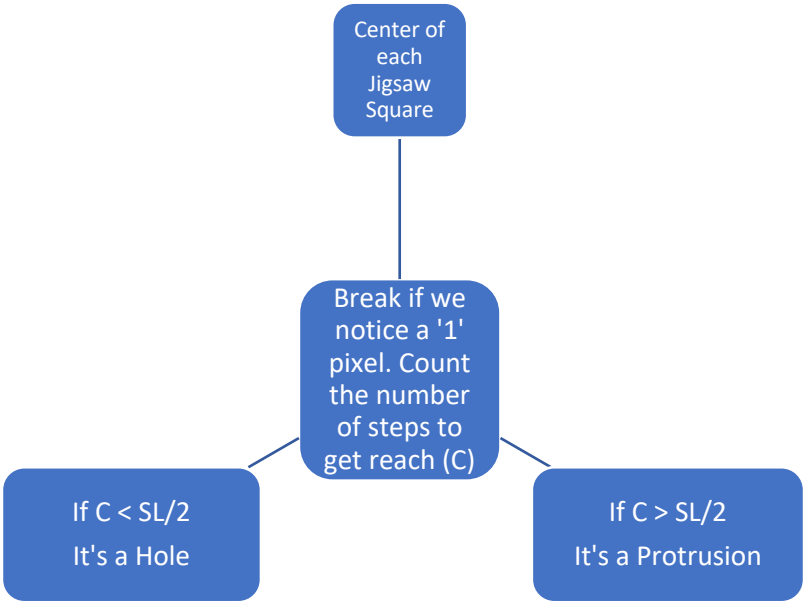
Sample output of Connected Component Labelling is shown in the previous page. This page has the flow chart for first stage and second stage.


Input Jigsaw Image
Binarize and Invert if needed
Input Jigsaw Image
0's Background
1's Foreground
Connected Component Labelling – Unique 16 labels (Image A)
Get min start_y and corresponding min start_x for each unique label
Square size is also determined by (end_x – start_x) of minimum start_y = 46
Frame square with similar 16 unique labels starting from start_x and start_y (Image B)
Image A (xor) Image B – left with protrusions and holes
Start from center of each square and check if protrusion or hole is present
Assumption 1: No protrusion to the left of each Jigsaw - true in the given image
Assumption 2: If a protrusion or hole is present, it is reachable from center of square
Assumption 3: Only one of protrusion or hole is present in a given side. Both cannot be present

Algorithm to Check for protrusions and Holes in each side:



As shown from the above diagram, I traverse from center of the pixel to top, right, bottom and left side of each Labelled Xor'ed Jigsaw.



Here SL - Square length determined by the above flow chart algorithm. I get an array of protrusions and holes for each Jigsaw. An example is given below

Example Array of length 4 for Protrusion and Holes for Jigsaw – 1:

GIVEN IMAGE	TOP	RIGHT	BOTTOM	LEFT
PROTRUSIONS	1	0	0	0
HOLES	0	1	0	0

Rotation:

ROTATED ONCE	TOP	RIGHT	BOTTOM	LEFT
PROTRUSIONS	0	1	0	0
HOLES	0	0	1	0

ROTATED TWICE	TOP	RIGHT	BOTTOM	LEFT
PROTRUSIONS	0	0	1	0
HOLES	0	0	0	1

ROTATED TRICE	TOP	RIGHT	BOTTOM	LEFT
PROTRUSIONS	0	0	0	1
HOLES	1	0	0	0

Reflection:

REFLECTED VERTICALLY	TOP	RIGHT	BOTTOM	LEFT
PROTRUSIONS	0	0	1	0
HOLES	0	1	0	0

REFLECTED HORIZONTALLY	TOP	RIGHT	BOTTOM	LEFT
PROTRUSIONS	1	0	0	0
HOLES	0	0	0	1

- This array is framed automatically for every Jigsaw
- Rotating this array and reflecting this array is a proxy for rotating and reflecting the image
- Rotations (90 degrees each time) and Reflections (Horizontal and Vertical) are compared with every other Jigsaw
- Rotating once – Right shift the values in the array
- Vertical Reflection – Top and Bottom interchanges
- Horizontal Reflection – Left and Right interchanges
- If they match with any rotated or reflected jigsaw, the counter of the matched Jigsaw is increased
- Else, the jigsaw is added to the unique objects
- After Finding the unique objects, count the number of unique objects.

Algorithm Implemented (C++):

main() Function:

- Read given *board.raw* image using *fileRead()* function
- Convert 1D image to 2D using *image1Dto2D()* function
- Allocate memory for Binary image, 1st stage M image and final output using *allocMemory2D()*
- Convert the grayscale image to Binary using fixed thresholding. A threshold of 127 was chosen for this Binary Conversion
- Invert the image to have 0's in background and 1 in foreground
- Label each Jigsaw using *connectedComponentLabels()*
- Get unique labels in an array
- Get all the x and y indices of each label
- Get Minimum y of every label. This is start_y
- Get Minimum and Maximum x label for start_y
- Get square size by end_x – start_x
- Frame squares of given square size by starting from start_x and start_y of every label using 2 for loops
- Xor both the matrices we have
- Frame maps for Protrusions and Holes. Keys are the unique labels got and values are the arrays of protrusions and holes for each label
- Traverse from the center of the square to top, right, bottom and left. Put 1- if hole/protrusion is found in the array
- Initialize map for unique Jigsaw Label
- Start from the first unique label. First unique label is automatically put to the map
- From second label to 16th label, each label's protrusions and holes array is checked with rotates/reflected versions of previous unique Jigsaws
- Get the size of unique Jigsaw Map – Outputs the number of unique labels

connectedComponentLabels() Function:

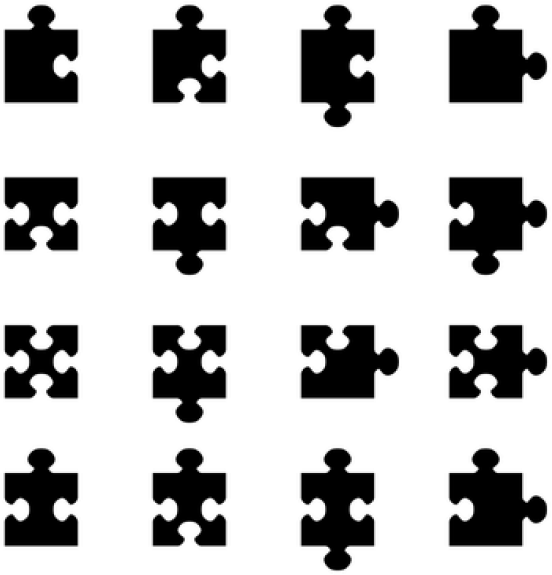
- Traverse through the image pixels using 2 for loops
- If all the previous 4 connected components are zero, give a unique label for the present pixel
- If there are labels, give the present pixel, the minimum of previous labels and update the look up table using insertLookUpTable()
- In the second stage, traverse from highest label value to the lowest label value in a for loop
- Check of the label (l) is present in the second column of the lookup table
- Get all the corresponding right column labels (rls)
- Get the minimum of right column labels (minL)
- Relabel the label (l) and corresponding right labels (rls) also to the minimum label (minL) in the image
- Left with 16 unique labels in the image

insertLookUpTable() Function:

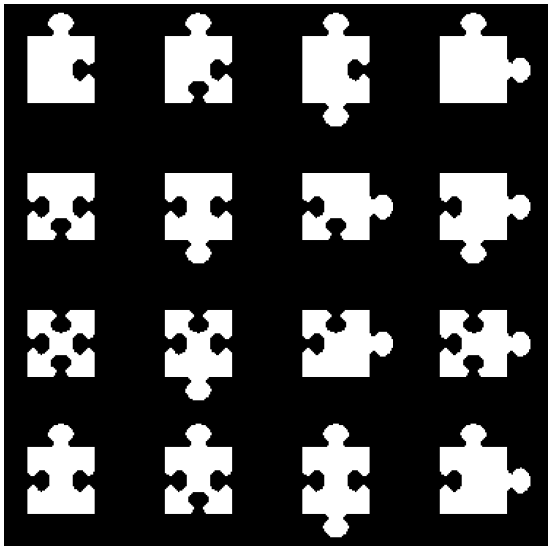
- Inputs are label x and y
- Check if the labels pair x and y are already present in the look up table (Order interchanging is also checked)
- Else, insert the min(x,y) to the left/1st column and max(x,y) to the right/2nd column
- Return the next index position to be inserted

➢ EXPERIMENTAL RESULTS:

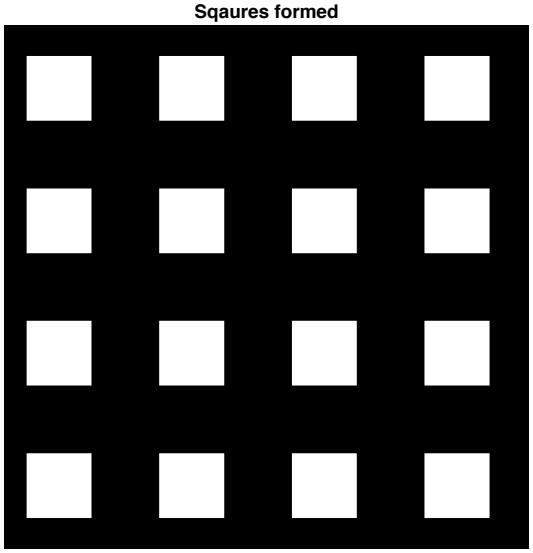
Input Board Image



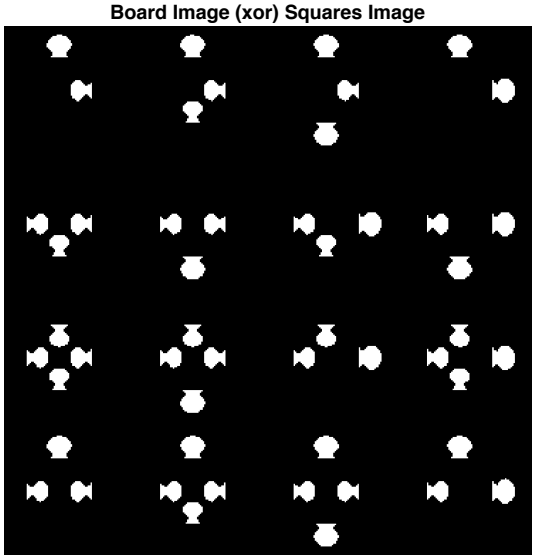
Input Board Image - Binarized and Inverted



Sqaures formed

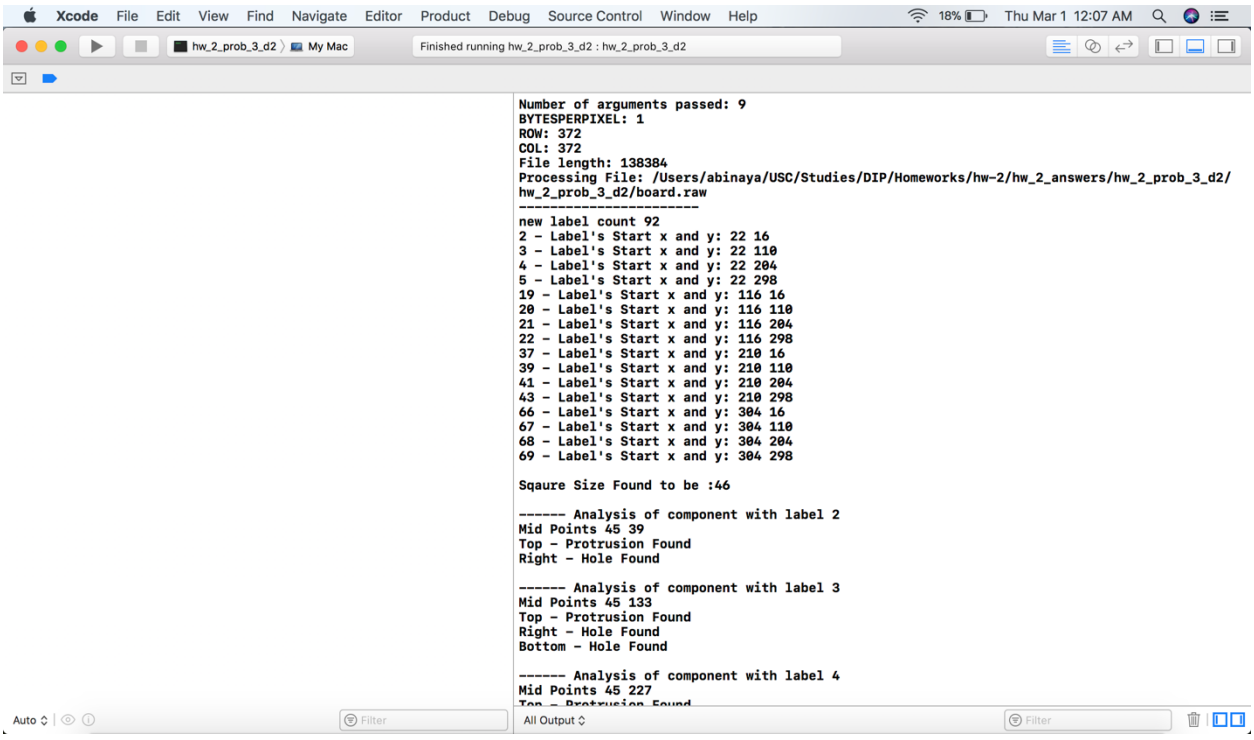


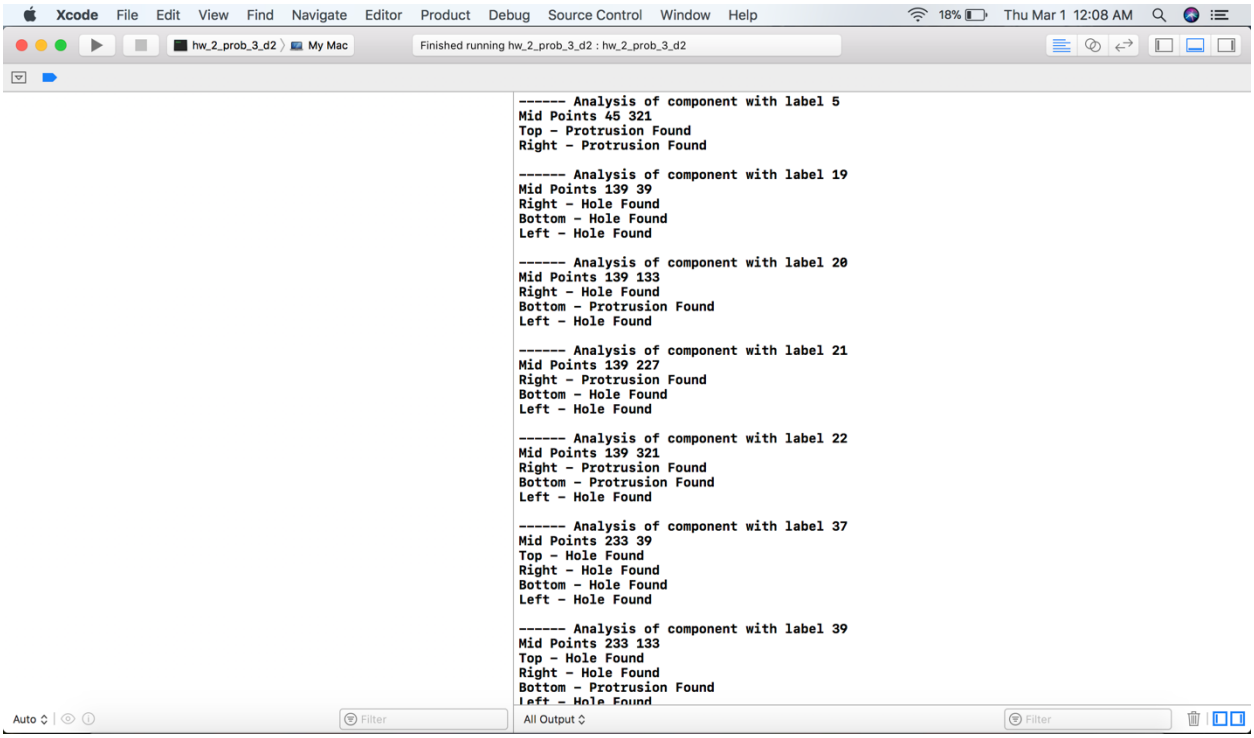
Board Image (xor) Squares Image

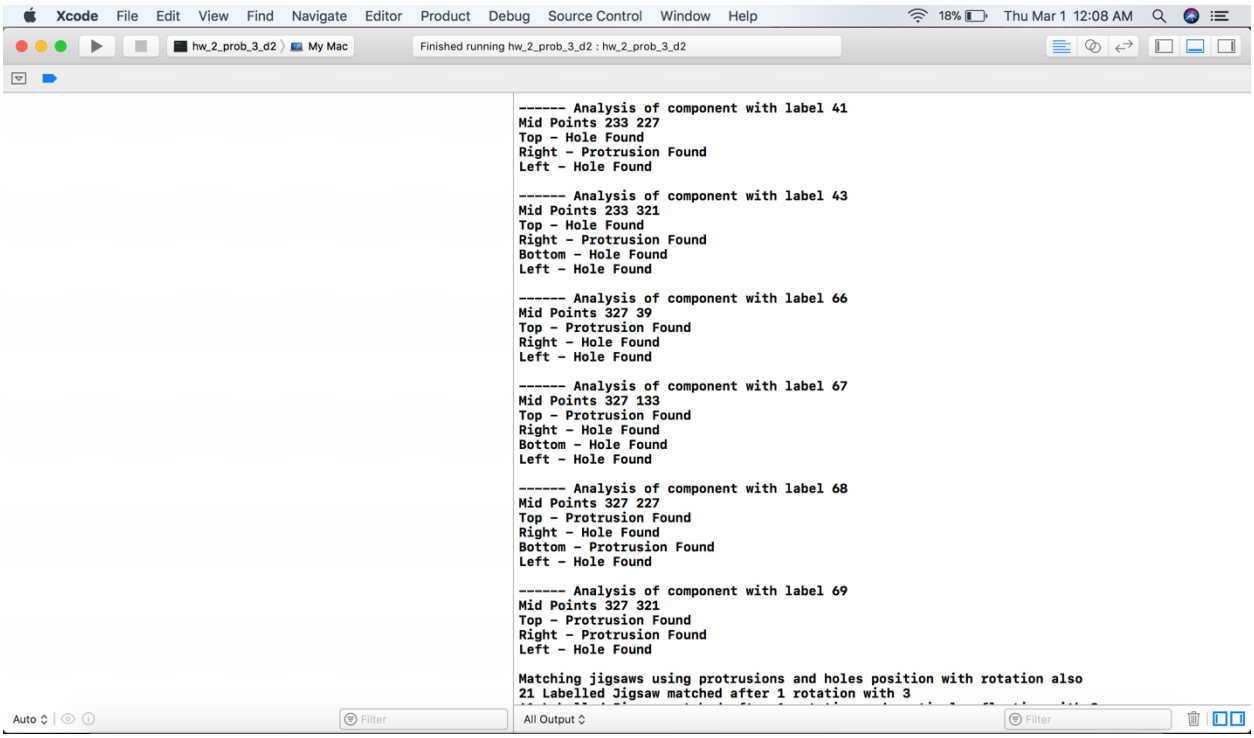


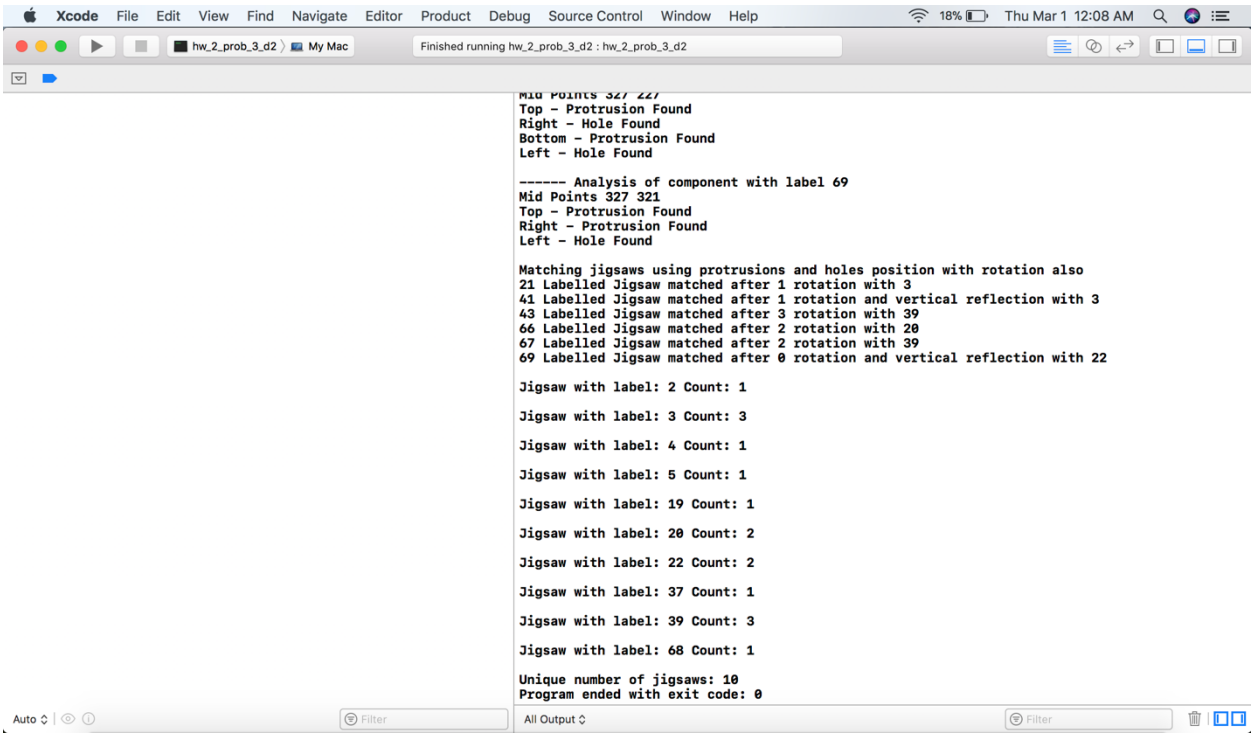
2	3	4	5
19	20	21	22
37	39	41	43
66	67	68	69

Unique Labels given to each Jigsaw was in the above shape and order




Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help
18% Thu Mar 1 12:08 AM
hw_2_prob_3_d2 My Mac
Finished running hw_2_prob_3_d2 : hw_2_prob_3_d2
------ Analysis of component with label 5
Mid Points 45 321
Top - Protrusion Found
Right - Protrusion Found
------ Analysis of component with label 19
Mid Points 139 39
Right - Hole Found
Bottom - Hole Found
Left - Hole Found
------ Analysis of component with label 20
Mid Points 139 133
Right - Hole Found
Bottom - Protrusion Found
Left - Hole Found
------ Analysis of component with label 21
Mid Points 139 227
Right - Protrusion Found
Bottom - Hole Found
Left - Hole Found
------ Analysis of component with label 22
Mid Points 139 321
Right - Protrusion Found
Bottom - Protrusion Found
Left - Hole Found
------ Analysis of component with label 37
Mid Points 233 39
Top - Hole Found
Right - Hole Found
Bottom - Hole Found
Left - Hole Found
------ Analysis of component with label 39
Mid Points 233 133
Top - Hole Found
Right - Hole Found
Bottom - Protrusion Found
Auto
Filter
All Output
Filter


Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help
18% Thu Mar 1 12:08 AM
hw_2_prob_3_d2 My Mac
Finished running hw_2_prob_3_d2 : hw_2_prob_3_d2
------ Analysis of component with label 41
Mid Points 233 227
Top - Hole Found
Right - Protrusion Found
Left - Hole Found
------ Analysis of component with label 43
Mid Points 233 321
Top - Hole Found
Right - Protrusion Found
Bottom - Hole Found
Left - Hole Found
------ Analysis of component with label 66
Mid Points 327 39
Top - Protrusion Found
Right - Hole Found
Left - Hole Found
------ Analysis of component with label 67
Mid Points 327 133
Top - Protrusion Found
Right - Hole Found
Bottom - Hole Found
Left - Hole Found
------ Analysis of component with label 68
Mid Points 327 227
Top - Protrusion Found
Right - Hole Found
Bottom - Protrusion Found
Left - Hole Found
------ Analysis of component with label 69
Mid Points 327 321
Top - Protrusion Found
Right - Protrusion Found
Left - Hole Found
Matching jigsaws using protrusions and holes position with rotation also
21 Labelled Jigsaw matched after 1 rotation with 3
Auto
Filter
All Output
Filter



No of Unique Jigsaw's in the given image = 10

➢ DISCUSSION:

- Thus, for the given input board image, I found an elegant and simple way to count the number of unique Jigsaws.
- Connected component analysis and morphological operation like logical XOR were useful in figuring out a simple way to get protrusions and holes.
- Segmenting each Jigsaw and Rotating/Reflecting them would have caused a heavy work.
- But when I was able to frame arrays of protrusions and holes, a simple right shift was used to get the information that I would get from the rotated image itself.
- Thus, we found that there are 10 unique Jigsaws in the image. Other 6 Jigsaws matched with rotated or reflected versions of the previous unique Jigsaws found.
- Though there were a few assumptions made, the algorithm I developed worked well regardless of them.
- The entire problem shows that simple morphological operations can be used in a variety of applications!

APPENDIX
(DESCRIPTION OF FUNCTIONS USED FROM DIP_MyHeaderFile.h)

fileRead() function:

- Reads the given filename to a 1D array

allocMemory2D() function:

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

allocMemory3D() function:

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

Image1Dto2D() function:

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

Image1Dto3D() function:

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

Image2Dto1D() function:

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

Image3Dto1D() function:

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

seperateChannels() function:

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

combineChannels() function:

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

fileWrite() function:

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

fileWriteHist() function

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

freeMemory2D () function

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

freeMemory3D () function:

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

Histogram2DImage () function:

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

(DESCRIPTION OF FUNCTIONS USED FROM STKMasks.h)

returnShrinkConditionalMasks(),
returnThinConditionalMasks() and
returnSkeletonizeConditionalMasks() Functions:

- Return Conditional Masks of respective bonds

returnShrinkBondSize(),
returnThinBondSize() and
returnSkeletonizeBondSize() Functions:

- Return bond size

returnShrinkThinUnconditionalMasks() and returnSkeletonizeUnconditionalMasks() Function:

- Return respective Masks