EE569 – Homework 2 – Mar 4, 2018

A Report on

Geometric Image Modification

Digital Halftoning – Grayscale and Color

Morphological Processing

All the codes were implemented in C++

except for Problem 1-part B, where I have used MATLAB code for calculating the Homography Matrix and its inverse

Submitted by:

ANURAG SYAL

MASTERS IN ELECTRICAL ENGINEERING

MULTIMEDIA AND CREATIVE TECHNOLOGIES

USC ID: 9954-6583-64

Problem 1: Geometric Image Manipulation

Part B: Homographic Transformation and Image Stitching

Abstract and Motivation

Spatial operations are very basic operations of image processing. Such operations enable us to geometrically modify an image. There are 3 basic operations which help is geometric manipulation of an image:

1. Scaling – Changing dimensions of an image, example, changing a 640x480 image to 1000x2000
2. Rotation – Changing the orientation of the image, example, rotation the image along z axis by 45°
3. Translation – Changing the position of the image with respect to its origin or shifting the origin of the image itself with respect to a global origin.

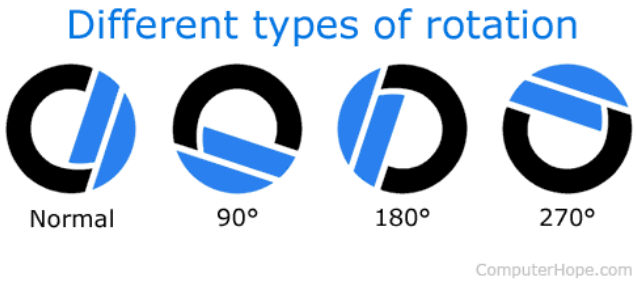
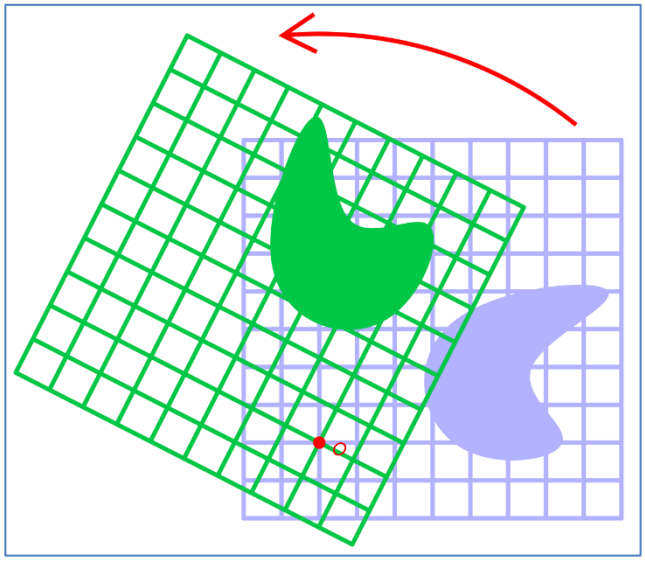


Figure 1: Example showing image rotation in multiples of 90°



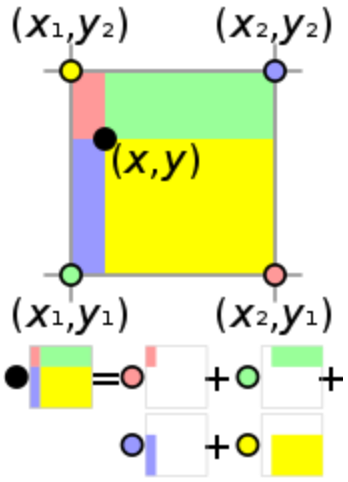


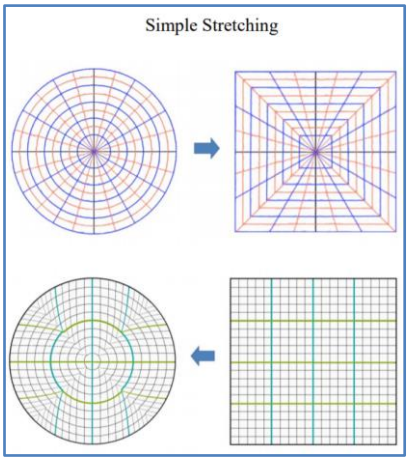
Figure 2: a) A more general form of rotation b) Bilinear Interpolation

Bilinear Interpolation is a technique to smoothly manipulate an image. When a geometric operation is being performed on an image, sometimes the pixels are left empty. Now in normal manipulation algorithms the pixels are copied over. This produces several artefacts in the image and the image overall looks blocky and has several black spots in between. On the other hand, using Bilinear Interpolation, one can fill the empty pixels by interpolating or superimposing the values of nearby pixels.

Approach and Procedures

Geometric Warping – Circular Warping and De-warping

There are many algorithms available for Circular warping and de-warping operations as described below:

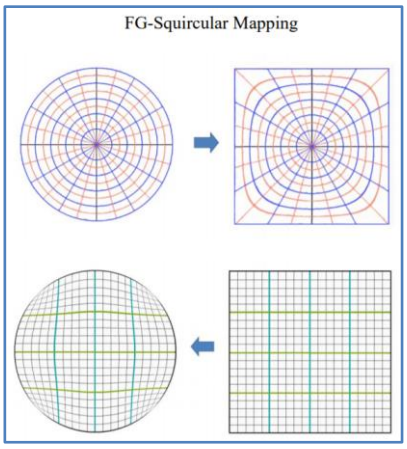


Disc to square mapping:

$$x = \begin{cases} sgn(u)\sqrt{u^2+v^2} & when\ u^2 \geq v^2 \\ sgn(v)\,\frac{u}{v}\sqrt{u^2+v^2} & when\ u^2 < v^2 \end{cases} \qquad y = \begin{cases} sgn(u)\,\frac{v}{u}\sqrt{u^2+v^2} & when\ u^2 \geq v^2 \\ sgn(v)\sqrt{u^2+v^2} & when\ u^2 < v^2 \end{cases}$$

Square to disc mapping:

$$u = \begin{cases} sgn(x)\,\frac{x^2}{\sqrt{x^2+y^2}} & when\ x^2 \geq y^2 \\ sgn(y)\,\frac{x\,y}{\sqrt{x^2+y^2}} & when\ x^2 < y^2 \end{cases} \qquad v = \begin{cases} sgn(x)\,\frac{x\,y}{\sqrt{x^2+y^2}} & when\ x^2 \geq y^2 \\ sgn(y)\,\frac{y^2}{\sqrt{x^2+y^2}} & when\ x^2 < y^2 \end{cases}$$



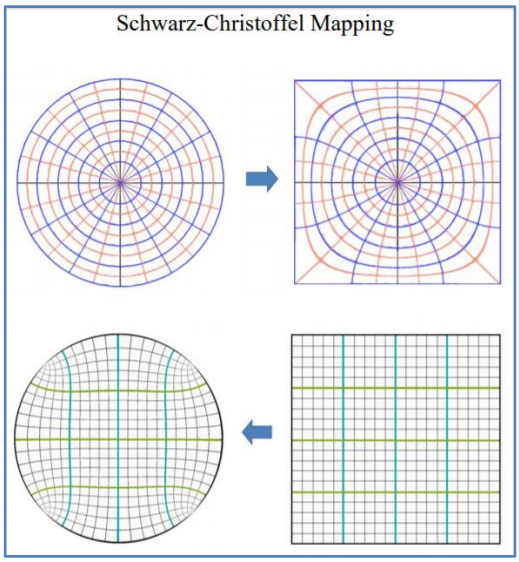
Disc to square mapping:

$$x = \frac{sgn(uv)}{v\sqrt{2}}\sqrt{u^2+v^2-\sqrt{(u^2+v^2)(u^2+v^2-4u^2v^2)}}$$

$$y = \frac{sgn(uv)}{u\sqrt{2}}\sqrt{u^2+v^2-\sqrt{(u^2+v^2)(u^2+v^2-4u^2v^2)}}$$

Square to disc mapping:

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



Disc to square mapping:

$$x = Re\left(\frac{1-i}{-K_e}F\left(\cos^{-1}\left(\frac{1+i}{\sqrt{2}}(u+v\,i)\right),\frac{1}{\sqrt{2}}\right)\right)+1$$

$$y = Im\left(\frac{1-i}{-K_e}F\left(\cos^{-1}\left(\frac{1+i}{\sqrt{2}}(u+v\,i)\right),\frac{1}{\sqrt{2}}\right)\right)-1$$

Square to disc mapping:

$$u = Re\left(\frac{1-i}{\sqrt{2}}\,cn\left(K_e\frac{1+i}{2}(x+y\,i)-K_e,\frac{1}{\sqrt{2}}\right)\right)$$

$$v = Im\left(\frac{1-i}{\sqrt{2}}\,cn\left(K_e\frac{1+i}{2}(x+y\,i)-K_e,\frac{1}{\sqrt{2}}\right)\right)$$

where **F** is the incomplete Legendre elliptic integral of the 1st kind
cn is a Jacobi elliptic function

$$K_e = \int_0^{\frac{\pi}{2}} \frac{dt}{\sqrt{1-\frac{1}{2}\sin^2 t}} \approx 1.854$$

Figure 3: Circular Mapping a) Simple Stretching b) FG-Squircular c) Schwarz-Christoffel Mapping

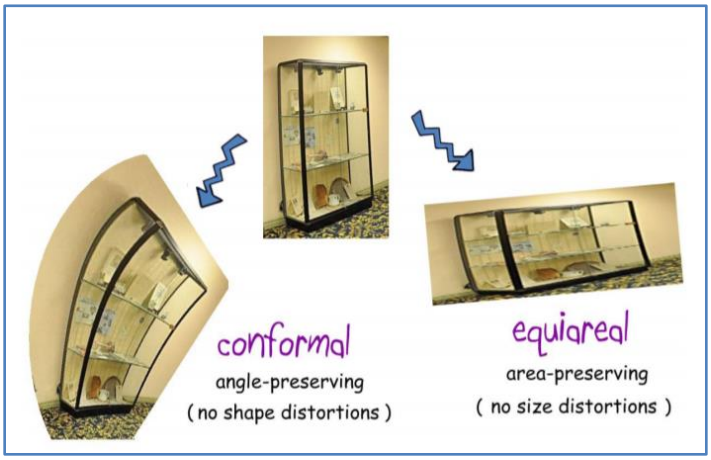


Figure 4: Difference - Conformal and Equiareal Manipulation

There are 2 types of mappings. Conformal and Equiareal. While equiareal mapping preserves the overal area and dimesnions of the original maps, but the image looks heavily distorted as the angles are not preseved.

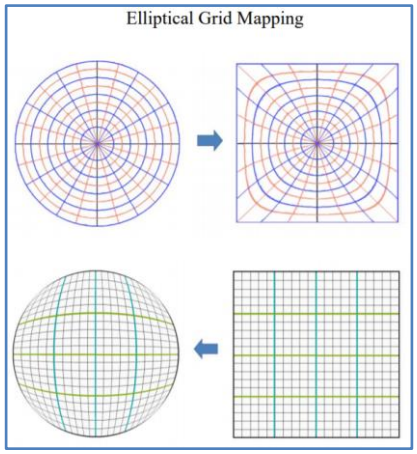
While doing a gemetric process, there are 2 things to be taken care about:

1. The mappings should be **easily computable**
2. The mappings should be **reversible**
3. The mapping should be **angle preserving**

In all of the above three mapping, there are two problems:

1. The above mappings **are not angle-preserving**
2. The above mappings **are not easily computable**
3. **The corners are squished a lot** while converting to Circle from Square, specially in Simple stretching

There is a 4th type of mapping called as Elliptical Grid Mapping. It overcomes almost all the disadvantages of other mappings. It is not perfectly angle-preserving but invloves significantly fewer computations.



Disc to square mapping:

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

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

Square to disc mapping:

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

Figure 5: Conversion process for Elliptical Grid Mapping

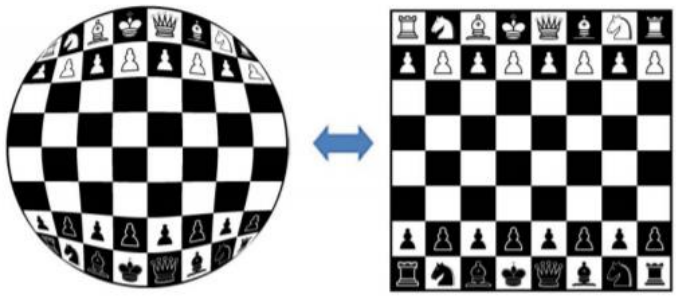


Figure 6: A Circular warping operation should reversible in nature

For the above type of circular warping method, there exists both types of operations:

Operation A - Convert a Square into a Disc
Operation B – (Inverse) Convert a Disc into a Square

Note: But in any case, the mapping alone does not guarantee a **smooth image**. Since the coordinates are returned in a float values and the arrays are accessed with integer values, some of the pixels are returned black. Hence, one must use **bilinear interpolation** to get the best possible result.

For implementing the above functions, a program in C++ was implemented. Following is the pseudocode for the algorithm:

1. User enters the image dimensions and the type of operations he wants
   a) Warp to Circle/Ellipse
   b) De-warp to Square/Rectangle
2. Read the image byte-by-byte in an array
3. Change from Image Space (I, J) to Cartesian Coordinates Space (X, Y)

```
float xCartesian = (float)j + 0.5;
float yCartesian = (float)height - 0.5 - (float)i;
```

4. Calculate the new coordinates UV according to the mapping
5. Convert back from Cartesian Coordinates Space (X, Y) to Image Space (I, J)

```
float newI_exact = (float)height - 0.5 - newY;
float newI_nearest = roundf( newI_exact*100 )/100;
```

6. Use bilinear interpolation to sample the input image

```
int delta_newI = newI_exact - newI_nearest;
int newI = (int)(newI_nearest);

float newJ_exact = newX - 0.5;
float newJ_nearest = roundf( newJ_exact*100 )/100;
int delta_newJ = newJ_exact - newJ_nearest;
int newJ = (int)(newJ_nearest);

if(newI>height-1)
      newI = height-1;

if(newI<0)
      newI = 0;

if(newJ>width-1)
      newJ = width-1;

if(newJ<0)
      newJ = 0;

int tempI = newI+1;
```

```
int tempJ = newJ+1;

if(tempI > height-1)
      tempI = height-1;

if(tempJ > width-1)
      tempJ = width-1;

for(int k=0; k<channels; k++){

resultImageVector[i][j][k] =
(unsigned char)
( delta_newI *
(delta_newJ*ImageVector[newI][newJ][k] +
(1-
delta_newJ)*ImageVector[newI][tempJ][k])
+
(1-delta_newI)*(delta_newJ*ImageVector[tempI][newJ][k] +
(1-delta_newJ)*ImageVector[tempI][tempJ][k]) );
```

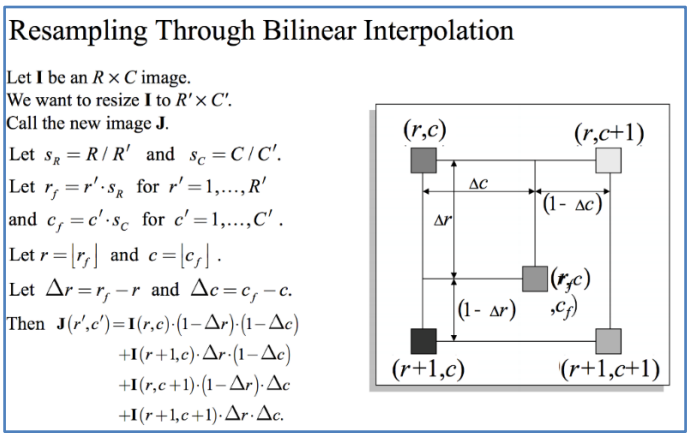


Figure 7: The maths behind Bilinear Interpolation (Source: Ryan Phan)

7. Result vector is written to a file

Results and Discussion

panda.raw



puppy.raw



tiger.raw

experimental image – kriti.raw

Figure 8: a) original Image b) Warped to Circle c) De-warped

Observations

1) Images without bilinear interpolation have lots of empty spots. This is because the mapping in done in floating point values but the array access is done in integer.
2) **There are some artefacts present on the boundary of the de-warped images** as compared to original image. It is because of the mapping being limited in computation of floating point values.
3) **Another reason is introduction of circular defects,** because a circle cannot be perfectly warped to a line
4) The mappings are reversible in nature, the boundaries and centres of the warped, de-warped and original images match.

Figure 9: Mapping without bilinear interpolation

Part B: Homographic Transformation and Image Stitching

Abstract and Motivation

Changing the image plane is an advanced operation of image processing. Such operations enable us to geometrically modify an image in three dimensions. It allows us to perform operation like rotate an image in Y-axis, perform shear operation. Matching a plane of an image with another allows us to perform panoramic stitching.

Figure 10: Example showing image stitching

Approach and Procedures

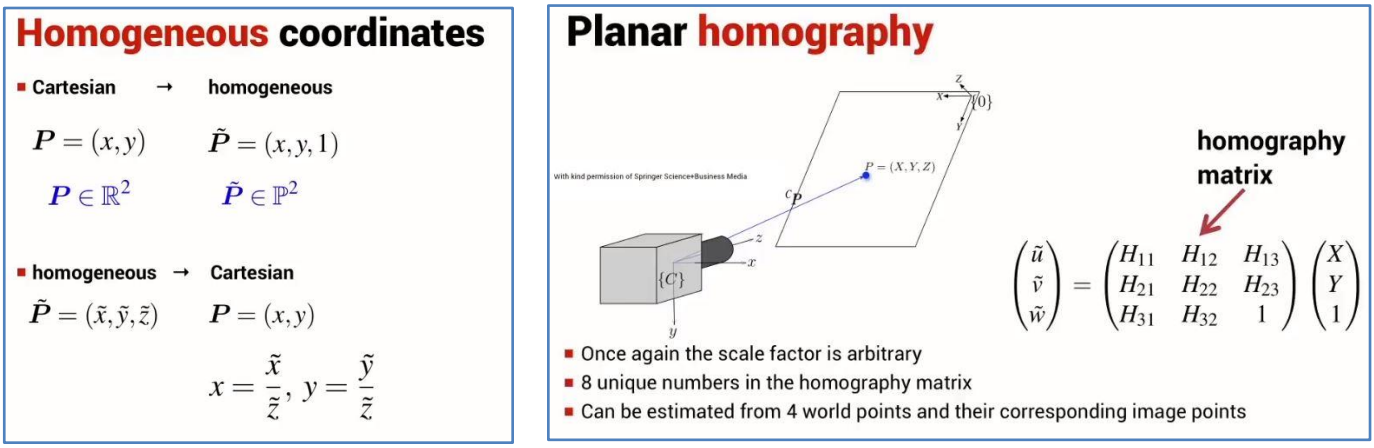


Figure 11: a) Conversion to Homogeneous Coordinate System b) Planar Homography

Planar Homography is a technique that is used to match the planes of two images. If there are at least 4 points in both the images which belong to a common plane, then we can map one image to another. But, since the operation occurs in a 3D space, we first must convert the 2D points to 3D space by using the homogeneous coordinate system. Image planes are matched using Homography and the points are converted back to 2D points.

Bilinear Interpolation is a technique to smoothly manipulate an image. When a geometric operation is being performed on an image, sometimes the pixels are left empty. Now in normal manipulation algorithms the pixels are copied over. This produces several artefacts in the image and the image overall looks blocky and has several black

spots in between. On the other hand, using Bilinear Interpolation, one can fill the empty pixels by interpolating or superimposing the values of nearby pixels.

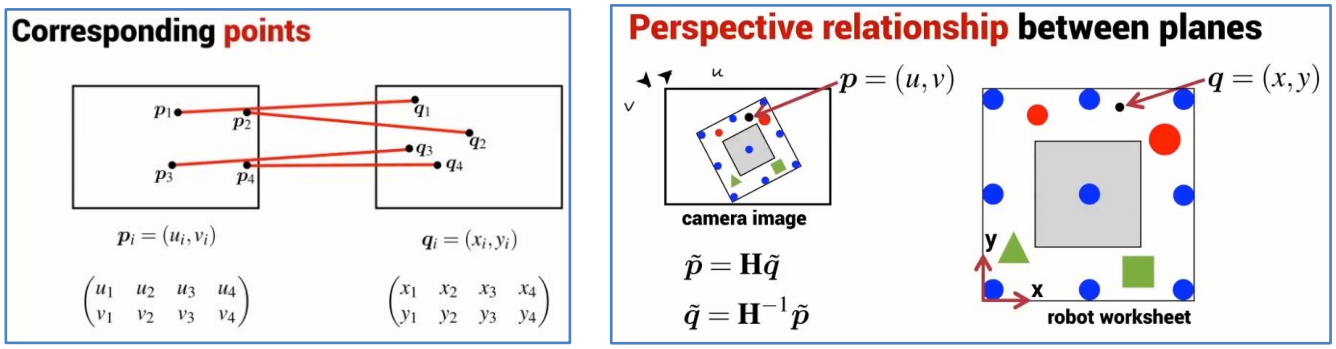


Figure 12: Planar matching

For the above type of method, there exists both types of operations:

Operation A – Map an image from plane 1 to plane 2
Operation B – (Inverse) Map an image from plane 2 to plane 1

Note: But in any case, the mapping alone does not guarantee a **smooth image**. Since the coordinates are returned in a float values and the arrays are accessed with integer values, some of the pixels are returned black. Hence, one must use **bilinear interpolation** to get the best possible result.

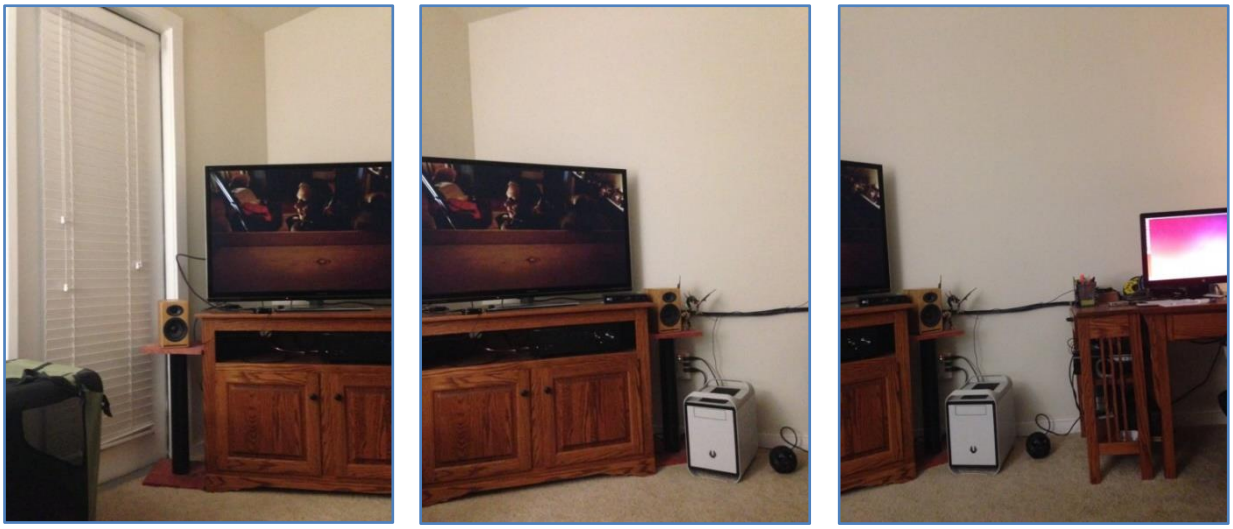


Figure 13: Images to be stitched a) Left b) Middle c) Right

For implementing the above functions, a program in C++ was implemented. Following is the pseudocode for the algorithm:

1. User enters the image dimensions and 3 input images
2. Read the image byte-by-byte in an array
3. Place the middle image in a black board having a sufficiently larger size than all the images combined together
4. Find the common active points between the images

Pairing between Middle and Left Images		
Middle Image:	**Left Image:**	**Description**
Point 1 = 975, 1117	Point 1 = 277, 467	Stand Shelf Door Top-Left Corner
Point 2 = 978, 1210	Point 2 = 279, 557	Stand Shelf Door Bottom-Left Corner
Point 3 = 1077, 1105	Point 3 = 371, 470	Stand Shelf Door Top-Right Corner
Point 4 = 1079, 1199	Point 4 = 373, 563	Stand Shelf Door Bottom-Right Corner
Point 5 = 1018, 660	Point 5 = 322, 31	Where walls meet
Point 6 = 1103, 1062	Point 6 = 398, 429	Shiny point above the right Shelf Door knob
Point 7 = 992, 834	Point 7 = 295, 203	Shiny point on TV left
Point 8 = 1160, 845	Point 8 = 465, 200	Left Shelf Door knob

Pairing between Middle and Right Images		
Middle Image:	**Right Image:**	**Description**
Point 1 = 1288, 1134	Point 1 = 134, 493	Inverter Top-Left Corner
Point 2 = 1293, 1209	Point 2 = 138, 567	Inverter Bottom-Left Corner
Point 3 = 1345, 1138	Point 3 = 189, 491	Inverter Top-Right Corner
Point 4 = 1350, 1218	Point 4 = 193, 568	Inverter Bottom-Right Corner
Point 5 = 1208, 846	Point 5 = 47, 200	TV Top right corner of screen
Point 6 = 1279, 1076	Point 6 = 125, 436	Light Behind TV
Point 7 = 1223, 1029	Point 7 = 66, 392	Top-Right corner of empty black void
Point 8 = 1226, 1067	Point 8 = 69, 432	Bottom-Right corner of empty black void

Figure 14: Common Active Points between Left, Middle and Right images

5. Using the above points, Homography matrices (H) where calculated for both left-middle images and right-middle mappings.
6. Since traversal happens on the blackboard images, there was a need to calculate inverse Homography matrices (H^{-1}) for both middle-left and middle right mappings. (See MATLAB file ImageStitching.m)

Inverse Homography Matrix - Left to Middle		
0.505215	0.005251	-284.408458
-0.181230	0.709900	-262.099172
-0.000544	-0.000015	1.308559

Inverse Homography Matrix - Right to Middle		
1.111073	-0.01377	-1280.1
0.183995	1.035588	-896.924
0.000468	-6.74E-05	0.500662

Figure 15: Inverse Homography Matrix (H^{-1}), this was calculated using MATLAB

7. Change from Image Space (I, J) to Cartesian Coordinates Space (X, Y)

```
float xCartesian = (float)j + 0.5;
float yCartesian = (float)height - 0.5 - (float)i;
```

8. Calculate the new coordinates UV according to the mapping

```
//Convert back to Cartesian Coordinates
float newX_left = newHomogeneousXY_left[0][0]/newHomogeneousXY_left[2][0];
float newY_left = newHomogeneousXY_left[1][0]/newHomogeneousXY_left[2][0];

float newX_right = newHomogeneousXY_right[0][0]/newHomogeneousXY_right[2][0];
float newY_right = newHomogeneousXY_right[1][0]/newHomogeneousXY_right[2][0];
```

```
//Change from XY space to IJ space
float newI_left_exact = (float)height - 0.5 - newY_left;
float newI_left_nearest = roundf( newI_left_exact*100 )/100;
int delta_left_newI = newI_left_exact - newI_left_nearest;
int newI_left = (int)(newI_left_nearest);

float newJ_Left_exact = newX_left - 0.5;
float newJ_left_nearest = roundf( newJ_Left_exact*100 )/100;
int delta_left_newJ = newJ_Left_exact - newJ_left_nearest;
int newJ_left = (int)(newJ_left_nearest);

float newI_right_exact = (float)height - 0.5 - newY_right;
float newI_right_nearest = roundf( newI_right_exact*100 )/100;
int delta_right_newI = newI_right_exact - newI_right_nearest;
int newI_right = (int)(newI_right_nearest);

float newJ_right_exact = newX_right - 0.5;
float newJ_right_nearest = roundf( newJ_right_exact*100 )/100;
int delta_right_newJ = newJ_right_exact - newJ_right_nearest;
int newJ_right = (int)(newJ_right_nearest);
```

9. Convert back from Cartesian Coordinates Space (X, Y) to Image Space (I, J)

```
float newI_exact = (float)height - 0.5 - newY;
float newI_nearest = roundf( newI_exact*100 )/100;
```

10. Use bilinear interpolation to sample the input image. Also, the images pixels must have averaged to get smoother image stitch at the image boundaries.

```
if( (newI_left >= 0) && (newJ_left >= 0) && (newI_left < height) && (newJ_left <
width) ){

unsigned char leftPixel = (unsigned char) ( delta_left_newI *
(delta_left_newJ*ImageVector1[newI_left][newJ_left][k] + (1-
delta_left_newJ)*ImageVector1[newI_left][tempJ_left][k]) +
                                    (1-
delta_left_newI)*(delta_left_newJ*ImageVector1[tempI_left][newJ_left][k] + (1-
delta_left_newJ)*ImageVector1[tempI_left][tempJ_left][k]) );

//blackBoardImageVector[i][j][k] = leftPixel;

if( i>=height && i<2*height && j>=2*width && j<3*width ){
      int pixelValue = (int)leftPixel + (int)blackBoardImageVector[i][j][k];
      pixelValue = pixelValue/2;
```

```
      blackBoardImageVector[i][j][k] = (unsigned char)pixelValue;
} else {
      blackBoardImageVector[i][j][k] =
leftPixel;
}

      newJ = 0;

int tempI = newI+1;
int tempJ = newJ+1;

if(tempI > height-1)
      tempI = height-1;

if(tempJ > width-1)
      tempJ = width-1;

for(int k=0; k<channels; k++){
```

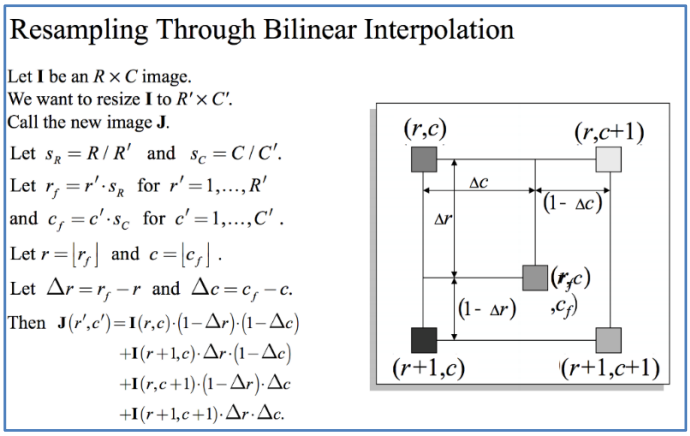


Figure 16: The maths behind Bilinear Interpolation (Source: Ryan Phan)

11. Result vector is written to a file

Results and Discussion

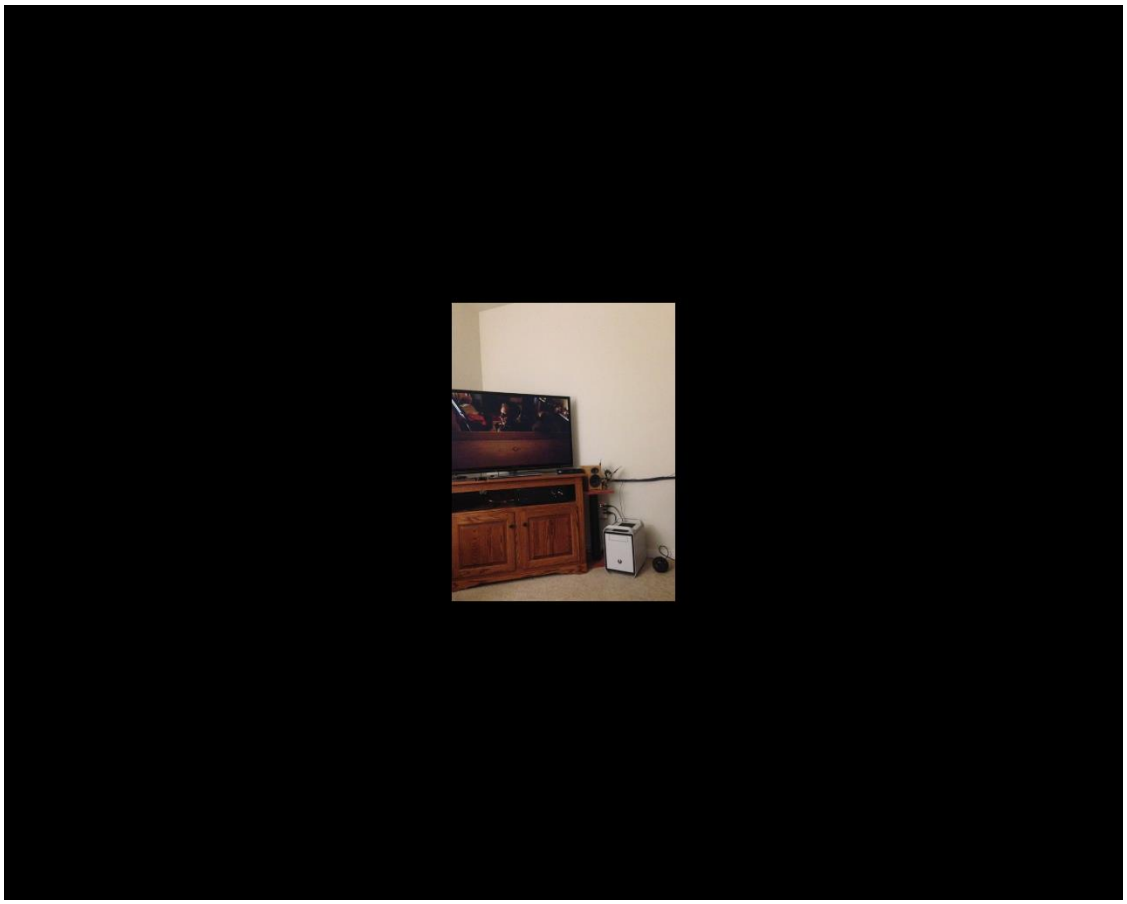


Figure 17: Middle image placed on the black-board

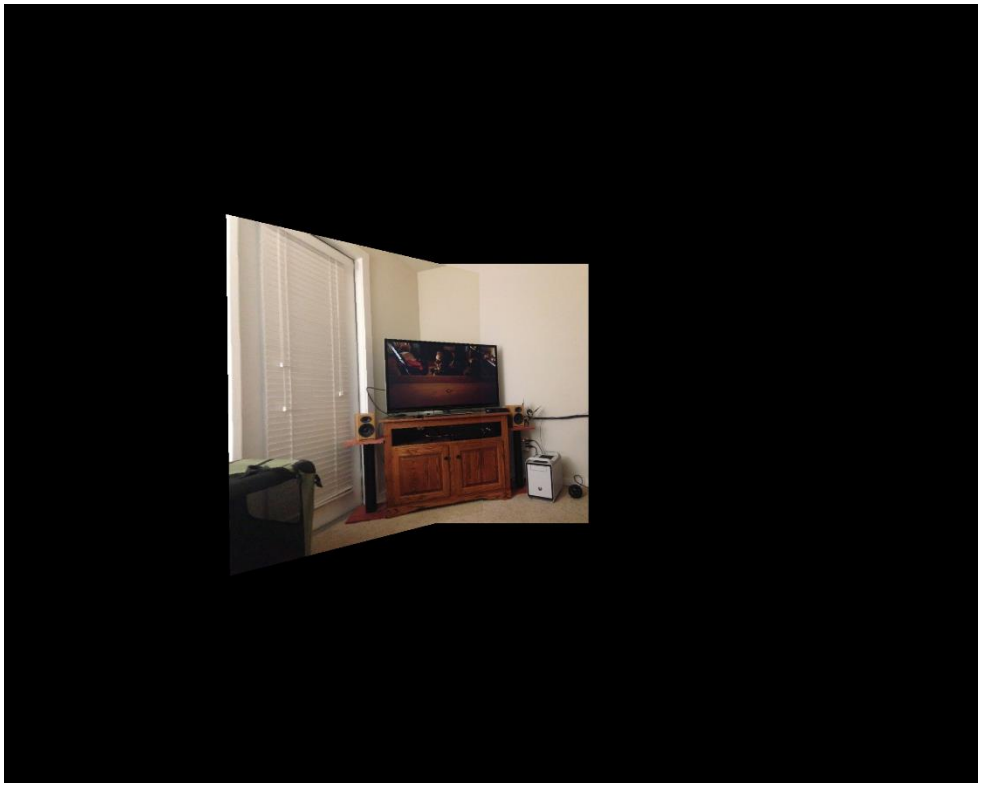


Figure 18: Left image overlaid over Middle image

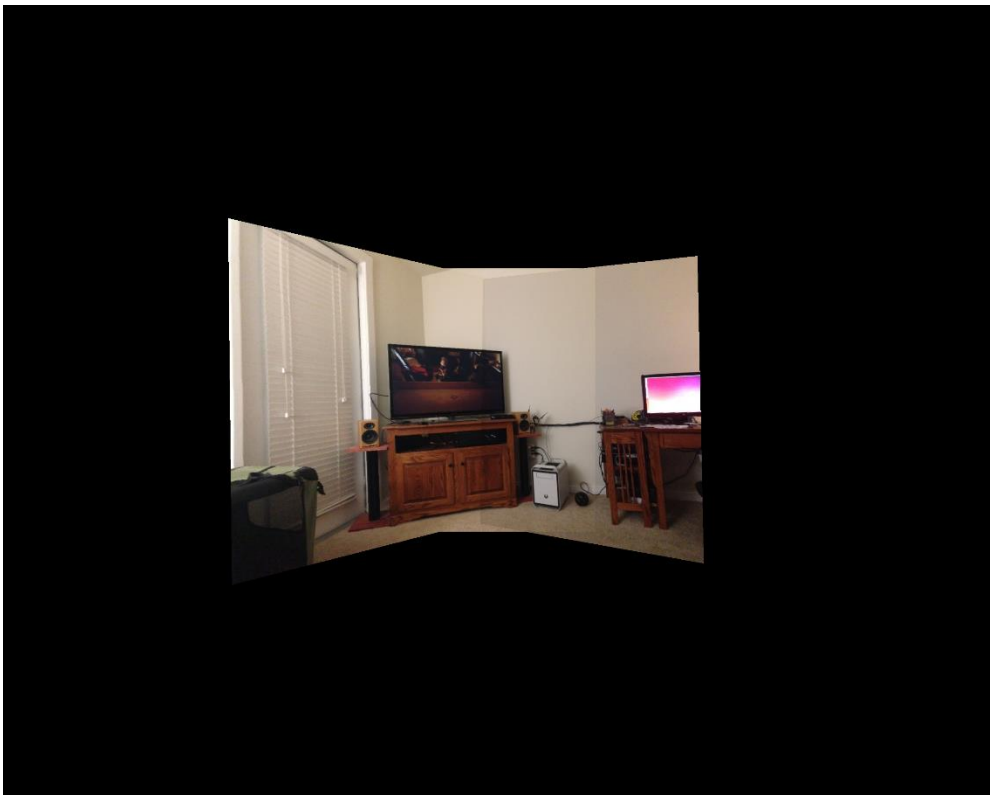


Figure 19: Right Image overlaid on the Composite of Left and Right images

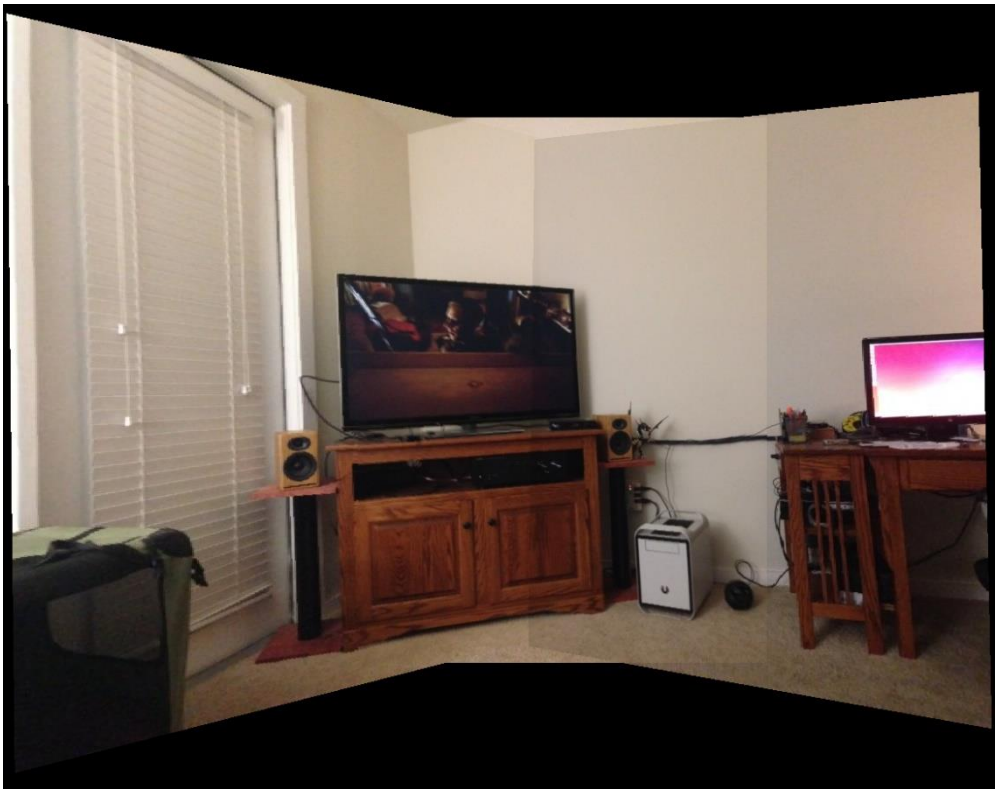


Figure 20: Composite of Stitched images – Cropped

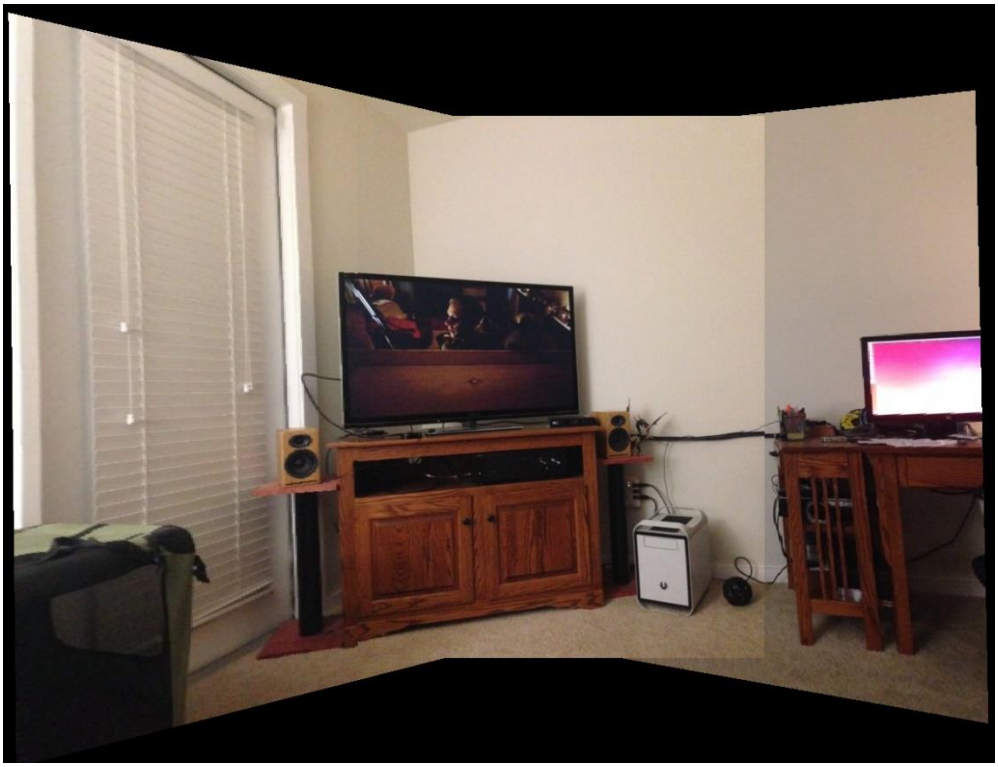


Figure 21: Composite of Stitched images – Smoothened

Observations

1) The right image is a bit darker than the other two images which leads to patches at the images boundary
2) If averaging of images is not done at the image boundaries, then there is a distinct colour change at the boundary of the images.
   a) The best way to merge is histogram matching of right image with middle image and merge
   b) Other way, which is currently used in this report, is to weight the pixel values. But this is only effective at the image boundaries. The right image will still appear darker.
      Pixel value = 90 % of Middle Image Pixel value + 10% of Right Image Pixel value
3) Images without bilinear interpolation have lots of empty spots. This is because the mapping in done in floating point values but the array access is done in integer.
4) The mappings are reversible in nature. All the images are recoverable by using the inverse operation.

Conclusion

The mappings are reversible in nature, the boundaries and centres of the warped, de-warped and original images match. The mappings for Homographic plane matching are also reversible in nature. Bilinear interpolation is a handy technique to smoothen out the image. Implementations for both image warping and image stitching have been done in C++.

Histogram matching can be done to improve the result for image stitching.

Resources

Image Warping - https://arxiv.org/ftp/arxiv/papers/1509/1509.06344.pdf

Homogeneous Coordinates and Homography - https://robotacademy.net.au/masterclass/the-geometry-of-image-formation/?lesson=780

Problem 2: Digital Halftoning

Part A: Basic Dithering

Abstract and Motivation

Digital Halftoning - In practical terms, in the field of image processing, while designing printers, it is necessary to use some form of error diffusion of dithering to process the image. Because while printing, we don't have the luxury of printing up to 255 levels. In a printer, the "dot" is either there or not there. There is no mid-level. Both dithering and error diffusion allow us to produce images that have fully saturated pixels, i.e. pixels of value 255, belonging to a certain color. But the algorithm works in way such that it maintains the overall integrity of the image while saturating the pixels. Thus, the overall look and feel of the image is maintained.

A typical use of **dither** is converting a greyscale image to black and white, such that the density of black dots in the new image approximates the average grey level in the original. According to Wikipedia, dither is an intentionally applied form of noise used to randomize quantization error, preventing large-scale patterns such as color banding in images. Dither is routinely used in processing of both digital audio and video data, and is often one of the last stages of mastering audio to a CD.

Approach and Procedures

There are 3 methods for Dithering that have been explored here.

`Fixed Thresholding`

If pixel value > threshold, new pixel value = 255 else it becomes 0.

`Random Thresholding`

This is a bit different from fixed thresholding. In this case, a threshold level is decided for each pixel. **The threshold is decided randomly for each pixel**. All the pixels are assigned a value of either 0 or 255 based on the threshold. If pixel value > threshold, new pixel value = 255 else it becomes 0.

`Bayer Dithering Matrices`

This is an empirical method. A short summary denoting the steps of the method is listed here:

1. An even numbered kernel-matrix is defined. This kernel can be of size 2, 4 or 8. The matrixes have been found using the recursive Bayer -method:

$$I_2(i,j) = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix} \qquad I_{2n}(i,j) = \begin{bmatrix} 4 * I_n(x,y) + 1 & 4 * I_n(x,y) + 2 \\ 4 * I_n(x,y) + 3 & 4 * I_n(x,y) \end{bmatrix}$$

Figure 1: Recursive formula for calculating Bayer Matrices, n = 2, 4 or 8

2. The kernel is convolved over the whole image pixel-by-pixel
3. As this is done, a new threshold level is generated at run-time for each pixel
4. Assigning a color tone based on threshold

   Dual Toning - Pixels are assigned a value of either 0 or 255

   IF pixel value > 127 => new pixel value = 255 **ELSE** new pixel value = 0

   Quad Toning - Pixels are assigned a value of either 0, 85, 170 or 255

   IF pixel value > 42 => new pixel value = 85 **ELSE** new pixel value = 0

   ELSE IF pixel value > 127 => new pixel value = 170 **ELSE** new pixel value = 85

   ELSE IF pixel value > 212 => new pixel value = 255 **ELSE** new pixel value = 170

Results and Discussion

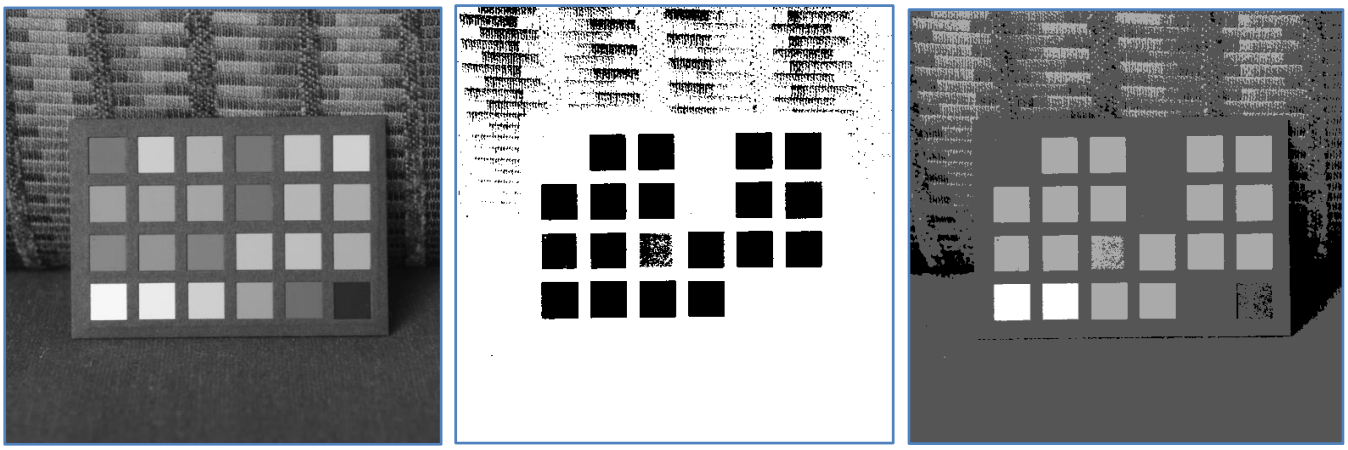


Figure 2: a) Original image - colorchecker.raw b) Fixed Threshold Dual-tone c) Fixed Threshold Quad-tone

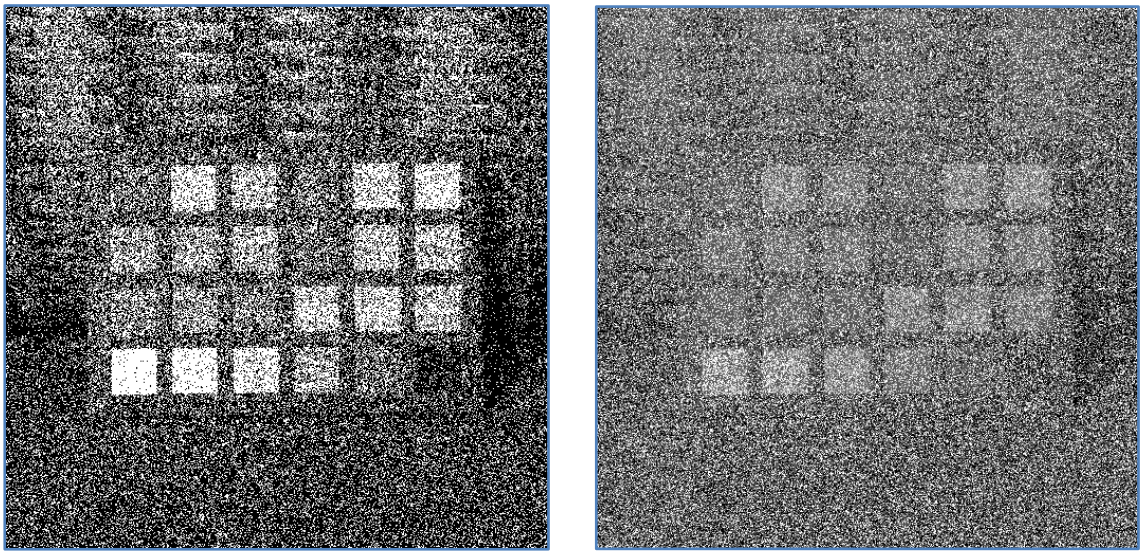


Figure 3: a) Original image - colorchecker.raw b) Random Threshold Dual-tone c) Random Threshold Quad-tone

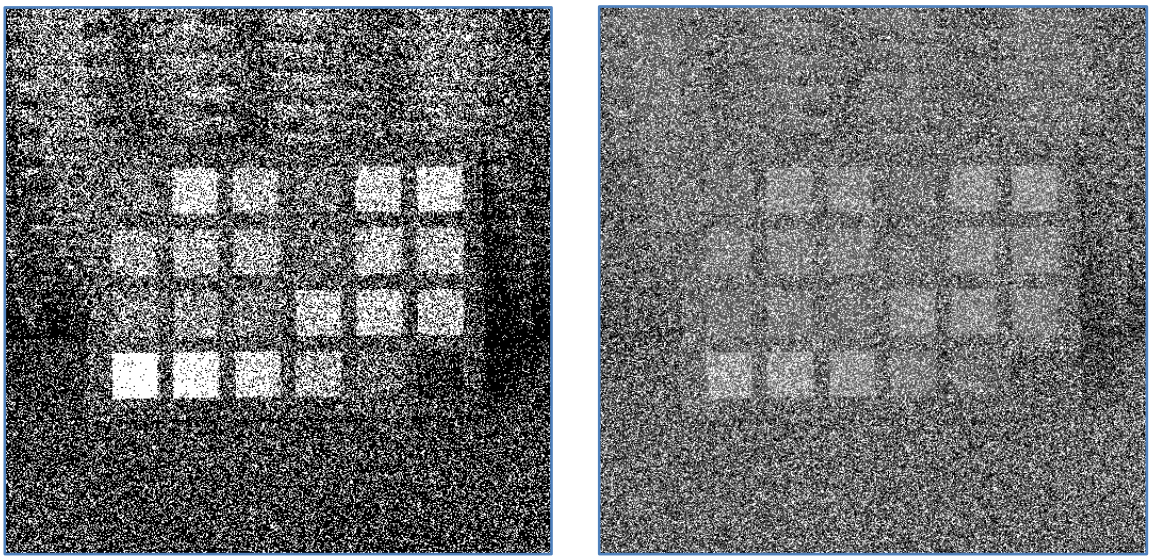


Figure 4: a) Original image - colorchecker.raw b) Random Threshold Dual-tone c) Random Threshold Quad-tone

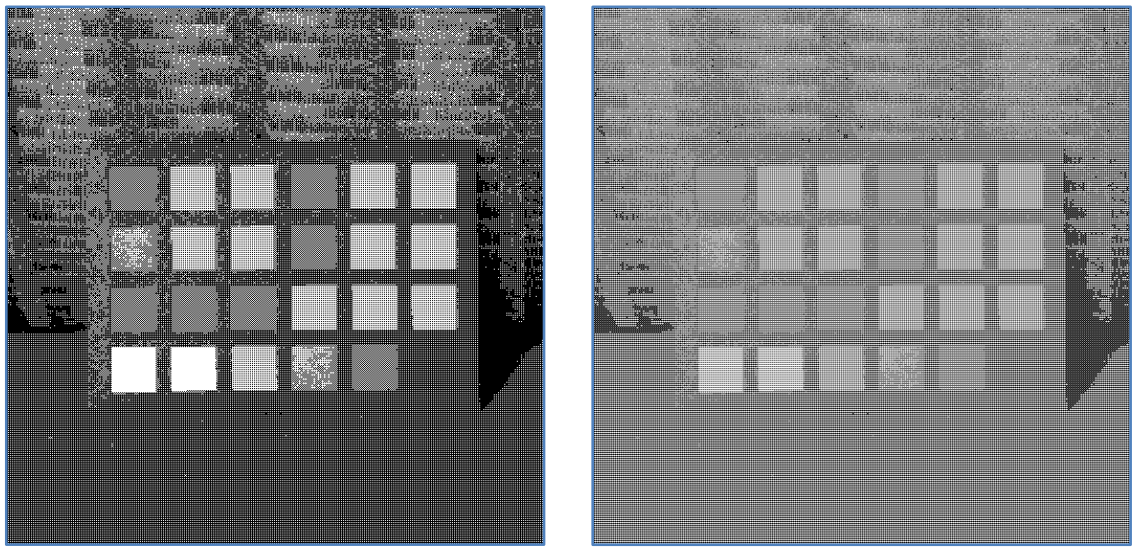


Figure 5: Using 2nd Order Bayer Matrix a) Dual-tone b) Quad-tone

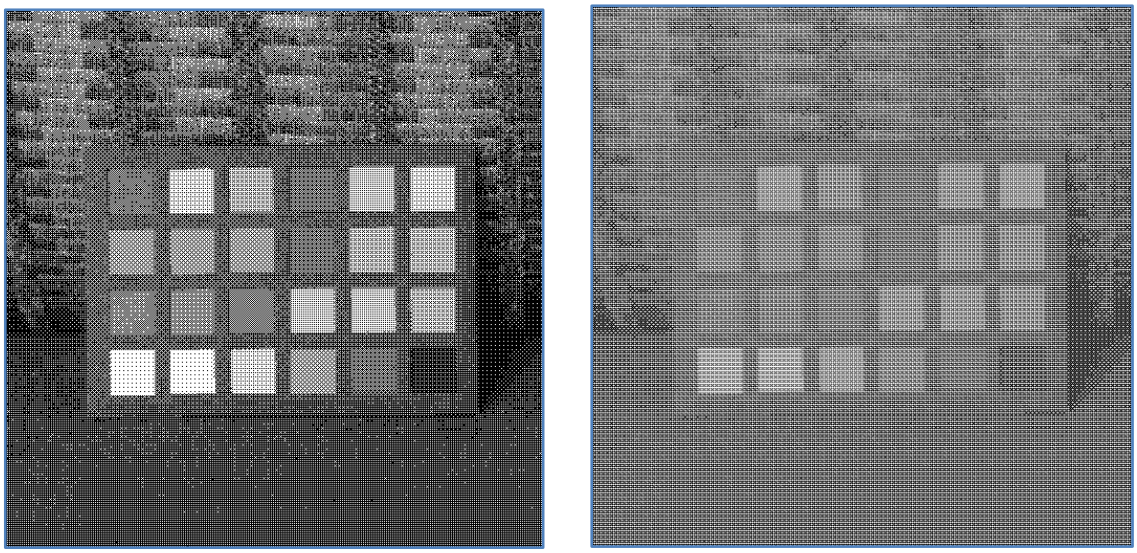


Figure 6: Using 4th Order Bayer Matrix a) Dual-tone b) Quad-tone

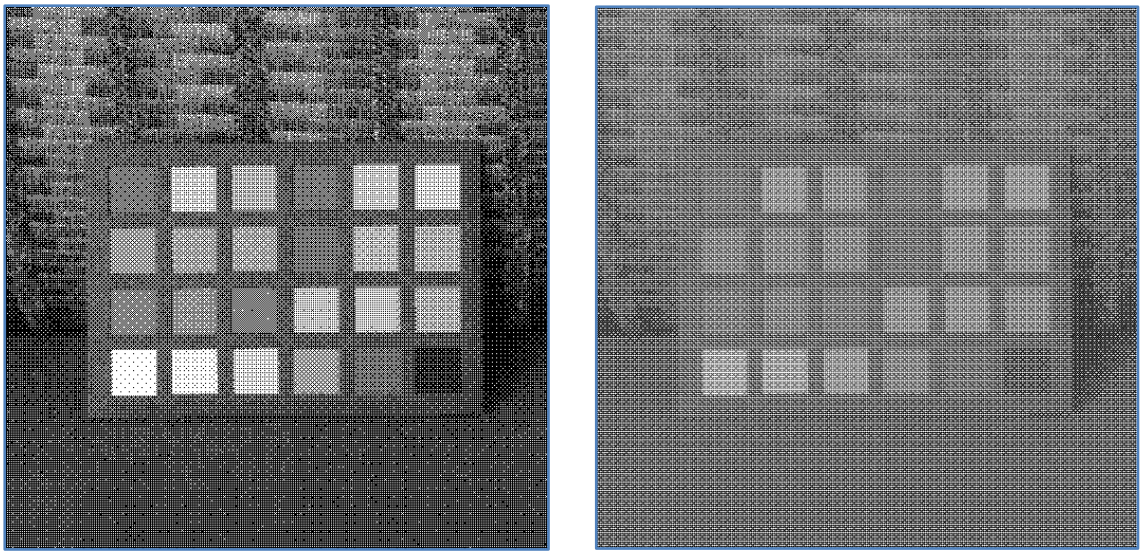


Figure 7: Using 8th Order Bayer Matrix a) Dual-tone b) Quad-tone

Observations

1) **Dual tone vs Quad tone - Dual Tone Results look better overall:** There is marked difference between dual and quad tone results. The quad tone results not only look duller, they appear a lot whiter than they should be. The reason could be that more pixels are shifter towards shades whiter than themselves, thus making the image look overall washed off. The only time when quad tone results look better is when the dithering is done using fixed thresholds.
2) **Fixed vs Random vs Bayer Matrix Method – Results using Bayer Matrices look better overall:** There is significant improvement in the images produced using the Bayer matrices. The dithering noise is distributed quite evenly. Although the noise distributed evenly is Random thresholding as well, but the image looks a lot grainier and courser than the one produced using dithering matrices.
    a. Image produced with 8^{th} order Bayer matrix is the best
    b. Image produced with 2^{nd} order Bayer matrix is missing one color, bottom right tile (colorchecker)
    c. Images produced using random thresholds are different every time the program is run. Although the difference is not apparent to the human eye, but a side-by-side comparison reveals that.
    d. Quad tone image in fixed thresholding is better than dual tone image

Part B: Error Diffusion and Part C: Separable Error Diffusion for Color Halftoning

Abstract and Motivation

Error diffusion is another method of halftoning in which the quantization error (residual) is distributed to neighboring pixels that have not yet been processed. Its main use is to convert a multi-level image into a binary image, which can be later used in printing. But it has other applications as well.

According to Wikipedia, Error diffusion is classified as an area operation, because what the algorithm does at one location influences what happens at other locations. This means buffering is required, and complicates parallel processing. Point operations, such as ordered dither, do not have these complications.

MORE INFO: Error diffusion has the tendency to enhance edges in an image. This can be used to increase sharpness in the image. It can be also used to make text in images more readable than in other halftoning techniques.

Approach and Procedures

In this report, 3 methods for Error Diffusion have been explored.

Floyd-Steinberg

1. The following matrix is used for error diffusion.

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

2. The image is processed in a Serpentine manner which is implemented as below:

```
kernelSize = 3
for(int p=0; p<outputHeight; p++){
        for(int q=0; q<outputWidth; q++){
            unsigned char oldPixel;
            unsigned char newPixel = 0;

            if((diffusionMethod == 1) && (p%2 == 0)){
```

```
                oldPixel =
                SingleChannelVector[p+(kernelSize/2)][width-1-q-(kernelSize/2)][0];
            } else {
                oldPixel =
                SingleChannelVector[p+(kernelSize/2)][q+(kernelSize/2) ][0];
            }
```

3. Basically, for an odd-numbered row, the kernel traverses normally (along the width of the image), but for an even-numbered row, the kernel traverses in the reserve direction
4. Kernel matrix is also reversed when the traversing is being done on an even numbered row as shown below

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

Jarvis Judice Ninjke (JJN)

1. The following matrix is used for error diffusion

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

2. The kernel is traversed normally

Stucki

1. The following matrix is used for error diffusion

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

2. The kernel is traversed normally

Separable Error-Diffusion for Color Halftoning

1. The kernel meant for Floyd-Steinberg method is used
2. The whole color-space is considered as a cube RGBW-CMYK. All the colors form vertices of a cube
3. It is determined that to which vertex is the pixel closest to
4. The Euclidean distance between the pixel and the vertex is calculated
5. Pixel is assigned the value of the vertex, the residual (error) is calculated
6. Error is distributed in the neighboring pixels using Floyd-Stenberg method
7. The kernel is traversed is a Serpentine fashion

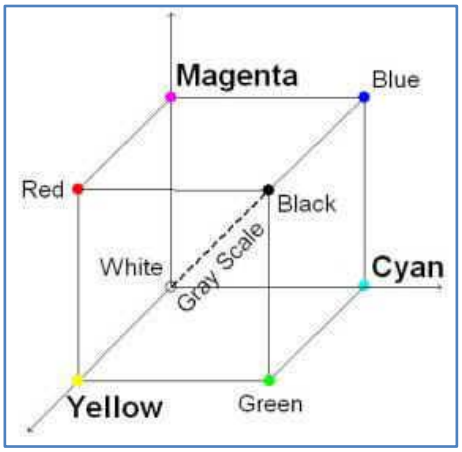


Figure 8: Color Cube with 8 vertices as color vectors

Results and Discussion

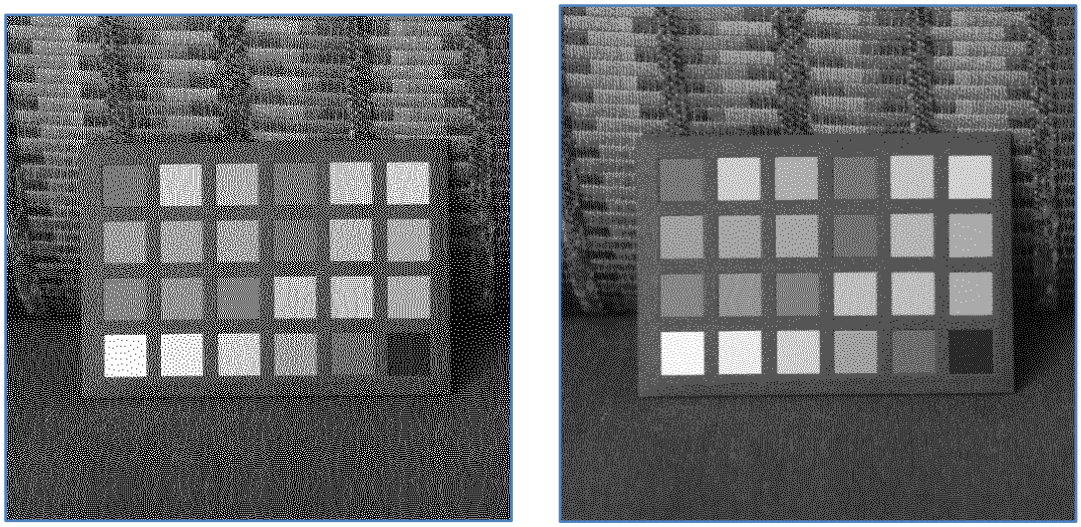


Figure 9: Floyd-Steinberg a) Dual-tone b) Random Threshold Quad-tone

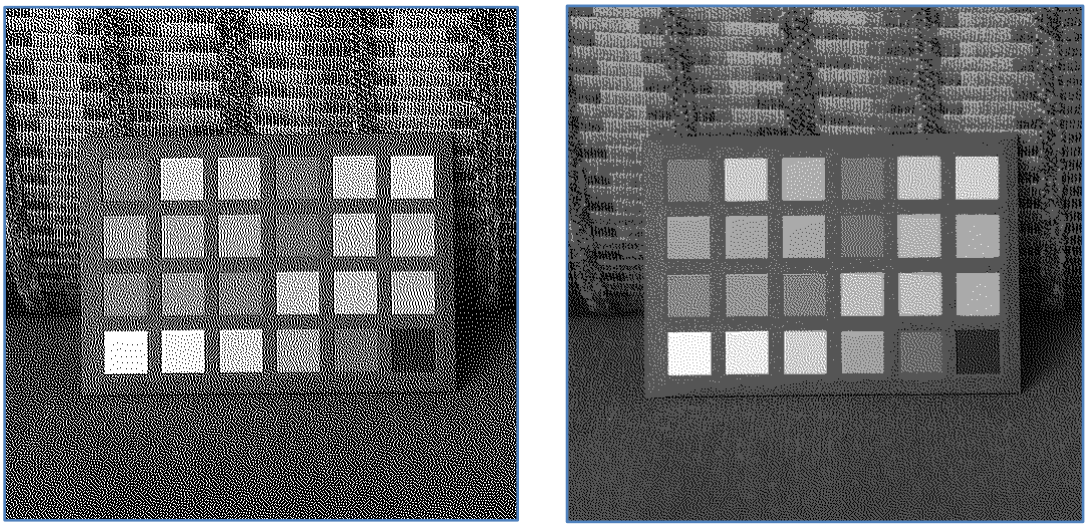


Figure 10: JJN a) Dual-tone b) Quad-tone

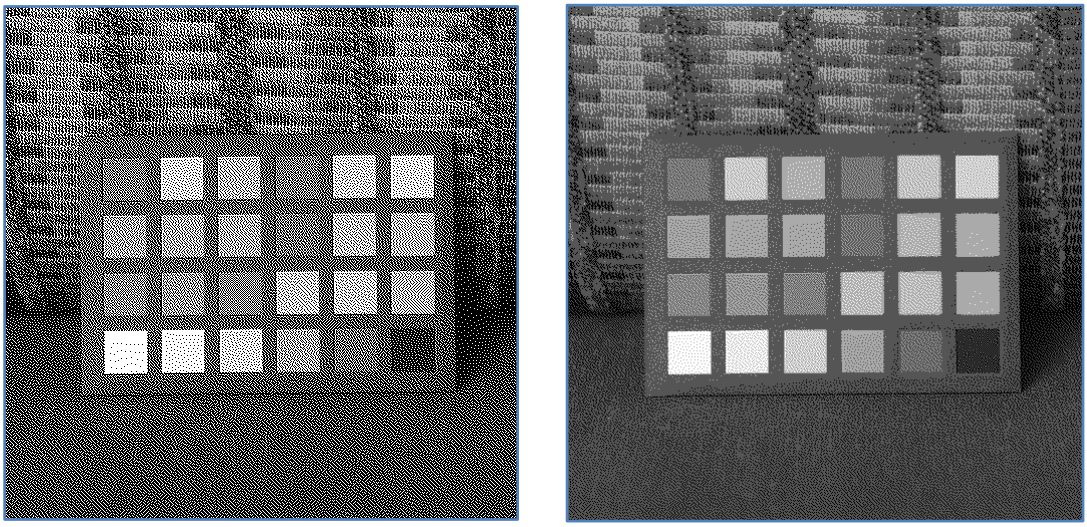


Figure 11: Stucki a) Dual-tone b) Quad-tone

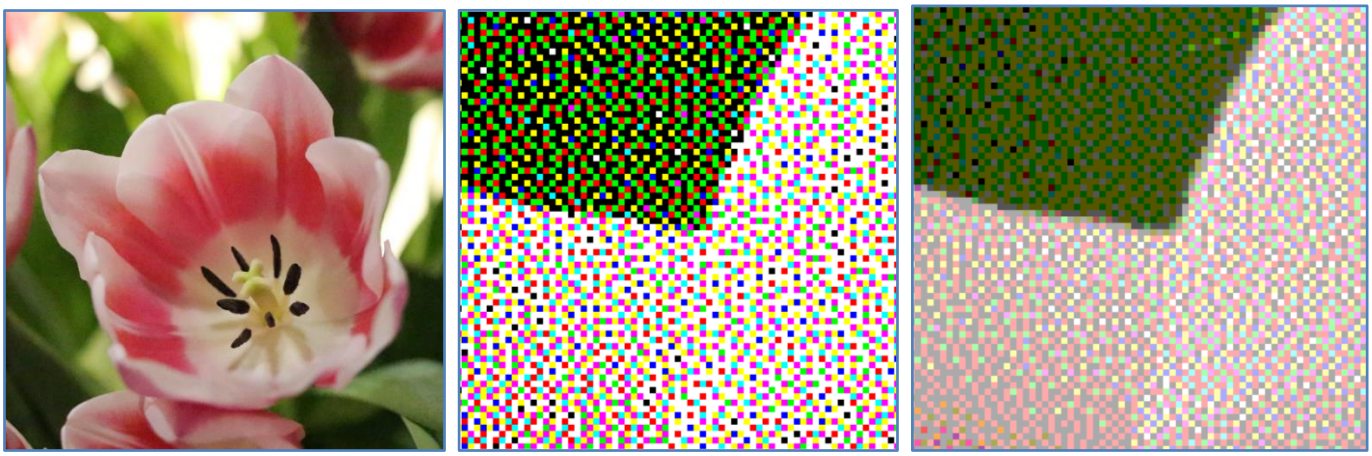


Figure 12: a) Original Image b) Zoomed-In Separable ED - 2 tones c) Zoomed-In Separable ED - 4 tones

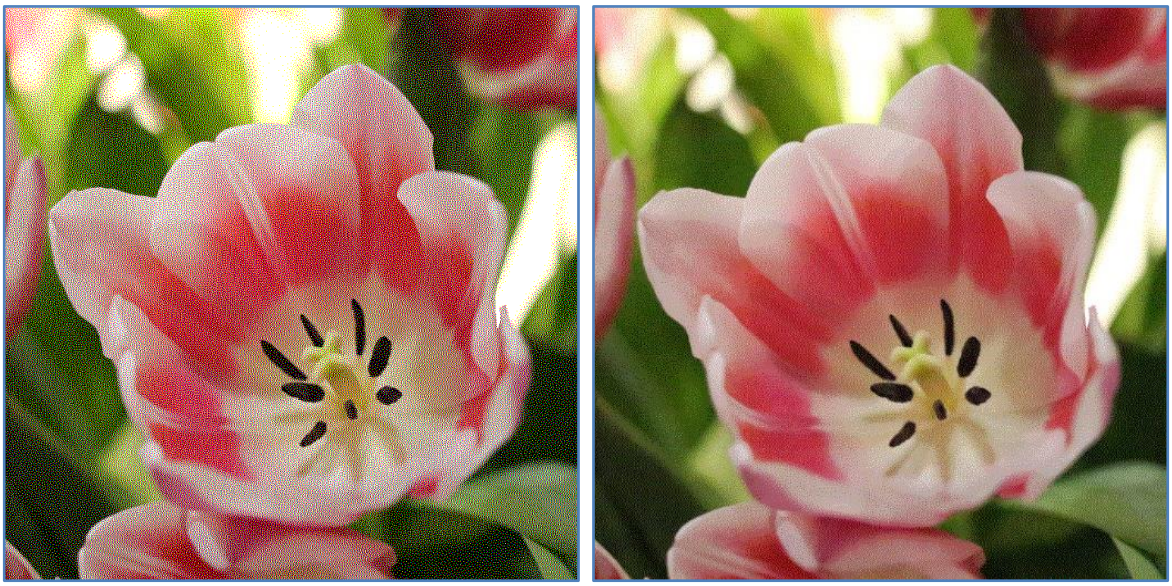


Figure 13: Color Half Toning Using Separable Error Diffusion a) Dual-tone b) Quad-tone

Observations

In the above set of images, it can be observed that:

1. Images processed with Floyd-Steinberg method are the best results. There are little or no speckles in the images. The visual quality is good. It should work good in printers as well because the denominator is 16 which is a power of 2, which is easily calculable by a machine having low processing power.
2. In theory, images produced with JJN and Stucki methods should ideally be produced in more time because of the denominators, 48 and 42, respectively. Such method numb are not easily calculable in printers
3. The zoomed-in dual and quad tone images show that the pixels are quantified to 2 and 4 tones. The RGB pixels can be very easily seen in dual-tone image
4. Quad-tone image is much smoother and closer in visual quality to the original image

Conclusion

Floyd Stein produces bed image and is easy on processor as well. Further improvement:

1) A further improvement can be application of Median filter while diffusing the error to smoothen out image. But this method could produce a highly contrasting image

2) Another method could be kernel having a denominator which is of the order of 2 to power of n could be used

Problem 3: Morphological Processing

Abstract and Motivation

Morphological image processing is a collection of non-linear operations related to the shape or morphology of features in an image. In general, morphological operations rely only on the relative ordering of pixel values, not on their numerical values, therefore, they are especially suited to the processing of binary images. Morphological operations can also be applied to greyscale images such that original pixel values are unknown and therefore their absolute pixel values are of no or minor interest. We just define background and a foreground on images. In some cases, morphological processes even form the basis for low-level computer vision applications.

Morphological techniques probe an image with a small shape or template called a structuring element. The structuring element is positioned at all possible locations in the image and it is compared with the corresponding neighborhood of pixels. Some operations test whether the element "fits" within the neighborhood, while others test whether it "hits" or intersects the neighborhood:

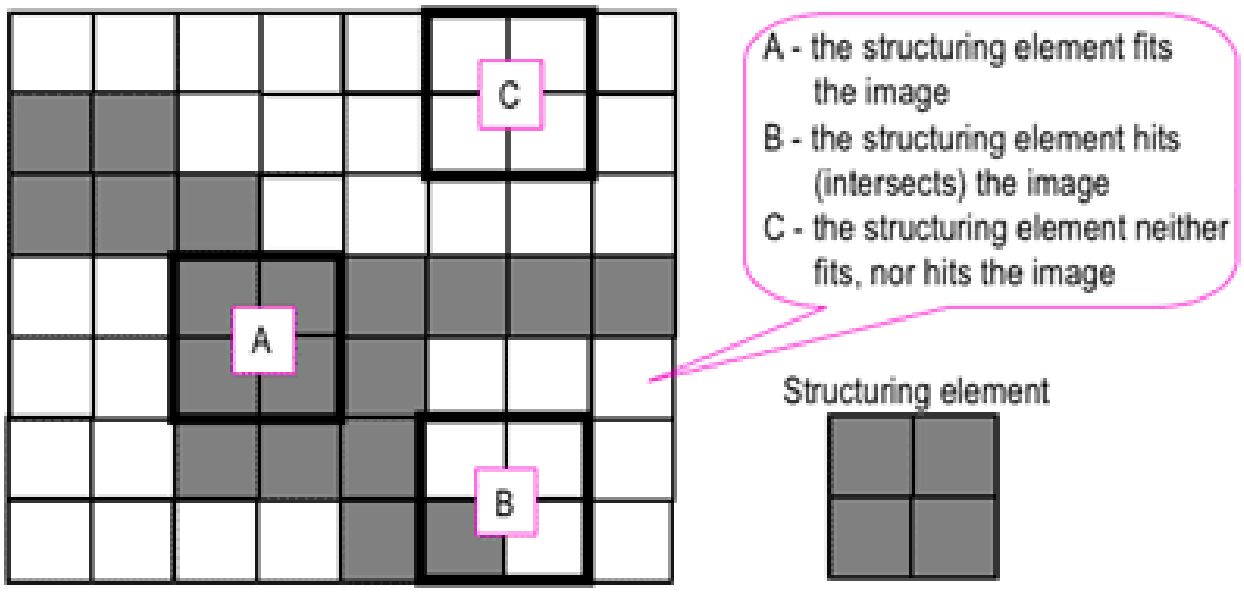


Figure 1: Probing of an image with a structuring element

A morphological operation on a binary image creates a new binary image in which the pixel has a non-zero value only if the test is successful at that location in the input image. The structuring element is a small binary image, i.e. a small matrix of pixels, each with a value of zero or one:

- The matrix dimensions specify the size of the structuring element
- The pattern of ones and zeros specifies the shape of the structuring element
- An origin of the structuring element is usually one of its pixels, although generally the origin can be outside the structuring element

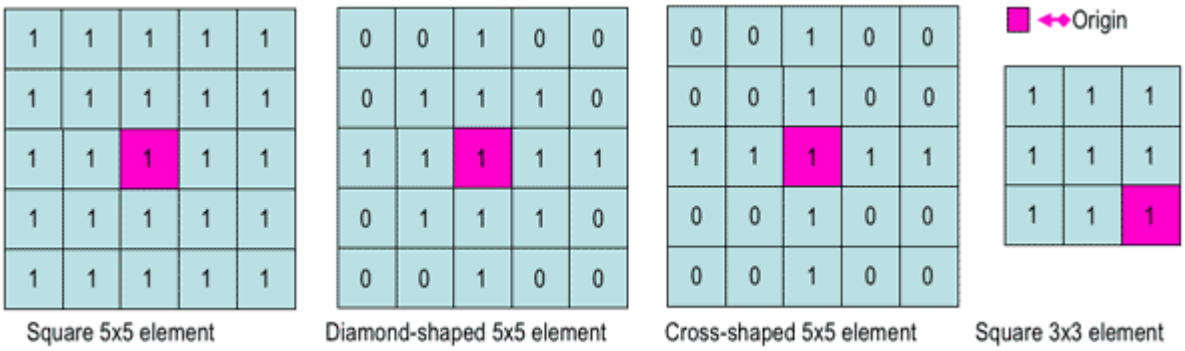


Figure 2: Examples of simple structuring elements

A common practice is to have odd dimensions of the structuring matrix and the origin defined as the center of the matrix. Structuring elements play the same role in morphological image processing as convolution kernels in linear image filtering.

Approach and Procedures

All the subparts involve morphological processing; hence the approach is similar. The approach and formation of code (written in C++) is explained here.

General Process for Shrinking, Thinning, Skeletonizing, Thickening, etc.

The process is carried out in 2 stages. In both the stages:

1) Image is padded with zeros at the boundaries to facilitate a filter operation by a 3x3 kernel. So overall, the operation turns out to be 5x5, because a 3x3 kernel operation followed by another 3x3 is equivalent to 5x5.
2) Images are traversed pixel-by-pixel. With the subject pixel as the center, the 8 neighboring pixels are stacked to form a byte as shown below:

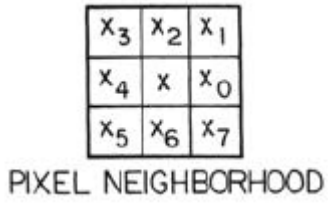


Figure 3: Stacking of neighboring pixels to form a byte

3) The pixel neighborhood defined the bond of the pixel
   A. Presence of a one-up, one-down, one-left and one right increases the value of bond by 2
   B. The corner pixels increase bond value by 1 each
   C. If no neighboring pixels are present, then bond value = 0 => Singleton pixel
      Max bond value = 12 => Completely covered pixel
4) At any stage, the formed pixel byte is compared against the available patterns for that operation:
   A. For each of the operations: shrinking, thinning and skeletonizing, there are different patterns for each value of bond
   B. Hit-and-Miss transform: If the pattern of the stacked pixel matches with the designated patterns, then the pixel is marked for erasure/addition.

C. If we are performing a reduction (controlled erosion) process, i.e. shrinking, thinning or skeletonizing, then we perform erasure of foreground. Else, we do addition to background

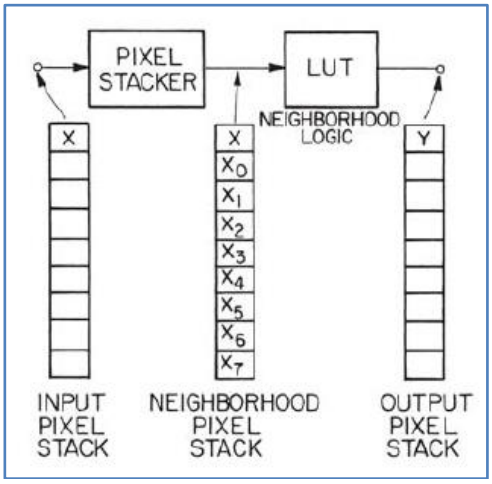


Figure 4: General form of Hit-and-Miss transform

$$G(j,k) = X \cap [\overline{M} \cup P(M, M_0, \ldots, M_7)]$$

Figure 5: Formula used for erasure

The formula showed in figure 5 is main formula that is used for reduction operations. The $\overline{M}$ matrix is obtained after negating the Marker matrix (M) that we get at the end of stage 1. P (M, M_0, … M_7) is an erasure inhibiting logical variable. The detailed description is given below.

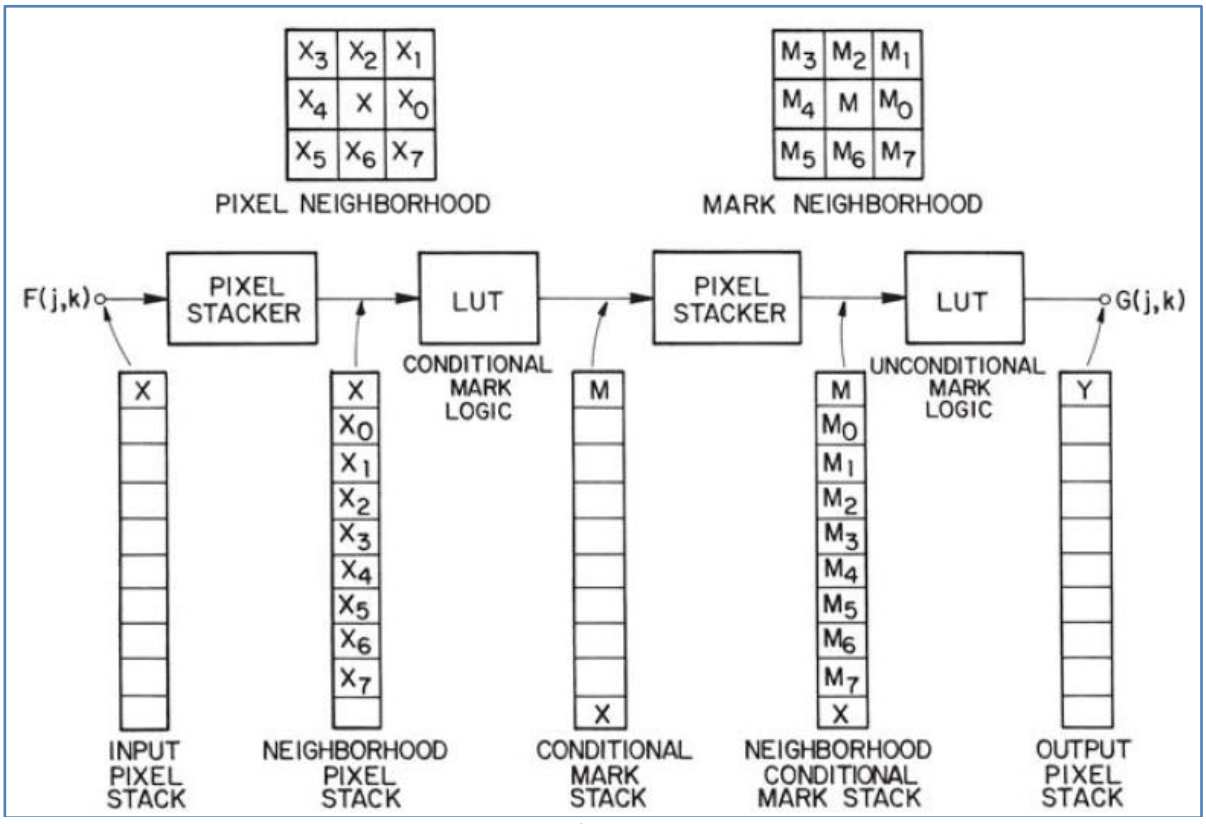


Figure 6: Flowchart of a Morphological Process

Stage 1 – This involves marking of pixels for erasure. For comparison, we use the patters below:

Shrink, Thin and Skeletonize Conditional Mark Patterns [$M = 1$ if hit]

Table	Bond	Pattern
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 / 0 0 1 ; 0 0 0 / 0 1 0 / 1 0 0
S	2	0 0 0 / 0 1 0 / 0 1 1 ; 0 0 0 / 0 1 0 / 1 1 0 ; 0 1 0 / 0 1 0 / 0 0 0 ; 0 1 0 / 0 1 0 / 0 0 0
S	3	0 0 1 / 0 1 1 / 0 1 1 ; 1 1 0 / 0 1 0 / 0 1 0 ; 1 0 0 / 1 1 0 / 0 1 1 ; 0 0 0 / 0 1 0 / 0 1 1 ; 0 0 0 / 0 1 0 / 1 1 0 ; 0 0 0 / 0 1 0 / 0 1 1 ; 0 0 0 / 0 1 0 / 1 0 0 ; 0 1 1 / 0 1 0 / 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 / 0 1 1 / 0 1 0 ; 0 0 0 / 1 1 0 / 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 / 0 1 1 / 0 0 0 ; 1 1 0 / 1 1 0 / 0 0 0 ; 0 1 1 / 1 1 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 / 0 0 1 ; 1 1 1 / 1 1 0 / 0 0 0 ; 0 1 1 / 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 1 / 0 1 1 / 1 1 1 ; 0 1 1 / 0 1 1 / 0 1 1
STK	7	1 1 1 / 1 1 1 / 1 0 0 ; 0 0 1 ; 0 1 1 / 1 1 0 / 1 1 0 ; 0 1 1 / 0 0 1 / 1 0 0 ; 1 1 1 / 1 1 1
STK	8	0 1 1 / 1 1 1 / 1 1 0 ; 0 0 0 ; 0 1 1 / 1 1 1 / 1 1 0 ; 1 1 1 / 0 1 1 / 0 0 0 ; 1 1 0 / 1 1 1
STK	9	1 1 1 / 0 1 1 / 1 1 1 ; 1 1 1 ; 1 1 0 ; 1 0 0 ; 0 0 1 ; 0 1 1 / 0 1 1 / 1 1 1 ; 1 1 1 ; 1 1 0 ; 1 1 0 ; 1 1 1 ; 1 1 1 ; 0 1 1 / 1 1 1 / 1 0 0 ; 0 0 1 ; 1 1 0 ; 1 1 1 ; 1 1 1 ; 1 1 1
STK	10	1 1 1 / 1 1 1 / 1 1 1 ; 1 0 1 ; 0 1 1 / 1 1 1 / 1 1 0 ; 1 1 1 ; 1 1 1 / 1 0 1 / 1 1 1 ; 1 1 1
K	11	1 1 1 / 1 1 1 / 1 1 0 ; 0 1 1 ; 1 1 1 / 1 1 1 / 1 1 1 ; 1 1 1 ; 0 1 1 / 1 1 0 / 1 1 1 ; 1 1 1

Figure 7: Patterns according to bond values

Stage 2 – In stage 1, the pixel marked for erasure are subject to rechecking to avoid total erasure of the image. A second stage is required, because how much ever controlled the process is, total erasure is cannot be avoided in any case. The second stage ensures a check that connectivity is not broken. The second stage is common for all reduction operation, i.e. shrinking, thinning and skeletonizing. For comparison, we use the patters below:

Shrink and Thin Unconditional Mark Patterns — **[$P(M, M_0, M_1, M_2, M_3, M_4, M_5, M_6, M_7) = 1$ if hit][a]**

Pattern	Pattern
Spur: 0 0 *M* / *M* 0 0 / 0 *M* 0 ; 0 0 0 / 0 *M* 0 / 0 *M* 0 ; 0 0 0 / 0 *M* 0 / 0 0 *M* ; 0 0 0 / 0 *M* 0 / 0 0 0 Single 4-connection: 0 0 0 / 0 *M* 0 / 0 0 0 ; 0 0 0 / 0 *M* 0 / 0 *M* 0 ; 0 0 0 / 0 *M* 0 / 0 *M M* ; 0 0 0 / 0 *M* 0 / 0 0 0	Corner cluster: *M M D* / *M M D* / *D D D*
L Cluster: 0 0 *M* / 0 *M M* / 0 0 0 ; 0 0 *M* / *M M* 0 / 0 *M* 0 ; *M M* 0 / 0 *M M* / 0 0 0 ; *M* 0 0 / *M M* 0 / 0 0 0 ; 0 0 0 / *M M* 0 / 0 *M* 0 ; 0 0 0 / 0 *M M* / 0 0 0 ; 0 0 0 / 0 *M M* / 0 *M* 0 ; 0 0 0 / 0 *M M* / 0 0 *M*	Tee branch: *D M* 0 / 0 *M D* / 0 0 0 ; 0 0 *D* / 0 *M D* / 0 0 *D* ; *D* 0 0 / *D M* 0 / *D* 0 0 ; *D M D* / *M M M* / 0 0 0 ; 0 *M* 0 / *M M* 0 / 0 0 0 ; 0 *M* 0 / *M M* 0 / 0 *M* 0 ; *D M D* / *D M D* / 0 0 0 ; *D M D* / *D M D* / 0 *M* 0
4-Connected offset: 0 *M M* / *M M* 0 / 0 0 0 ; 0 *M* 0 / *M M* 0 / 0 0 *M* ; 0 0 *M* / 0 *M M* / 0 *M* 0 ; 0 *M M* / *M M* 0 / 0 0 0	Vee branch: *M D M* / *D M D* / *A B C* ; *M D C* / *D M B* / *M D A* ; *C B A* / *D M D* / *M D M* ; *A D M* / *B M D* / *C D M*
Spur corner cluster: 0 *A M* / *M B* 0 / 0 0 0 ; 0 0 *M* / *A M* 0 / *M* 0 0 ; 0 *M B* / *A M* 0 / 0 0 0 ; *M* 0 0 / 0 *M B* / 0 *A M*	Diagonal branch: *D M* 0 / 0 *M D* / *M* 0 0 ; *D* 0 *M* / *M M* 0 / 0 *M D* ; 0 *M M* / *M M* 0 / *M* 0 *D* ; *M* 0 *D* / *D* 0 *M* / 0 *M D*

[a] $A \cup B \cup C = 1 \quad D = 0 \cup 1 \quad A \cup B = 1$.

Figure 8: Patterns according to bond values

Stage 1 and Stage 2 are performed repeatedly till a steady state is reached, that is, there is no change in the results. Below is an example that was solved for giving an insight into the process:

Input X							Stage 1 M							Mbar							Stage 2 Phit						
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0	0	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0

from Stage 1 W = x AND Mbar							from Stage2 Q= x AND Phit							Output Y= W OR Q						
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	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	0	1	1	1	1	0	0	0	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 9: Solved Example – Thinning a 2x5 block

Input X							Stage 1 M							Mbar							Stage 2 Phit						
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0	1	0	0	0	1	0	1	0	1	1	1	0	1	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0	1	1	1	1	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0

from Stage 1 W = x AND Mbar							from Stage2 Q= x AND Phit							Output Y= W OR Q						
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	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	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	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 10: Solved Example - 3x5 block

Part A: Shrinking – Count Stars

Approach and Procedures

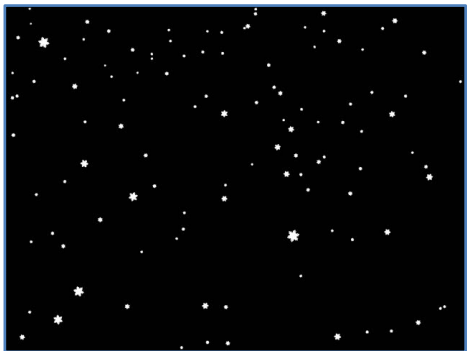


Figure 11: Counting stars in Stars.raw

Step 1: Binarize and Clean the image

1. The image contains some gray values between 0 and 255. Make them 0 or 255 by applying a threshold of 127. This threshold is necessary since the shrinking operation is meant for a binary image.
2. Count the number of singleton pixels, i.e. pixels of size 1.
3. After counting a singleton pixel, erase it from the image.

Step 2: Apply Morphological filter – Stage 1 of Shrinking

1. Pad the image at the boundaries – pre-process for a 3x3 kernel
2. Create M matrix – mark the pixels to be erased
3. Calculate $\bar{M}$ = NOT(M) following
4. Calculate intermediate matrix W = X AND $\bar{M}$

Step 3: Apply Morphological filter – Stage 2

1. Pad the image at the boundaries – pre-process for a 3x3 kernel
2. Create P_{hit} matrix from M matrix in step 2 – unmark the pixels marked for erasure
3. Calculate intermediate matrix Q = X AND P_{hit}

Step 4: Calculate the result of this iteration

1. Create result matrix Y = W OR Q
2. Find the number of stars reduced in this iteration. This can be found by counting the number of singleton pixels, i.e. pixels of size 1. Save the result in a variable, preferably a stack (C) or a vector(C++).
3. Add the result to the previous count to get the total number of stars.
4. After counting a singleton pixel, erase it from the image.

Step 5: Check against the original image

1. Check the difference between Y and X, i.e. the input and output images of this iteration
2. Repeat steps 2-5 till Y and X become the same, i.e. no difference between successive iterations

Results and Discussion

Star Size	3	4	6	7	9	11	12	14	15	17	18	20	Total
Count of stars	10	47	20	11	7	8	2	3	1	1	1	1	112

Table 1: Number of Stars and their sizes

There are total 112 stars in the image. It took 13 iterations to reach the convergence. Star size is defined as the maximum L2 distance between two pixels x and y belonging to the same star. But interestingly, there is an empirical relation that was discovered while calculating the size, i.e. given by:

Star Size = Floor value of (1.5833 x iteration)

This was found manually, but even programmatically it was found to be true. Because, it takes 2 iterations to shrink a 3x3 size block to a single pixel and Floor (2 x 1.5833) = Floor (3.17) = 3. The same is true for a star of size 20 also. Since it takes 13 iterations to reduce a 20x20 block to a single pixel and Floor (13 x 1.5833) = Floor (20.58) = 20.

```
root@syalanurag1991-vb:/media/sf_Shared/HW2/Code/C++/bin# ./ObjectsInImage 1 ../../../stars.raw ../../../stars_shrink.raw

The size of the file is: 307200bytes
Enter image size as <height> <width> <1-grayscale/3-color>, e.g. 100 50 3
640 480 1

Foreground: White

Converting to Binary ...
```

```
Counting object - Iteration #1

Stats:
Stars counted in this iteration: 0
Star size found for iteration  : 0
   Stage 1
   Stage 2


Counting object - Iteration #2

Stats:
Stars counted in this iteration: 10
Star size found for iteration  : 3
   Stage 1
   Stage 2


Counting object - Iteration #3

Stats:
Stars counted in this iteration: 47
Star size found for iteration  : 4
   Stage 1
   Stage 2


Counting object - Iteration #4

Stats:
Stars counted in this iteration: 20
Star size found for iteration  : 6
   Stage 1
   Stage 2
```

```
Counting object - Iteration #5

Stats:
Stars counted in this iteration: 11
Star size found for iteration  : 7
   Stage 1
   Stage 2


Counting object - Iteration #6

Stats:
Stars counted in this iteration: 7
Star size found for iteration  : 9
   Stage 1
   Stage 2


Counting object - Iteration #7

Stats:
Stars counted in this iteration: 8
Star size found for iteration  : 11
   Stage 1
   Stage 2


Counting object - Iteration #8

Stats:
Stars counted in this iteration: 2
Star size found for iteration  : 12
   Stage 1
   Stage 2


Counting object - Iteration #9

Stats:
Stars counted in this iteration: 3
Star size found for iteration  : 14
   Stage 1
   Stage 2
```

```
Counting object - Iteration #10

Stats:
Stars counted in this iteration: 1
Star size found for iteration  : 15
   Stage 1
   Stage 2


Counting object - Iteration #11

Stats:
Stars counted in this iteration: 1
Star size found for iteration  : 17
   Stage 1
   Stage 2


Counting object - Iteration #12

Stats:
Stars counted in this iteration: 1
Star size found for iteration  : 18
   Stage 1
   Stage 2


Counting object - Iteration #13

Stats:
Stars counted in this iteration: 1
Star size found for iteration  : 20
   Stage 1
   Stage 2

Total number of stars    : 112
Number of different sizes: 12
Distribution of sizes    :
   3   4   6   7   9  11  12  14  15  17  18  20
  10  47  20  11   7   8   2   3   1   1   1   1

Success!!
```

Figure 12: Console results for this problem

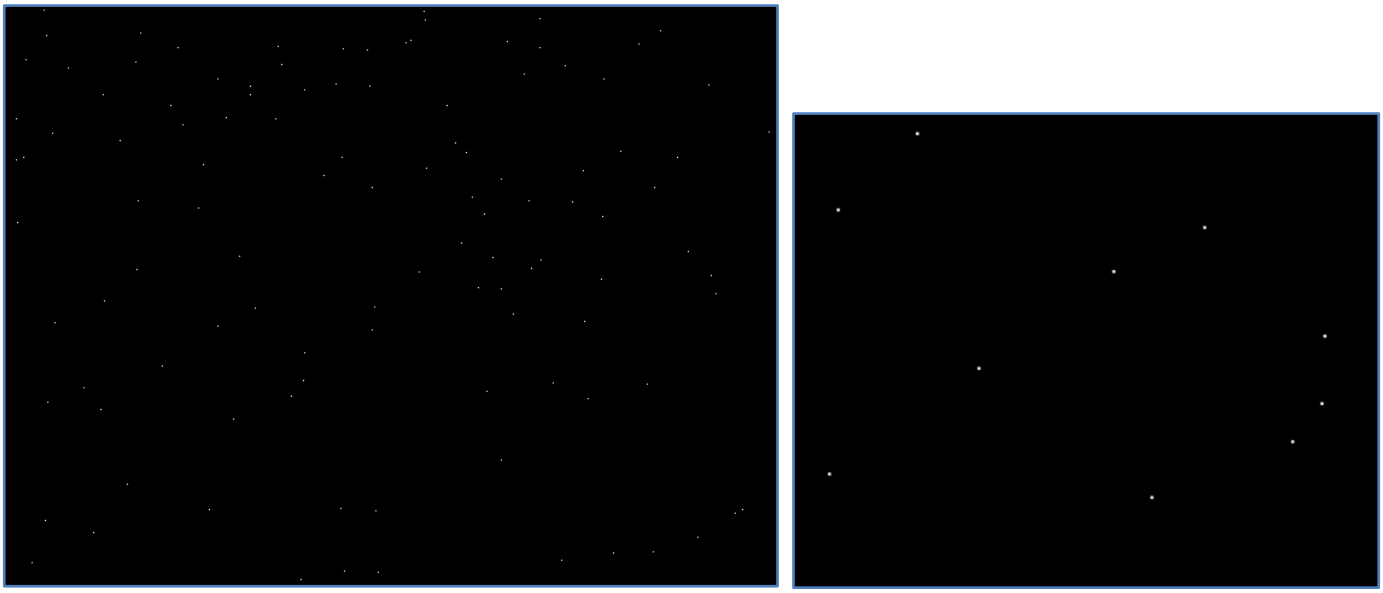


Figure 13: a) Final image after 13 iterations of shrinking b) Zoom over a portion of the final image

Part B: Thinning

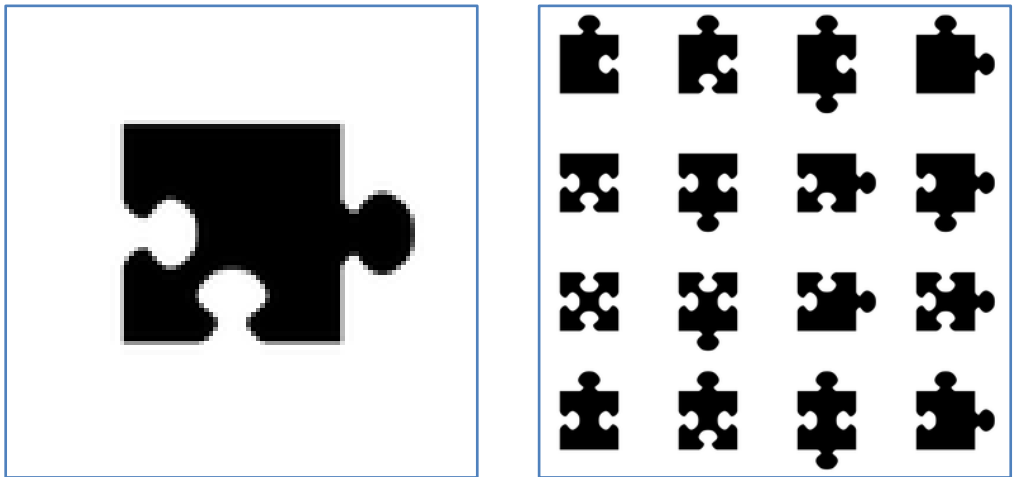


Figure 14: a) Thinning jigsaw_1.raw b) board.raw

Step 1: Binarize and Clean the image

The image contains some gray values between 0 and 255. Make them 0 or 255 by applying a threshold of 127. This threshold is necessary since the shrinking operation is meant for a binary image.

Note: I have applied a threshold of 50 here. I will explain the reason later.

Step 2: Apply Morphological filter – Stage 1 of Thinning

1. Pad the image at the boundaries – pre-process for a 3x3 kernel and traverse over the image
2. Create M matrix – mark the pixels to be erased
3. Calculate $\bar{M}$ = NOT(M) following
4. Calculate intermediate matrix W = X AND $\bar{M}$

Step 3: Apply Morphological filter – Stage 2

1. Pad the image at the boundaries – pre-process for a 3x3 kernel
2. Create P_{hit} matrix from M matrix in step 2 – unmark the pixels marked for erasure
3. Calculate intermediate matrix Q = X AND P_{hit}

Step 4: Calculate the result of this iteration

1. Create result matrix Y = W OR Q
2. Find the number of stars reduced in this iteration. This can be found by counting the number of singleton pixels, i.e. pixels of size 1. Save the result in a variable, preferably a stack (C) or a vector(C++).

Step 5: Check against the original image

1. Check the difference between Y and X, i.e. the input and output images of this iteration
2. Repeat steps 2-5 till Y and X become the same, i.e. no difference between successive iterations

Results and Discussion

It took 21 iterations to get the final thinned image, when the threshold was 50.

VERY IMPORTANTY NOTE: If the threshold for binarizing the image is kept 127, then the operation breaks the connectivity of the image. This means then that some of the pixels in the middle of the image are accidently changed to 0 (white background) which leads to undesirable results.

Threshold	Number of Iterations
50	21
100	20
127	20
150	24
200	22

Table 2: Comparison Threshold vs Iterations for jigsaw_1.raw

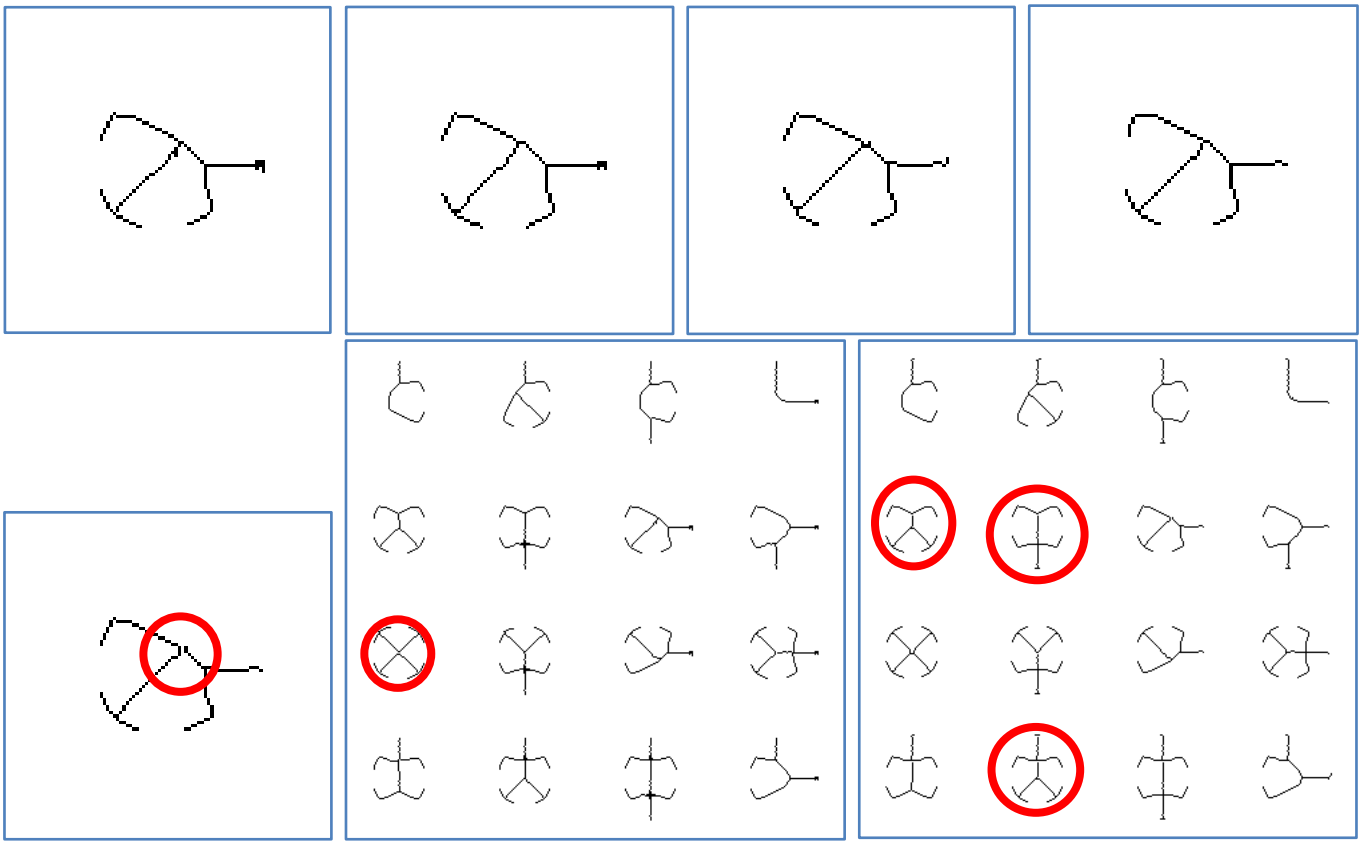


Figure 15: Final images for jigsaw_1.raw and board.raw for different levels of thresholds, breaks are circled in red
50, 100, 150, 200 and 127 for jigsaw_1.raw
0 and 127 for board.raw

Input						
			X			
0	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	1	1	0	0
0	0	1	1	1	0	0
0	0	1	0	1	1	0
0	0	0	0	0	0	0

Stage 1						
			M			
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	1	1	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0

			Mbar			
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	0	0	1	1
1	1	1	0	1	1	1
1	1	1	1	0	1	1
1	1	1	1	1	1	1

Stage 2						
			Phit			
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	0	0

	from Stage 1								from Stage2								Output						
W	=	x AND Mbar							Q=	x AND Phit							Y=	W OR Q						
0	0	0	0	0	0	0			0	0	0	0	0	0	0		0	0	0	0	0	0	0
0	1	0	0	0	0	0			0	0	0	0	0	0	0		0	1	0	0	0	0	0
0	0	1	0	0	0	0			0	0	0	0	1	0	0		0	0	1	0	1	0	0
0	0	1	0	1	0	0			0	0	0	0	0	0	0		0	0	1	0	1	0	0
0	0	1	0	0	1	0			0	0	0	0	1	0	0		0	0	1	0	1	1	0
0	0	0	0	0	0	0			0	0	0	0	0	0	0		0	0	0	0	0	0	0

Figure 16: Solved example where the image breaks while thinning

Part C: Skeletonizing

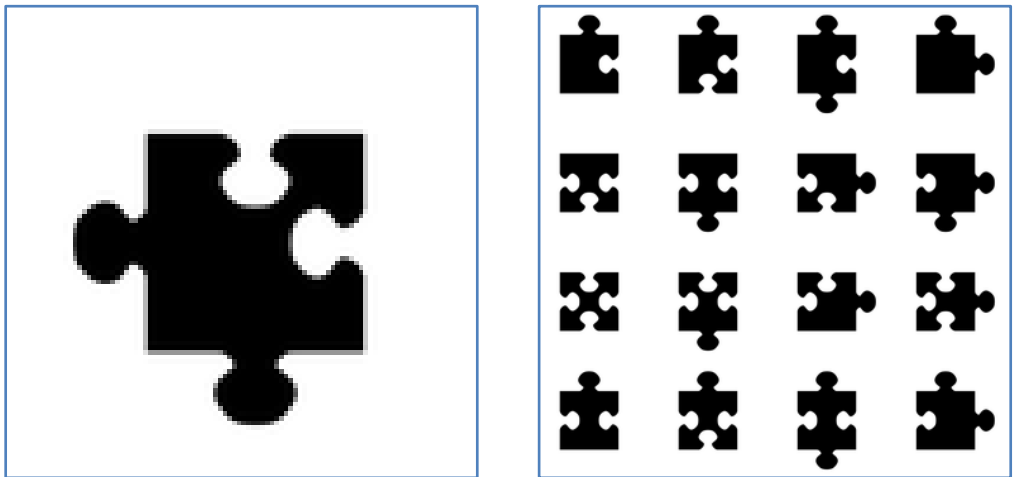


Figure 17: a) Thinning jigsaw_2.raw b) board.raw

Step 1: Binarize and Clean the image

The image contains some gray values between 0 and 255. Make them 0 or 255 by applying a threshold of 127. This threshold is necessary since the shrinking operation is meant for a binary image.

Note: I have applied a threshold of 50 here. I will explain the reason later.

Step 2: Apply Morphological filter – Stage 1 of Skeletonizing

1. Pad the image at the boundaries – pre-process for a 3x3 kernel and traverse over the image
2. Create M matrix – mark the pixels to be erased
3. Calculate $\bar{M}$ = NOT(M) following
4. Calculate intermediate matrix W = X AND $\bar{M}$

Step 3: Apply Morphological filter – Stage 2

1. Pad the image at the boundaries – pre-process for a 3x3 kernel
2. Create P_{hit} matrix from M matrix in step 2 – unmark the pixels marked for erasure
3. Calculate intermediate matrix Q = X AND P_{hit}

Step 4: Calculate the result of this iteration

1. Create result matrix Y = W OR Q

2. Find the number of stars reduced in this iteration. This can be found by counting the number of singleton pixels, i.e. pixels of size 1. Save the result in a variable, preferably a stack (C) or a vector(C++).

Step 5: Check against the original image

1. Check the difference between Y and X, i.e. the input and output images of this iteration
2. Repeat steps 2-5 till Y and X become the same, i.e. no difference between successive iterations

Results and Discussion

It took 15 iterations to get the final skeletonized image, when the threshold was 50.

VERY IMPORTANTY NOTE: If the threshold for binarizing the image is kept above 50, then the operation breaks the connectivity of the image. This means then that some of the pixels in the middle of the image are accidently changed to 0 (white background) which leads to undesirable results.

Threshold	Number of Iterations
50	15
100	15
127	16
150	22
200	16

Table 3: Comparison Threshold vs Iterations for jigsaw_1.raw

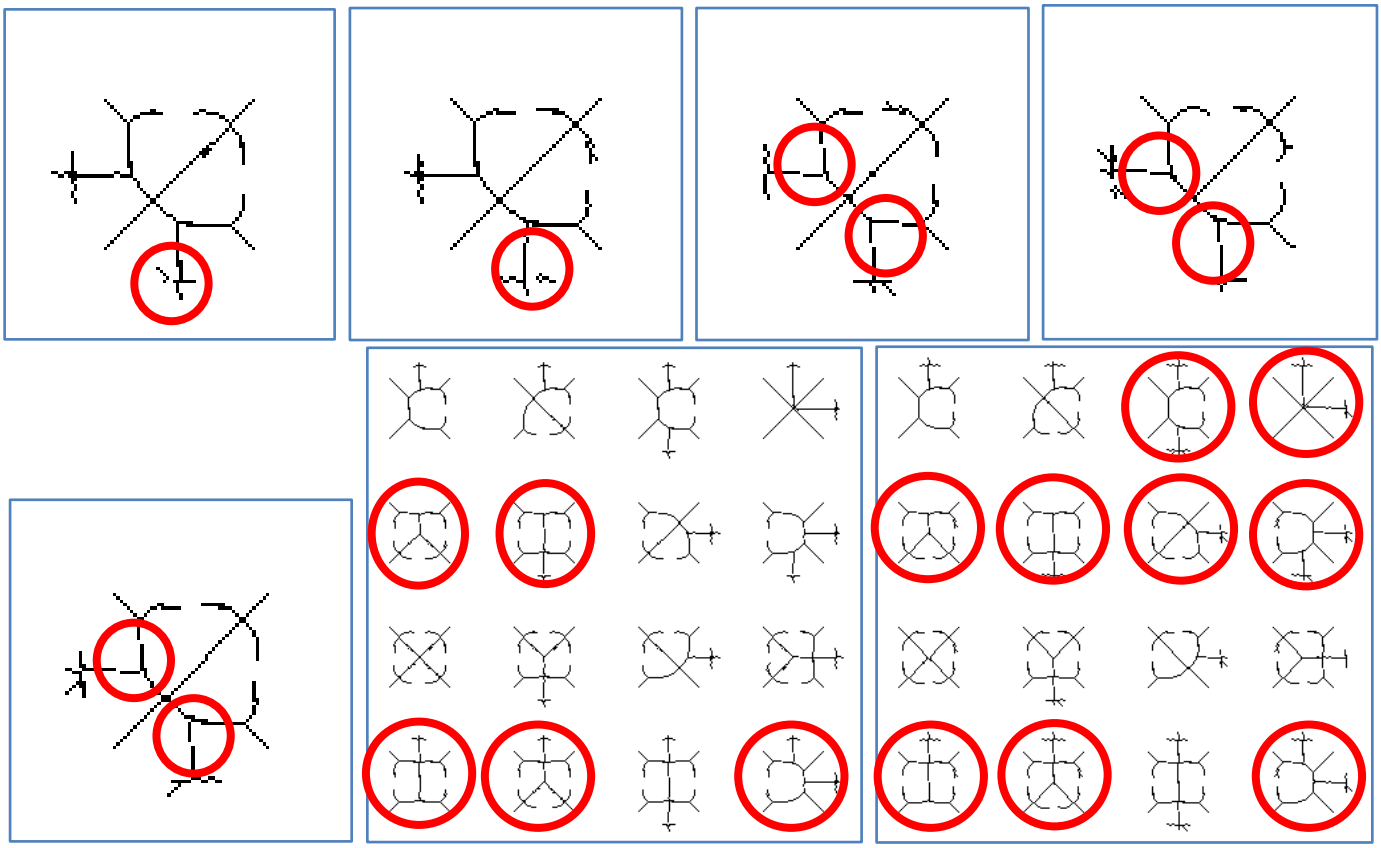


Figure 18: Final images for jigsaw_1.raw and board.raw for different levels of thresholds, breaks are circled in red
50, 100, 150, 200 and 127 for jigsaw_1.raw
0 and 127 for board.raw

Conclusion

Shrinking, thinning and skeletonizing are controlled erosion processes. The process of shrinking was used to find the total number of starts in the star.raw image. The processes break connectivity in images having thin necks.

Part D: Counting Game – Puzzle Board

Abstract and Motivation

The task was to find the total number of puzzle pieces in the image board.raw. Also, the number of unique pieces in the image is to be found. One can see that by visual inspection, there are 10 unique pieces in the image.

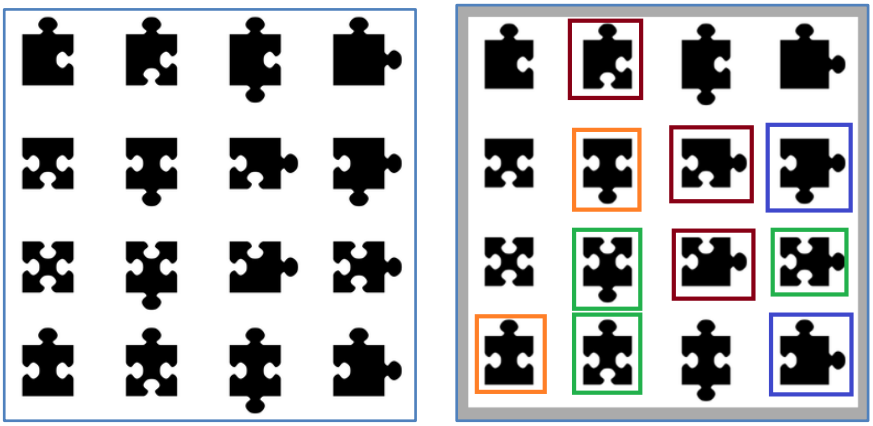


Figure 19: Subject image board.raw and the number of unique pieces, Repetitions are marked in similar colors

VERY IMPORTANT NOTE: *One can think that morphological operation like Shrinking can be used to count the number of pixels. But NO, shrinking cannot be used, because some of the puzzles pieces shrank to 2 dots.*

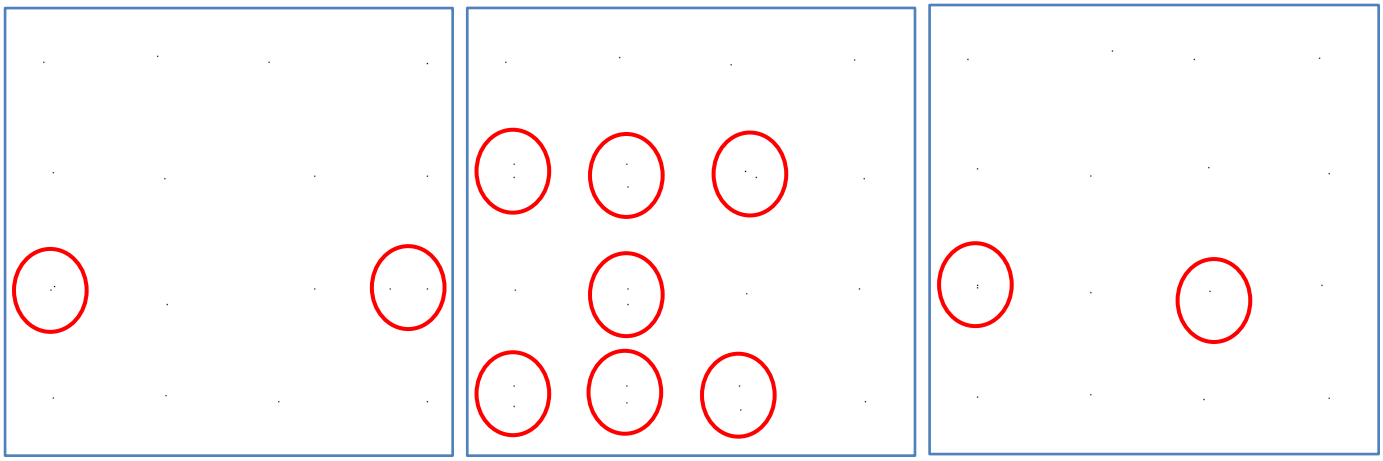


Figure 20: Shrunk images for board.raw at threshold 50, 127 and 200

The above images where shrunk after thresholding them at different levels. But there are always some artefacts that make at least 1 or 2 pieces shrink to more than 1 dots.

Approach and Procedures (Intermediate Results included)

Since shrinking is not a good approach. A different algorithm was devised. It involves the concept of **morphological thickening**. The algorithm is detailed in the next section. Following is a summary:

1. Morphological thickening to form bounding rectangle
2. Create a label matrix by traversing over thickened images containing rectangles
3. Use label matrix to extract individual objects
4. Compare each object with every object in the image at all possible orientations to find unique objects

1. Process for Morphological Thickening to help form label matrix

This process helps to expand separated objects in a binary image. The aim was to find the bounding box that encompasses that object. To do this, customized filters where used as shown below. These filters morphologically transform an irregularly shaped object to become a rectangle.

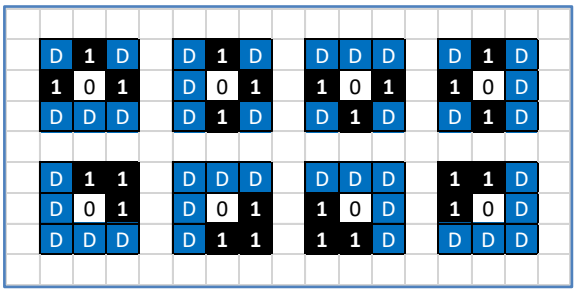


Figure 21: Customized filters to filling objects to become rectangle, D = Don't care

To demonstrate how this work, an example has been solved here:

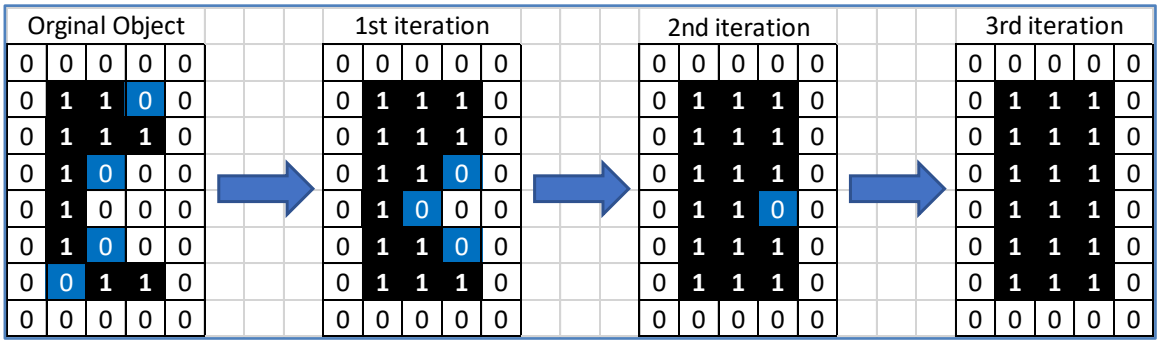


Figure 22: Example showing expansion of an object to form a rectangle, blue pixels will be changed to 1

Following steps were used in the algorithm for hole-filling.

Step 1: Binarize and Clean the image

The image contains some gray values between 0 and 255. Binarize them (0 or 255) by applying a threshold of 50.

Step 2: Apply Hit-and-Miss transform

1. Pad the image at the boundaries – pre-process for a 3x3 kernel and traverse over the image
2. If the neighborhood matches with the filters shown above, change them from 0 to 1

Step 3: Check against the original image

1. Check the difference between Y and X, i.e. the input and output images of this iteration
2. Repeat steps 2 till Y and X become the same, i.e. no difference between successive iterations

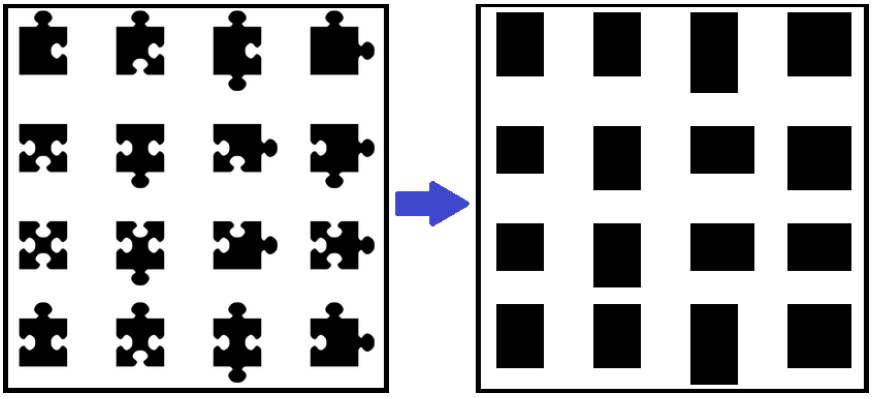


Figure 23: Result of thickening procedure explained above

2. Process for Labeling

This process helps to mark different rectangles in an image with a different label. Since we already know that the objects are present in a bounding box, we can mark them. To do this, customized filters where used as shown below. These filters help in finding out **the starting of a rectangle** and **the area inside a rectangle**.

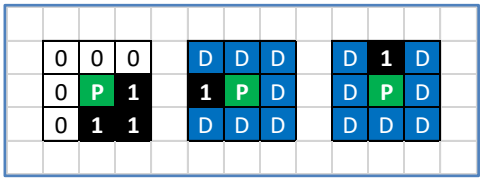


Figure 24: Customized filters to find a connected rectangle and label it, D = Don't care, P = Pixel being examined a) Filter to find top-left corner of a rectangle b) Check left neighbor c) Check upper neighbor

To demonstrate how this work, an example has been solved here:

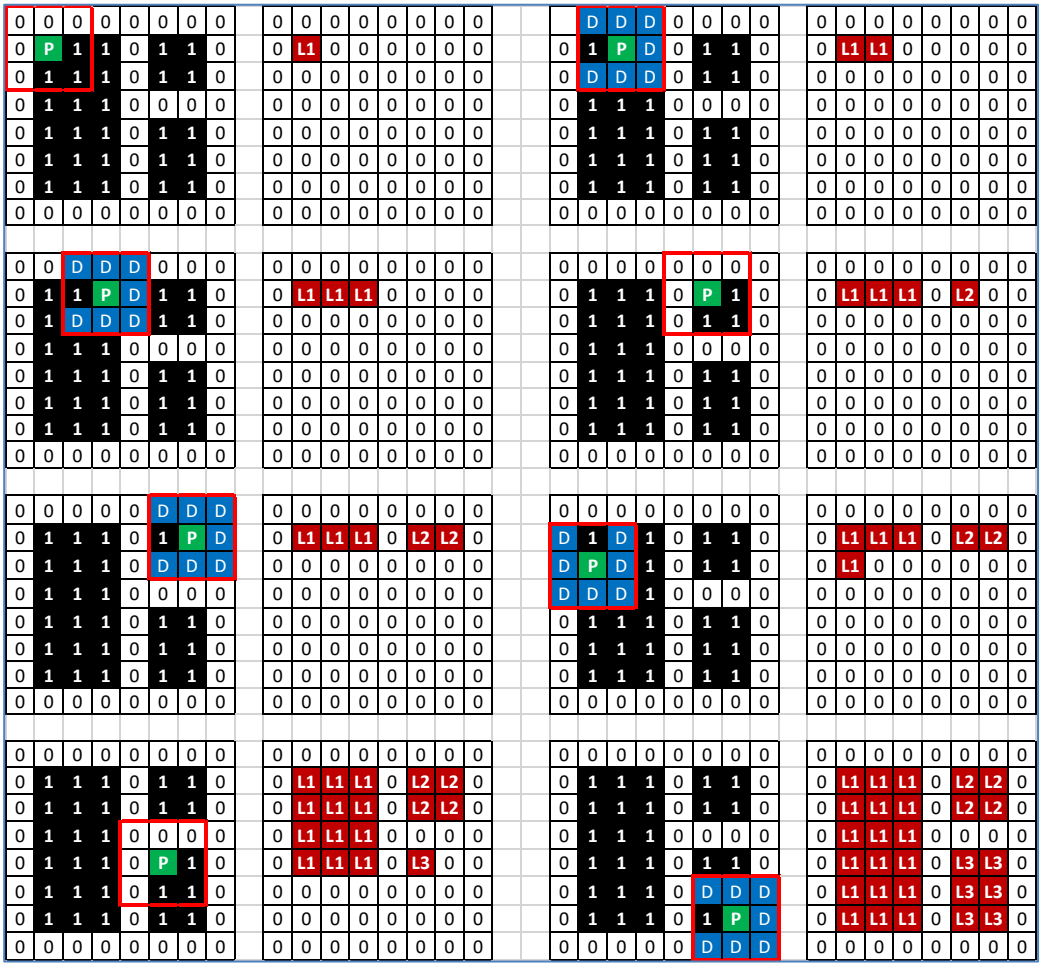


Figure 25: A solved example to show the approach used for labelling

Following steps were used in the algorithm for labelling:

Step 1: Apply Hit-and-Miss transform

1. Pad the image at the boundaries – pre-process for a 3x3 kernel and traverse over the image
2. If neighborhood of a pixel matches with the top-left corner filters as shown above, that means we have encountered the first pixel of the rectangle
3. **Create** a new label and assign it to corresponding position of the image in a new **label-matrix** (array)

Step 2: Assign labels by looking at the nearest neighbor

IF left-neighbor exists -> Assign the same label as of left-pixel to the current pixel

ELSE IF upper-neighbor exists -> Assign the same label as of upper-pixel to the current pixel

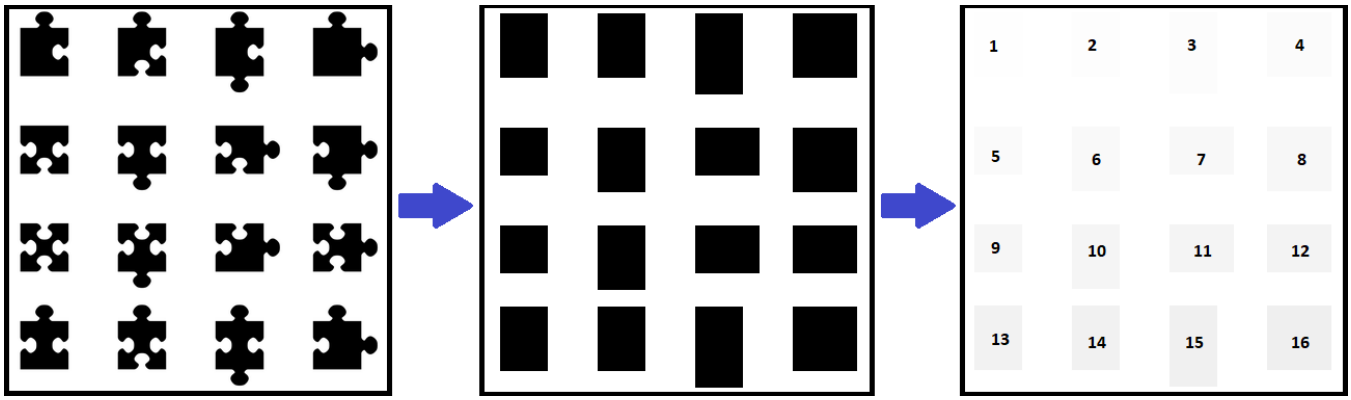


Figure 26: Result of labelling. The labels are gray levels from 1-16. For easy view, they have been purposefully marked

3. *Process for Extracting objects using the label-matrix*

This process helps to extract objects from the original image using the label-matrix created in the previous process. The following code, embodies the algorithm. This is an implementation in C++. The basic idea is to use stacks (C) or vectors (C++) to store the intermediary data to form images of individual objects.

```
vector<vector<vector<vector<unsigned char>>>> ExtractObjectsFromImage (vector<vector<vector<unsigned char>>> ImageVector_bin,
                                                                       type_labelData labelInformation){

    int height = ImageVector_bin.size();
    int width = ImageVector_bin.front().size();
    int channels = ImageVector_bin.front().front().size();

    vector<vector<vector<unsigned char>>> labelVector = labelInformation.labelVector;
    int label = labelInformation.label;

    vector<vector<vector<unsigned char>>> tempImageVector;
    vector<vector<vector<vector<unsigned char>>>> CollectionOfObjects (label, tempImageVector);

    for(int i=0; i<height; i++){
        int lastPositiveLabel = 0;
        vector<vector<unsigned char>> newRow;
        for(int j=0; j<width; j++){
            int currentLabel = 0;
            vector<unsigned char> newColumn;
            for(int k=0; k<channels; k++){
                currentLabel = labelVector[i][j][k];
                if(currentLabel>0){
                    lastPositiveLabel = labelVector[i][j][k];
                    newColumn.push_back(ImageVector_bin[i][j][k]);
                }
            }

            if(currentLabel>0){
                newRow.push_back(newColumn);
            }

            if((newRow.size()>0) && (currentLabel==0)) {
                CollectionOfObjects.at(lastPositiveLabel-1).push_back(newRow);
                lastPositiveLabel = 0;
                newRow.clear();
            }
        }
    }

    return CollectionOfObjects;
}
```

Figure 27: Code showing the algorithm for extraction of objects

Following steps were used in the algorithm for labelling:

Step 1: Create a vector of image vectors Since we already know the number of labels distributed

1. Number of objects = Number of labels distributed = Max(labels)
   Since we already know the number of labels distributed, we know the number total of objects.
2. Create a stack (C) or a vector (C++) of image vector having capacity = Number of objects

Step 2: Traverse over the label matrix but fetch data from the original image at the corresponding pixel position

1. Search for the first non-zero label, this indicates start of a rectangle
   a. Push all the pixels from different channels to form a column vector (here number of channels = 1)
   b. Subsequently push column vector to a row vector
2. If a 0 is encountered, means we have reached the end of a bounding rectangle
   If the current row size>0
   i. Push the row vector to an image object pertaining to that label
   ii. Empty the row vector, because there may be more non-zero labels in the row

This algorithm extracts the objects from the original image and stores them in a vector of image vectors.

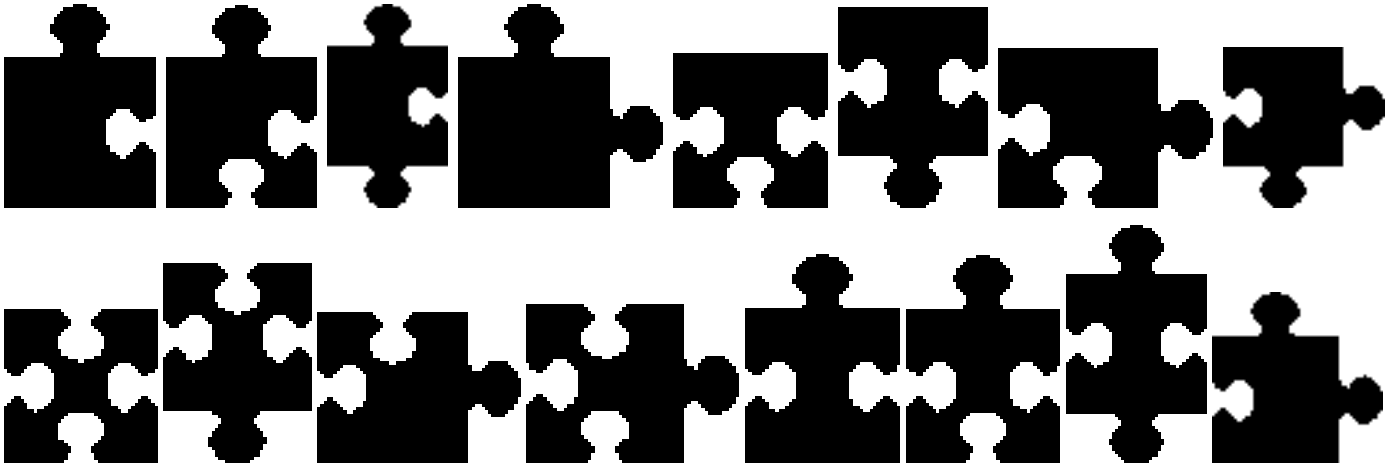


Figure 28: All the objects in the image are extracted and kept in a vector of image vector

4. Process for Finding the Unique Objects

Since the puzzle pieces can be in any orientation possible, we must check for all possible cases created by rotating the individual images in multiples of 90°, horizontal flipping and vertical flipping. In total there are:

4 rotations - > 0°, 90°, 180° and 270°

X 5 possible cases for each orientation -> No change, H-flip, V-flip, V-flip of H-flip and H-flip of V-flip

= 20 possible cases

But most of the cases are redundant. There are only 8 cases we need to check for:

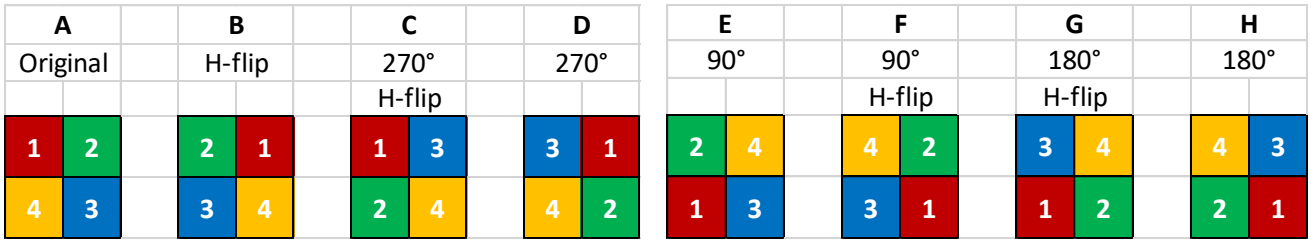


Figure 29: Possible orientations to be checked

Following steps were used in the algorithm for rotating and comparing the images with each other.

Step 1: Check orientation A (original orientation), Total Count = Max(labels)

1. Check with every other image in the stack
2. Use a function that calculates the difference between two images
   NOTE: Since the images have been modified from their original look, the match may not be 100% even if it is the same object, a threshold must be kept. In this problem, if similarity 98% or above is called a match
3. If a match is found
    a. Reduce the total count by 1
    b. Display the label of the object it matches to
    c. Break the loop

Step 2: Check orientation B

1. Flip the subject image horizontally
2. Check with every other image in the stack
3. Use a function that calculates the difference between two images
4. Similarity % = Max (Similarity % of current step, Similarity % of previous step)
5. If a match is found
    a. Reduce the total count by 1
    b. Display the label of the object it matches to
    c. Break the loop

Step 3: Check orientation C

1. Rotate the image from the previous step by 90°
2. Repeat steps 2.2-2.5

Step 4: Check orientation D

1. Flip the image from the previous step horizontally
2. Repeat steps 2.2-2.5

Step 5: Check orientation E

1. Rotate the image from the previous step by 180°
2. Repeat steps 2.2-2.5

Step 6: Check orientation F

1. Flip the image from the previous step horizontally
2. Repeat steps 2.2-2.5

Step 7: Check orientation G

1. Rotate the image from the previous step by 270°
2. Repeat steps 2.2-2.5

Step 8: Check orientation H

1. Flip the image from the previous step horizontally
2. Repeat steps 2.2-2.5

For optimization, an object is checked only with the object remaining in the list, i.e. the 1st object is compared with all the 15-other object, but the 2nd object is only compared with remaining 14, 3rd with remining 13 and so on. This is done because:
If the 2nd object matches with the 7th object, then the reverse is also possible, so no need to check again.

Results and Discussion

There are total 16 puzzle pieces and 10 unique puzzle pieces.
Puzzle pieces 2, 7 and 11 are the same
Puzzle pieces 6 and 13 are the same
Puzzle pieces 8 and 16 are the same
Puzzle pieces 10, 12 and 14 are the same

```
Filling gaps to nearest rectangle - Iteration #45
Filling gaps to nearest rectangle - Iteration #46
Filling gaps to nearest rectangle - Iteration #47
Filling gaps to nearest rectangle - Iteration #48
Filling gaps to nearest rectangle - Iteration #49

Labeling objects in the image ...

Extracting objects in the image ...

Object #1
Printing to file
Width: 46 Height: 62
Matches for this object:

   No match


Object #2
Printing to file
Width: 46 Height: 62
Matches for this object:

   Object #7 99.3%
   Object #11 99.23%


Object #3
Printing to file
Width: 46 Height: 78
Matches for this object:

   No match


Object #4
Printing to file
Width: 62 Height: 62
Matches for this object:

   No match


Object #5
Printing to file
Width: 46 Height: 46
Matches for this object:

   No match


Object #6
Printing to file
Width: 46 Height: 62
Matches for this object:

   Object #13 98.98%


Object #7
Printing to file
Width: 62 Height: 46
Matches for this object:

   Object #11 98.95%


Object #8
Printing to file
Width: 62 Height: 62
Matches for this object:

   Object #16 99.17%


Object #9
Printing to file
Width: 46 Height: 46
Matches for this object:

   No match


Object #10
Printing to file
Width: 46 Height: 62
Matches for this object:

   Object #12 99.02%
   Object #14 98.63%


Object #11
Printing to file
Width: 62 Height: 46
Matches for this object:

   No match


Object #12
Printing to file
Width: 62 Height: 46
Matches for this object:

   Object #14 98.91%


Object #13
Printing to file
Width: 46 Height: 62
Matches for this object:

   No match


Object #14
Printing to file
Width: 46 Height: 62
Matches for this object:

   No match


Object #15
Printing to file
Width: 46 Height: 78
Matches for this object:

   No match


Object #16
Printing to file
Width: 62 Height: 62
Matches for this object:

   No match


Total number of objects : 16
Total unique objects    : 10


Success!!
```

Figure 30: Console Outputs

Conclusion

The result was found to be in line with the visual inspection. Morphological thickening along with labelling is a good algorithm to extract objects within a single image. **A separation of AT LEAST 1 pixel is necessary for this to work.**