EE 569: Digital Image Processing

Homework #3

Faiyadh Shahid
Student ID: 4054-4699-70
fshahid@usc.edu

November 1, 2015

Table of Contents

1 Problem 1

1.1 Abstract and Motivation

Geometric and perspective transformation are widely used in computer graphics and computer vision. Usage of basic operations such as translation, rotation, scaling is not only limited to creating special effects; but also in real-world issues. The idea of implementing transformation is mainly achieved by moving from one coordinate system to another (e.g. Cartesian to polar) or finding a transformation matrix which can provide us with the desired outcome.

In this problem, we will look onto creating a special effect called swirl effect, whose principle intuition is derived from the rotational transformation around the center of the image. We will also look into capturing a scene in the 3D world coordinate system and projecting it onto a 2D image plane. The final output will create an illusion such that images are placed on the faces of a cube.

1.2 Approach and Procedure

1.2.1 Swirl Effect

The implementation of swirl effect is made possible by mapping various points on the Cartesian coordinate to a polar coordinate system in which we solve for radius and angle based on the x and y-positions of the image. The common formulation used are as follows:

$$r = \sqrt{x^2 + y^2}$$
$$\theta = \tan^{-1}\frac{y}{x}$$

where x and y represents rows and column-wise positions of the image respectively, and r & θ are the radius and angle parameters in our new transformed domain. While moving from one coordinate to another coordinate system, we need fix a center point as our point of reference for transformation. We choose center of the image as our reference of transformation. There are various transformation matrices that can be used to achieve swirling effect. However, all of the transformation matrices share a common idea where the angle is radius-dependent. One such matrix is:

$$T_\theta(x) = x * \cos(\varphi) + y * \sin(\varphi) + x_c$$
$$T_\theta(y) = x * \sin(\varphi) - y * \cos(\varphi) + y_c$$
$$\varphi = \frac{r}{k}\ (swirling\ angle)$$

where x_c and y_c are the center reference of the image as mentioned above; x and y represents the distance of the position of image from the center and the angle φ is dependent on radius (r) with a variable called swirling factor (k). The swirling factor can be tuned according to our will to achieve certain look. But if the value is set too less, the image gets warped in a weird fashion which may not suite our need. As a result, we need to check for a value that fulfills our requirements.

With reference to swirled image of Lena provided as an example, it was found that the image was at first horizontally flipped and then the swirling effect was applied on the image. Hence to stay consistent with our provided example, the following algorithm was followed in steps:

a) Calculate the center position of the image and invoke the user to provide a swirling factor.
b) Flip the image horizontally, which is achieved either flipping the columns of the image or creating a flip transformation.
c) Apply the afore-mentioned swirling transformation matrix on the image to obtain the desired output.

The result of the applied steps can be verified by executing the code.

1.2.2 Perspective Transformation and Imaging Geometry

In this problem, we are provided with five images and are required to project the images placed on the faces of a cube onto a 2D image plane. This is called forward mapping in imaging geometry where we map from 3D world coordinates to 2D image plane (3D-to-2D mapping). In this case we need to solve for two camera matrices in order to achieve the mapping. Let's assume for the sake of simplicity that center of camera lens is placed at (5,5,5) with a view along the direction of (-1, -1, -1) and center of the cube is placed at the origin of the world geometry. A detailed view can be seen in **Figure 1.2.1.**

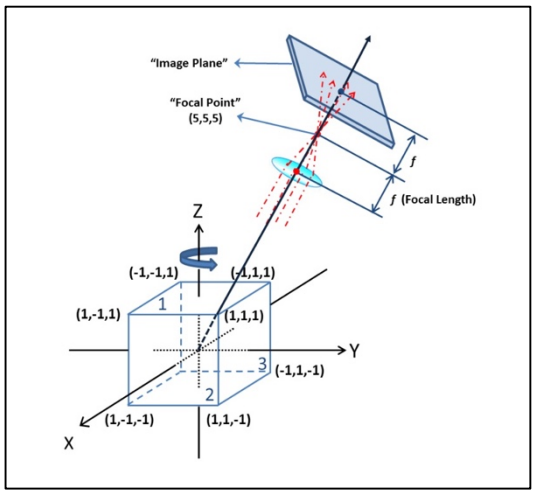


Figure 1.2.1: 3D Cube in the World Geometry

To make our mapping possible, we need to solve the following matrix for perspective transformation:

$$w\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = K[R|t] \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

where x and y corresponds to image plane coordinates and $[X \quad Y \quad Z]^T$ corresponds to 3D world coordinates; w is a scaling factor on which we will discuss later. K and [R | t] are two important camera matrices called intrinsic and extrinsic camera matrix. The purpose of extrinsic camera matrix is to map the points' location from world coordinates to camera coordinates, and the intrinsic matrix solves the

similar from camera coordinates to image plane coordinates. Both of them multiplied together gives us a transformation matrix which is used to move from 3D geometry to 2D geometry. For our given case, the matrices are found to be as follows:

$$[R|t] = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & 0 \\ -\frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} & \frac{15}{\sqrt{3}} \end{bmatrix}$$

$$K = \begin{bmatrix} \sqrt{3} & 0 & 0 \\ 0 & \sqrt{3} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Here, K includes two important points to be mentioned: a) the focal length of the image is chosen to be $\sqrt{3}$, which is problem specific; b) the image coordinates of the intersecting point between the optical axis and the image plane is brought to the origin (0,0). Once we have our camera matrices solved, we can obtain the image plane coordinates and then we have to convert the coordinates to image index based on pixel density (defined as the number of pixels per unit length), whose choice will make a difference in the result.

In terms of programming, the implementation of the entire process is quite straightforward. We divide the problem in two steps: a) one in which a program which will generate the location/intensity information of the images based on the unit length of the world coordinates between -1 and +1.; b) we implement a program to display the cube image on the image plane captured by the camera. We mainly focus on forward mapping in terms of implementation.

Pre-Processing In this step, at first we map five of our given images to each face of the cube. We disregard the bottom of the cube. When the user provides a location such as [-1,0,1] from the world coordinates, the program generates 3 intensity values of the particular image (RGB values) that will be obtained in the world coordinates. This is quite straightforward and involves simple transformed formulation. We first attempt to determine which face of the cube is the location in world coordinates is on and we get the pixel index accordingly using a simple formulation. Once we achieve the pixel index, we can search for RGB intensity values in that particular image. Let's take an example to get a detailed understanding of the process. If our first image is place on top face of the cube, we then know that x and y coordinates are varying and z is fixed. In our case, $z = 1$. If a user inputs [x,y,z] = [0,0,1], the program then knows that it is the first image and index user is looking for is the center pixel of the image via transformed formulation. In this example, the transformed formulation to find the pixel index is:

$$X\ (row\ index) = \frac{(x+1)}{2} * (Size - 1) + 1$$

$$Y\ (column\ index) = \frac{(y+1)}{2} * (Size - 1) + 1$$

where *X* and *Y* are the row and column indices of the image; *Size* corresponds to total size of the image. It is to be mentioned while programming the value is rounded to an integer since indices can not be a fraction. It is taken care such that the step size achieve is $\frac{2}{100} = .01$

Capturing 3D scene (Forward Mapping) In this step, we achieve a capture image from 3-D space to a 2-D image plane. After solving for both intrinsic and extrinsic camera matrices, we can obtain image plane coordinates as above from which we can get the index values of image that is to be placed in the particular coordinates. In case of forward mapping, we first obtain world coordinates for each pixel. Then, we obtain corresponding image indices based on transformation matrix and pixel density. Once the image indices are obtained, their corresponding pixel values are placed on the 2-D image plane which is our final capture image. There is a rotational matrix introduced in the entire coordinate system that can be tweaked to change the view of the image. It is implemented keeping the focal point fixed and the direction of camera steady (-1, -1, -1).

Z-buffering There is a caution we need to take for forward mapping. Since we are mapping 3D to 2D, there are several pixels from images that can be placed on the capture image. In this case, a technique inspired from computer graphics called z-buffering (depth buffering) is used. Each time a new pixel is encountered in the same location, the distance from pixel location to focal point (5,5,5) is stored in a buffer. Then, a comparison is held between two values and the one which is nearest to the pixel point is chosen.

Intuitively, we can take an example of observing a cube from outside such that we can only see its top, front and right face with images on it. However, if we pierce through the front face, we can then observe rear face of the cube having another image. Hence, in terms of 2-D mapping, we have two values placed on same location. It is natural to choose the one that is nearest to our point of observation since that will be visible and the other will be obscured.

Reverse Mapping The process of reverse mapping from camera coordinates to world coordinates is not quite straightforward. While in forward mapping, we map from 3 coordinates to 2 coordinates (x &y). However, in reverse mapping we need to go back from 2 coordinates to 3 coordinates. The first difficulty starts with obtaining Z information (depth) which is discarded in forward mapping. Along with that, we loose many pixels that were hidden by z-buffering. It is quite difficult to recover 3D coordinates without pre-knowledge of depth information.

1.3 Experimental Results

1.3.1 Swirl Effect

The result obtained for swirling effect for various values of swirled factor are provided in **Figure 1.3.1 & 1.3.2** (Lena & Kate respectively). Although Lena output image is already provided, I added to show the value of K for which we get the exact same image as given.

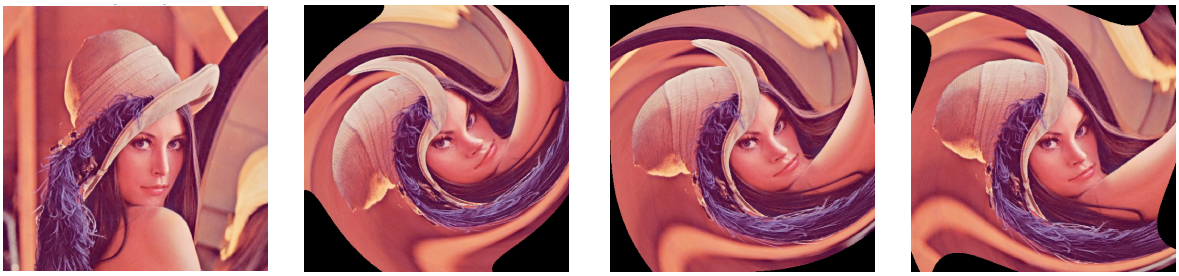


Figure 2.3.1: Left to Right: Original Image (Lenna.raw); Swirled image for K = 370, 500, 670

Figure 1.3.2: Left to Right: Original Image (Kate.raw); Swirled image for K =370, 520, 670

1.3.2 Perspective Transformation & Imaging Geometry

Before we start presenting and discussion on results, let us specify for ourselves the way the images are numbered and faces of the cube on which they are placed. **Figure 1.3.3** shows the order by which the images are numbered (from left to right).

Figure 1.3.3: Left to Right: Order of images in the cube: Image 1 > Image 2 > Image 3> Image 4> Image 5

Along with that **Figure 1.3.4** provides a view of how the unfolded cube image look like. In the figure, 6 is to be ignored since we are not placing an image in the bottom face.

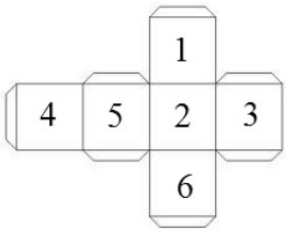


Figure 1.3.4: Unfolded cube image

Hence, it is observed that the faces of the cube will display images as written in **Table 1.3.1.**

Table 1.3.1: Designation of cube face with displayed image

Cube face	Image displayed
Top	baby.raw (image-01)
Front	baby_cat.raw (image-02)
Right	baby_dog.raw (image-03)
Rear	baby_panda.raw (image-04)
Left	baby_bear.raw (image-05)

The result obtained from the pre-processing step needs a verification on whether the intensity values obtained are correct. As discussed above, the first image placed on front face of the cube will have its center at [0, 0, 1]. So, we manually check for the intensity values in MATLAB using figure pen tool at pixel index of (100,100) for the image and compare with values generated by the program. We observe in **Figure 1.3.5** that both produces equal results, which serves to verify that our pre-processing is providing right output.

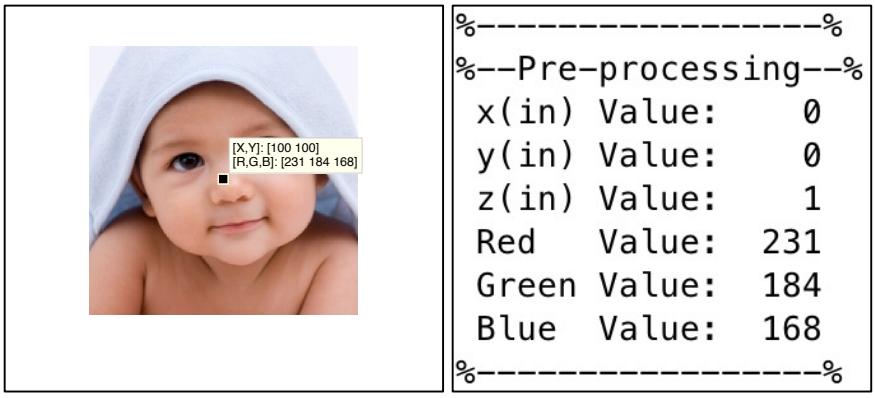


Figure 1.3.5: Left to Right: Manual checking of pixel values with pen tool in MATLAB; Intensity values generated by pre-processing code

The captured image that is obtained from the forward mapping is shown in **Figure 1.3.6**. The displayed result has a pixel density of 200 on an output image size of 200x200 with the specification as: focal length $f = \sqrt{3}$ and image width and height of 200 pixels.

Figure 1.3.6: Captured 3D image

1.4 Discussion

The results obtained from the swirl factor are quite satisfactory. Although change of K values in case of **Kate** image provides subtle changes which are not easily viewed by eyes, there are still some changes which becomes visible as we make the value of *K* higher or lower from the range (350-650) the results are displayed. Caution should be taken such that the value of *K* should not be decreased to a lower extent. **Figure 1.4.1** shows an example of how the swirled image is affected if we apply a lower value of $K = 200$. The effect is shown for both **Lenna** and **Kate** images.

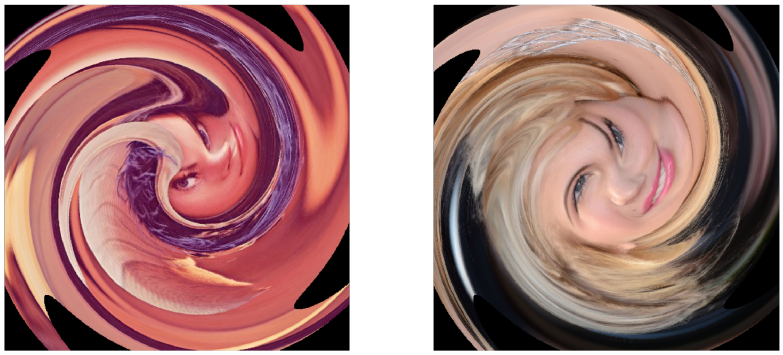


Figure 1.4.1: Swirled effect at K=200

Interestingly enough, it is also observed that if we increase the value of *K* in the range of 1000s, we get back our image as it was. It can be explained from the way we have considered our angle. Since we are dividing by a larger value of *K*, angle becomes zero, which in turn does not effect the transformation matrix. An example of swirling effect with $K = 10000$ is shown in **Figure 1.4.2**. As observed, the images are turning into their original image.

Figure 1.4.2: Swirled effect at K=10000

For perspective transformation, it is observed that the pixel density as defined above has an effect on the image. Pixel density is an important concept in analyzing the quality of image display. It is also called ppi (pixels per inches). In our case, we are considering meters as our measurement of units. It is observed that as we have a lower pixel density, we loose details in the captured image. On the other hand, we get better detailed image with increment of pixel density. Better clarity and sharpness is obtained. However, if we increase the pixel density to a greater extent we loose information on the image. Hence, we have to observe the pixel density that suits for our application. In our case, pixel density = 200 was found to be proper choice. **Figure 1.4.3** shows how variations in pixel density can make a difference in image quality.

Figure 1.4.3: Effect of pixel density on captured image (Left to Right): pixel density = 100,150,200,400

It is to be mentioned that pixel density should not be confused with resolution. Pixel density is meant to provide better clarity and sharpness to the image. It does not deal with the overall size of the image as resolution does with the image. It is observed that if the size of the captured image is increased to 800 and image is displayed with a pixel density of 800, there are certain artifacts that starts to show in the image. A type of distortion can be observed in the captured image of size 200 as well. **Figure 1.4.4** shows an example of artifacts that can be observed in the image. Source of such artifacts can be identified from the depth information provided as well as the pixel density that we can deal with in the captured image. Hence, to remove all these factors, we need to make our code smart enough to provide with better pixel density and more precise depth information.

Figure 1.4.4: Distortions and artifacts observed in the captured image

As expected if we rotate the angle of the rotational matrix introduced in the entire coordinate system by keeping the camera position fixed, we can observe the image on the other side. **Figure 1.4.5** displays the image that are obtained by changing the angle ($\theta = 0°$ and $\theta = 180°$)

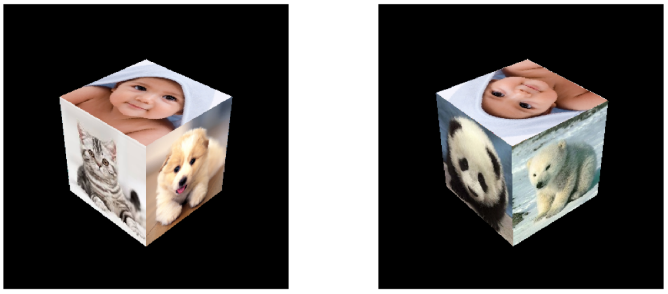


Figure 1.4.5: Cube rotated by various angles: (θ = 0° and θ = 180°)

2 Problem 2

2.1 Abstract and Motivation

Digital Halftoning is widely used in places where number of gray levels are limited to a certain range (e.g. printing, scanner) or there is an issue of getting better quality in a lesser gray values (such as, newsprint). Printers cannot produce as many colors as monitor can display. In extreme case, we either have to resort to using 0 or 1 corresponding to either a "dot" or "no dot". Hence, we use digital halftoning to create the illusion of displaying a large number of colors since human eye tries to create a gradient like imagery from a distance. Although if seen closer, a digital halftoned image may not be a better option.

In this problem, we will deal with many methods of implementing digital halftoned images. One way is using a dithering matrix whose output is based on a threshold determined from the dithering matrix. Another implementation will be based on diffusing error to the next pixels by using three sorts of error diffusion matrix. We will also be implementing halftoning on the color images. One method is by naively taking each channel separately and applying monochrome halftoning technique independently on each of them. Another method will be a better implementation which will conduct color halftoning in three channel jointly called vector color halftoning. For both color halftoning method, we will convert our image into CMYK from RGB since human color perception is based on CMY color palettes.

2.2 Approach and Procedure

2.2.1 Dithering Matrix

A very simple way to implement halftoning on a monochrome image is to use a dithering matrix which indicate how likely a dot will be turned on. The most commonly used dithering matrix is as follows:

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

where all these values indicate the possibility of the pixel being turned on in a descending order i.e. 0 indicates the pixel most likely to be turned on and 3 is the least likely one. We can recursively create $I_4(i,j)$, $I_8(i,j)$ and so on by using the following recursive formula [1]:

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

Now we move onto the steps required to implement a digital halftoned image from dithering matrix:
Step-1: Create I_4, I_8 using the recursion formula as stated above. The matrix should be of size (4x4) and (8x8) respectively.
Step-2: Normalize the pixels obtained in the above dithering matrix using the following formula:

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

where, *I(x,y)* is our desired dithering matrix and N is the number of pixels available in the matrix. In our case, N = 4 & 8.
Step-3: We multiply our threshold matrix *T(x,y)* by 255 since our original image has pixels ranging from 0 to 255.
Step-4: We run through the entire image. We use the following formula to detect whether the pixel will be turned on or not:

$$G(i,j) = \begin{cases} 255 & if\ F(i,j) > T(i\ mod\ N, j\ modn\ N) \\ 0 & otherwise \end{cases}$$

Here *F(i,j)* and *G(i,j)* are the input and output images. We have used the threshold as a determinant for the pixel to stay on or not. Finally, we obtain our output image.

2.2.2 Four intensity value screen

We can also think of a situation where our screen can display only four intensity levels. Let's say that four gray levels are 0, 85, 170, 255. We can come up with an algorithm to implement a half-toned image. The steps that are followed to implement this are as follows:
Step-1: We take the average of our threshold values as follows:

$$average1 = \frac{0+85}{2};\ average2 = \frac{85+170}{2};\ average3 = \frac{170+255}{2}$$

Step-2: Now we run through the entire image and consider three cases for each pixel values such that if the value is two threshold values i.e. i) 0 & 85, ii) 85 & 170, iii) 170 & 255. We also introduce an error (*error)* such that it is diffused in the next pixel once the previous pixel is provided a value.
Step-3: Now we can apply the following pseudo code to obtain the value of our pixel based on the corresponding threshold values and threshold averages.

```
average_n = (threshold_n1+threshold_n2)/2;
for i = 1: row
    for j = 1: col
        Id = I(r,c) + error
        if threshold_n1 < Id < threshold_n2
                if Id < average_n
                        error = Id - threshold_n1;
                        I(r,c) = threshold_n1;
                elseif Id < average_n
                        error = Id - threshold_n2;
                        I(r,c) = threshold_n2;
END PROCESS
```

Here average_n represents any threshold average taken above; *Id* is the new pixel value obtained by diffusing an error from the previous pixel and *I* is the original image. The pseudo code is written for one particular case. We apply it for all pixel values in the image in a single loop and get our desired output.

2.2.3 Error Diffusion

Error diffusion follows almost a similar concept of implementation as followed in four-intensity value screen [2.1.2]. However, this time the error is diffused in neighboring pixels based on the error matrix followed and we use hard thresholding with 128 in these cases. In this problem, we use three diffusion matrices as follows:

a) Floyd-Steinberg's error diffusion matrix:

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

b) Jarvis, Judice, and Ninke (JJN) error diffusion matrix:

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

c) Stucki error diffusion matrix:

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

The general steps that are followed to implement are as follows:

Step-1: We extend the original image so that we apply the error diffusion matrix on the pixels of our original image.

Step-2: We run through the entire image and we apply hard thresholding condition such that if the pixel value is less than 128, we turn it into 0 and if the pixel value is more than 128, we turn it into 1.

Step-3: We calculate the error by subtracting the pixel value from the value it has obtained from the hard thresholding conditioning.

Step-4: We diffuse the error to the neighboring pixels by adding the error with their corresponding scaling factors. For example: if we are applying Floyd-Steinberg's error diffusion matrix and consider that our center pixel is at position (2,2) on which hard thresholding is applied. Accordingly, we perform the calculation of error *(error)*. Now while diffusing the error to the pixel right of it, we add $\frac{7}{16} * error$ to that particular pixel. In this way, we generalize the process for all other neighboring pixels

based on the matrices being used. Scaling factor to be multiplied should be used from the corresponding location of the image.
After we iteratively continue the process for all the pixels of the image, we obtain a half toned image based on the matrix error diffusion. The error is diffused in the above mentioned method in serpentine fashion.

2.2.4 Scalar Color Halftoning

We can naively apply one of the error diffusion matrices applied on monochrome image to a color image by separating it into three channels. In this case, we will use Floyd-Steinberg's error diffusion matrix. The steps that are followed to implement this are quite straightforward and is as follows:

Step-1: Convert the RGB components of the image into CMY values by using a simple conversion formula as follows:

$$C = 1 - R; \quad M = 1 - G; \quad Y = 1 - B$$

Another method can be used to convert RGB to CMYK. This method is found to be more efficient since we have more information on Black (K) component. The procedure that is used to convert in this case is as follows:

$$K = \min(1 - R, 1 - G, 1 - B) \; or \; \max(R, G, B)$$

$$C = \frac{(1 - R) - K}{(1 - K)}$$

$$M = \frac{(1 - G) - K}{(1 - K)}$$

$$Y = \frac{(1 - B) - K}{(1 - K)}$$

Step-2: We apply Floyd-Steinberg's error diffusion matrix as our error diffusion reference to all the channels separately. Hence, we get a half-toned image for each channel in CMY

Step-3: We then convert back to RGB image by applying inverse conversion. The conversion used for another method is worth mentioning in terms of formula, which are as follows:

$$R = 1 - (C * (1 - K)) + K$$
$$G = 1 - (M * (1 - K)) + K$$
$$B = 1 - (Y * (1 - K)) + K$$

2.2.5 Vector Color Halftoning (MVBQ)

We can also apply color halftoning in three channels jointly by using a vector color halftoning technique. A method to achieve vector color halftoning was proposed by Shakad *et al.* [2] where they partitioned three color channels into 6 Minimum Brightness Variation Quadruples (MVBQ), which implies the fact that human eye mostly looks for brightness information than any other features of color. **Figure 2.1.1** shows the six tetrahedral volumes they used to find MVBQ type for each color information.

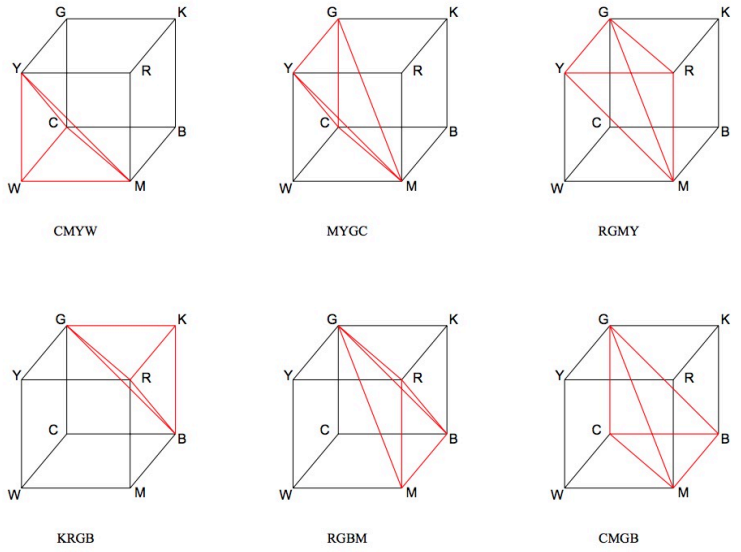


Figure 2.2.1: The partition of RGB cubes to six tetrahedral volumes. All the tetrahedrons are equal in volume, but are not congruent [2].

It is to be mentioned that the given partition is used for RGB color cubes. However, in our problem, we will be dealing with CMYK values. The steps followed to implement the algorithm are as follows:

Step-1: Convert RGB values of the image into CMYK values. We have deal with the values as a vector. Caution should be taken such that we do not take the channels separately.

Step-2: We run through the entire image ***CMY(i,j)*** and determine the MVBQ to which the vector belongs to. For this, we use the pseudo code provided in the paper [2] with modifications required for CMYK since the one in paper is written for RGB values. The pseudo code for the modified version is as follows:

```
function MVBQ (C, M, Y)
{
 if (C+M <1)
      if (M+Y <1)
           if (C+M+Y <2)        return CMYW;
           else                 return MYGC;
```

```
        else                        return RGMY;
else
  if (~(M+Y) <1)
        if (~(C+M+Y) <1)            return KRGB;
        else                        return RGBM;
else                                return CMGB;
}
```

To be noted, the notation ~ corresponds to not equal to (~=).

Step-3: We find the vertex ***V*** of MVBQ which is closest to ***CMY(i,j) + e(i,j)***. For this step, we can calculate the Euclidean distance and take the least values since we are looking for the nearest vertex or else we can hard code for each part to get the output. However, in both cases we need to consider for 6 cases of MVBQ.

Step-4: We compute the new quantization error using error using ***CMY(i,j) + e(i,j) – V*** and distribute the quantized error to the future pixels using any of the above mentioned error diffusion process applied on monochrome image. We can use Floyd-Steinberg's error diffusion in a serpentine manner to obtain our error diffused image.

Step-5: Finally, we convert our error diffused image of CMY values into RGB values. Hence, we have our final image.

We can also apply other error diffusion matrices to obtain a better result. However, for the sake of simplicity we apply Floyd-Steinberg's error diffusion to our CMY image.

2.3 Experimental Results

Before we apply our implemented algorithm, the images used for testing the algorithm are shown in **Figure 2.3.1.**

Figure 2.3.1: Original images used for the problem: (Left) Mandrill.raw ; (Right) Sailboat.raw

The results obtained from the first algorithm by applying dithering matrix on the mandrill image are shown in **Figure 2.3.2.**

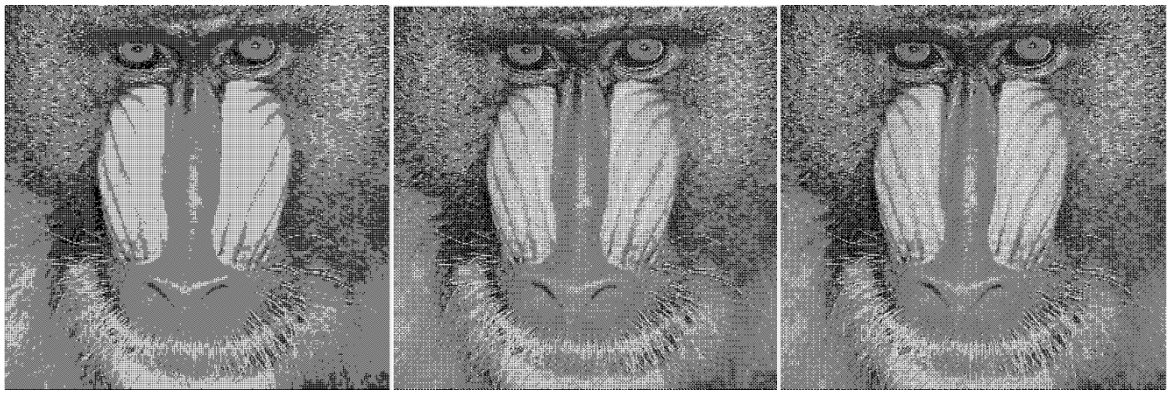


Figure 2.3.2: (Left to Right): Output for Dithering Matrix applying i) I2 matrix; ii) I4 matrix; iii) I8 matrix

The outcome that will be displayed by applying the afore-mentioned algorithm for four intensity value screen is shown in **Figure 2.3.3.**

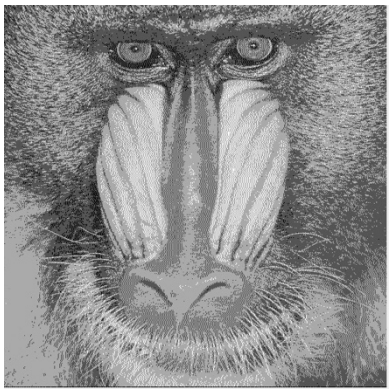


Figure 2.3.3: Output image for four-intensity level screen

The results obtained from three error diffusion process are shown in **Figure 2.3.4.**

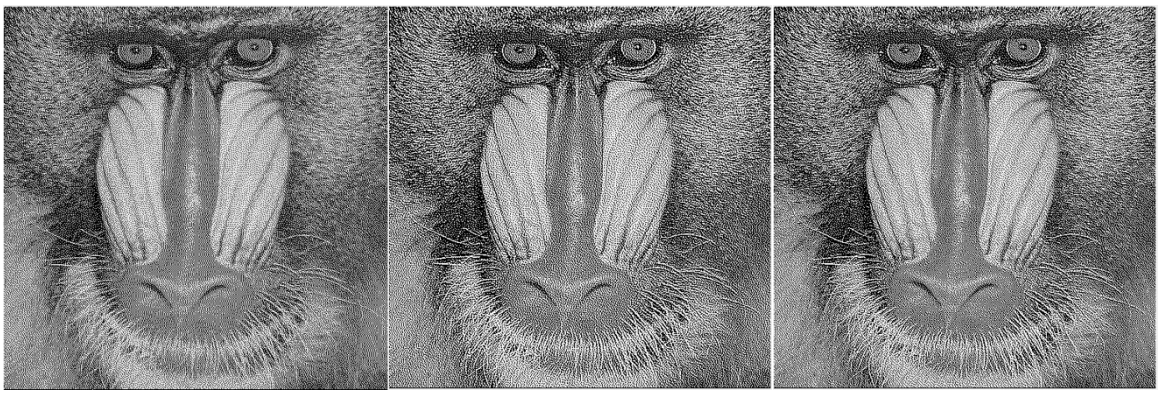


Figure 2.3.4: (From Left to Right) Half-toned image using Error Diffusion: i) Floyd-Steinberg's error diffusion; ii) Jarvis, Judice, and Ninke (JJN) error diffusion; iii) Stucki error diffusion

The converted image from RGB to CMYK of sailboat image is shown in **Figure 2.3.5.** There are two methods that were used to implement the conversion. The usage of K factor was included in the second case, which turns out to be reason for huge change in results.

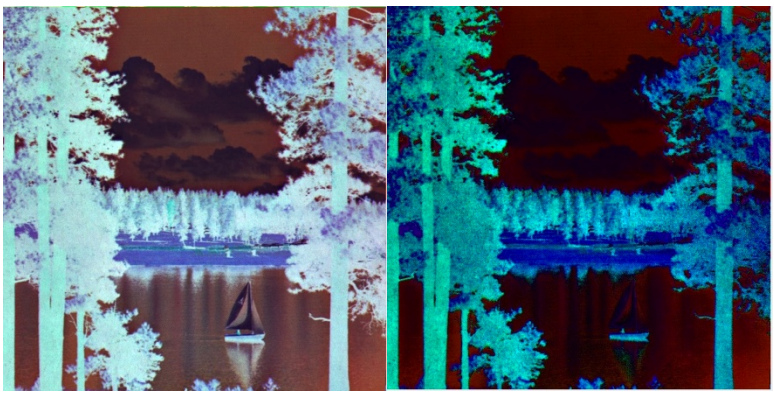


Figure 2.3.5: CMYK image of Sailboat.raw

The result obtained from scalar color halftoning; vector color halftoning without error diffusion and with error diffusion on sailboat image is shown in **Figure 2.3.6 & 2.3.7.** The result here are shown for both method of conversion used for color space: one using the conventional conversion formula and the other one is using K factor as its factor for conversion.

Figure 2.3.6: (Left to Right): Color Half-toning on sailboat.raw using i) scalar color halftoning; ii) vector color halftoning without error diffusion; iii) vector color halftoning with error diffusion. [Using conventional CMYK conversion]

Figure 2.3.7: (Left to Right): Color Half-toning on sailboat.raw using i) scalar color halftoning; ii) vector color halftoning without error diffusion; iii) vector color halftoning with error diffusion. [Using CMYK conversion –second method]

2.4 Discussion

It is seen that as we extend to use bigger dithering matrix, our results get better. The difference between three of the results obtained for dithering (Fig. 2.3.2) shows that I_8 provides more details. Intuitively, we can think of adding more conditions helps us getting better results. I tried to check for I_{16} and I_{32} **(Figure 2.4.1)** and found out that it is not always true that bigger matrix will provide us with better results. The difference between I_8 and I_{16} is very subtle. However, the artifacts and darkness observed in I_{32} shows that dithering matrix is not always a better option to be chosen.

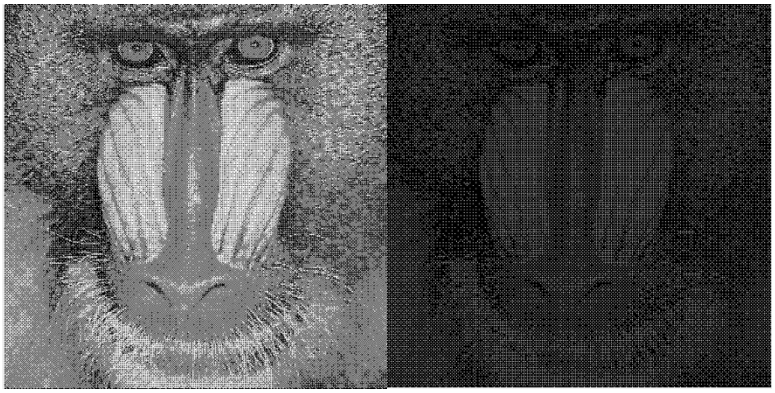


Figure 2.4.1: (Left to Right): Output for Dithering Matrix applying i) I16 matrix; ii) I32 matrix

The pattern of artifacts obtained in dithering matrix is obvious from the results. However, error diffusion provides better results and does not show any pattern of artifacts in the output. Along with that, the output of four intensity valued screen is better than all of them if we observe by zooming in. It is more detailed and more contrast is obtained. It does not show any pattern of artifacts like dithering matrix. **Figure 2.4.2** gives a detailed zoomed in view of the algorithms implemented. More details can be seen in the eye in case of four intensity valued screen.

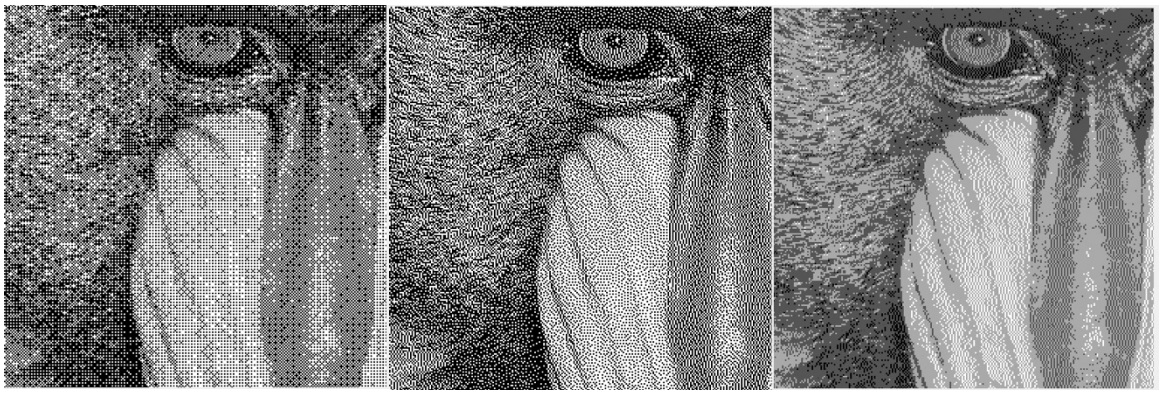


Figure 2.4.2: (Left to Right): Zoomed-in view of output half-toned image using: i) Dithering matrix; ii) Floyd-Steinberg's error diffusion; iii) Four intensity valued screen.

The concept of error diffusion can be related to adding noise to the image. In theory, it is possible to add random noise to the original image and use fixed thresholding to create a half-toned image.

However, practically it is very hard to create random noise. Hence, these error diffusion matrices mimic the behavior of adding noise to the image and thus getting a better output. The differences between three error diffusions are really subtle to be distinguished. Since the size of window applied for Floyd-Steinberg is smaller than the other two, Floyd-Steinberg diffuses error more than the other two error diffusions leading to a more smoothened error diffused result. Because of a larger window size, last two error diffusions are supposed to provide a more blurred image. However, it is very hard to find out from the results about these assumptions.

Two methods of color halftoning shows that it is a better option to conduct halftoning in three channels jointly i.e. taking them as a vector. The primary disadvantage of scalar halftoning is that it does not consider human visual characteristics [3]. Since it joins different artifacts obtained in three channels separately, it provides low frequency noise pattern. The low frequency noise pattern can be observed in **Figure 2.3.6** (Leftmost figure).

Vector color halftoning was found to provide different results based on the method use for conversion of CMYK-RGB. The second method, which is more precise, has been seen to provide better results compared to the conventional definition of CMY. Intuitively, it makes sense to add a K factor in the conversion since our algorithm for vector color halftoning includes K in its tetrahedron. Apart from that, it is seen from Figure 2.3.5 that conversion with CMYK considering K factor provides more information on the color values. Provided that the second conversion method is used to get the output, vector color halftoning seems to provide a better result than scalar halftoning. Low-frequency noise components are seen to be removed and the output looks to consider human visual characteristics. A side-by-side comparison of two images provide with better visual analysis **(Figure 2.4.3).** The clouds of the sky have more details and is less obscured by noise in the second image (from the left).

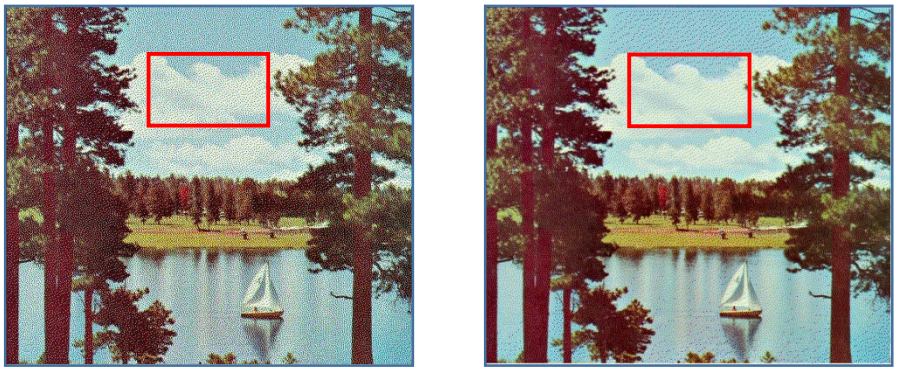


Figure 2.4.3: (Left to Right): i) Low noise frequency component in scalar color halftoning; ii) lesser noise frequency artifacts observed in vector color halftoning

Applying other two error diffusion matrices, it is observed that the images obtained had a color shift **(Figure 2.4.4).** It is possible that the error diffused in the vector was made too high such that the nearest vertex obtained moved from one tetrahedron to another. Hence, such shift in color is obtained.

Figure 2.4.4: (Left to Right): Vector color halftoning using i) Jarvis, Judice, Ninke (JJN) error diffusion; ii) Stucki error diffusion

There are many self-implementations I would like to attempt for monochrome halftoning. One of them using the algorithm implemented for four intensity value screen and Floyd- Steinberg's error diffusion altogether. Hence, we place more conditioning on the thresholding unlike the error diffusion hard thresholding method; which seemed to have provided detailed results than the others. Another method I would like to implement is adaptive threshold for error diffusion in which I would like to use dithering matrix as our threshold for determination of error diffusion. The method is almost similar as the one described. In this method, the thresholding is dependent on dithering matrix. So, the error will be calculated by subtracting dithering matrix from the pixel value, and the error will be diffused to the neighboring pixels using either of the error diffusion matrix.

3 Problem 3

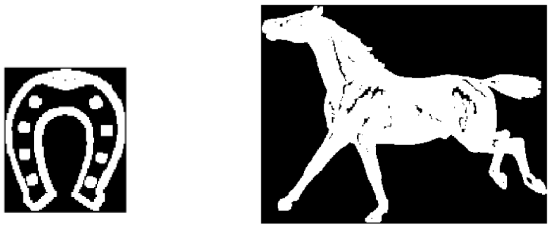
3.1 Abstract and Motivation

Morphological processing is mainly based on geometrical structure of the image. There are various steps taken morphological operations can be applied to an image. Such steps include dilation and erosion of image, creating a closed and open image, pre-processing steps such as, hole-filling filter, boundary smoothing filter. This sort of processing is widely used in medical forensics, finger-print detection, recognition and interpretation of objects in a scene.

In this problem, there are mainly three operations that will be performed: shrinking, thinning, and skeletonizing. Although it is hard to define the operations precisely, the mask from the pattern table comparing with which they are formed from actually make them unique. The usage of pattern table will be discussed broadly in Approach and Procedure [3.2]. We are provided with binary images which saves us a step from converting an image from grayscale to binary image. Otherwise we have to convert a grayscale image into binary image through thresholding.

3.2 Approach and Procedure

Before we start our discussion on algorithm used to implement, the images used for testing in this problem are attached in **Figure 3.2.1**.

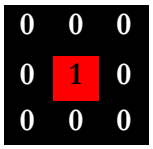


(i) (ii)

Figure 3.2.1: (From Left to Right) Original image provided: i) Horseshoe.raw; ii) Horse1.raw

3.2.1 Dilation, Erosion, Open image, Closed image

Before we move onto applying morphological processing on the image, we have to take care of certain concepts using which a better performance is obtained in the operations. Two of such concepts are dilation and erosion. The idea behind dilation is that it causes objects to grow in size whereas erosion causes objects to shrink. The amount by which dilation and erosion occur depends on the choice of the structuring element. A usual choice of structuring element **B** is taken as follows:



From the structuring element, dilation is defined as [4]:

$$A \oplus B = \{z | (Z = a + b, a \in A, b \in B\}$$

Erosion can be defined as follows [4]:

$$A \ominus B = \{z | (z + b \in X, b \in B\}$$

Hence, dilation makes an object larger by adding pixels with values of '1' around its edges whereas erosion makes it smaller by removing the pixels from its edges. In terms of programming, it is quite straightforward to create a function for erosion and dilation. We pass a window of size 3x3 and while passing it through the elements in the original image, we replace the center pixel by the maximum of the values in the window in case of dilation. For erosion, we replace the central pixel by the minimum of the values collected in a window.

From this we obtain the very important concept of opening and closed images. By changing the order of operation of erosion and dilation, we get two different results. This is due to the fact that erosion is not commutative. A closed image is obtained by dilating the image first and eroding the dilated image. On the other hand, an opened image is obtained by eroding the image first and dilating the eroded image. We can define the closed and open image as follows:

$$A \blacksquare B = (A \oplus B) \ominus B$$
$$A \odot B = (A \ominus B) \oplus B$$

where ■ denotes closed image and ⊙ denotes open image.

3.2.2 Hole-filling filter & Boundary smoothing filter

A hole is defined as a background region (0) surrounded by a connected border of foreground pixel values (1). While performing thinning and skeletonizing, we need to perform the pre-processing step of filling the hole. Hence, we need to implement a hole-filling filter. The main purpose of hole-filling filter is to fill all the holes in the binary image. The basic formula used to implement a hole-filling filter is:

$$X_k = (X_{k-1} \oplus B) \cap A^c$$

Here, we are basically performing a dilation operation with the previous pixel values and then taking the common element between dilated operation and the complement of ***A*** whose elements are 8-connected boundaries with each boundary enclosing a hole [4]. In terms of we need to look for the patterns of such intersection and apply the above formula to obtain our desired result. In this case, **B** is considered as follows:

0	1	0
1	1	1
0	1	0

It is a common mistake to think of hole-filling filter equivalent of dilation. However, it is not a true statement. In hole-filling, we first apply the dilation and then take the intersection of dilated image with complemented A which limits the filling operation inside the region of interest. It is often known as conditional dilation [4].

Boundary smoothing filter is applied on the hole-filled image since it leaves extra pixels because of its dilation operation. Hence, we need to smoothen the boundary so that we do not loose the geometric structure of the object. As we are applying a smoothing filter, it is enough to use an averaging filter, so called Gaussian averaging filter. It should be mentioned that once we apply averaging filter, we have to convert our image back to binary image through thresholding.

3.2.3 Shrinking, Thinning, Skeletonizing Algorithm

The procedure followed to implement shrinking, thinning and skeletonizing is similar. Hence, we can take a look at their implementation all at once. The procedure of implementation can be divided into two steps. Let us suppose that F(u, v) is our given image, M(u, v) is our secondary image implemented from the first step, and G(u, v) is our final morphological image implemented from the final step.

The top view of the algorithm is shown in **Figure 3.2.2:**

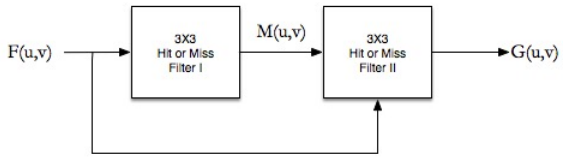


Figure 3.2.2: Top view of the algorithm

Step-1: 3 X 3 Hit-or-Miss Filter I in **Figure 3.2.3**

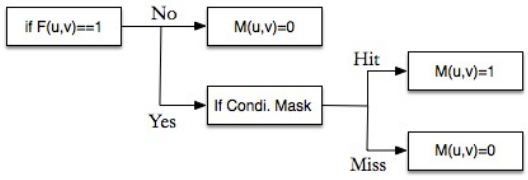


Figure 3.2.3: Step-1 implemented using Hit-or-Miss Filter I

Step-2: 3 X 3 Hit-or-Miss Filter II in **Figure 3.2.4**

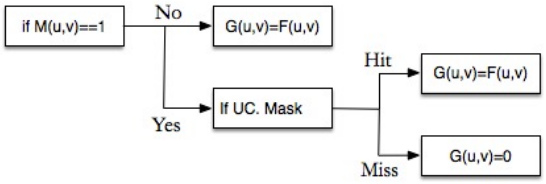


Figure 3.2.4: Step-2 implemented using Hit-or-Miss Filter II

In the first step, we compare the central pixel of our original image F(u,v) with the conditional mask provided in pattern table in a 3x3 window. In the second step, we compare the central pixel of our mask image M(u,v) with unconditional mask to obtain our final G(u,v). The difference between shrinking, thinning and skeletonizing happens exactly in the steps of comparing. In the pattern table,

shrinking, thinning and skeletonizing are denoted by S, T, K respectively. There are masks that are particularly allocated for each operation. We use them accordingly to find the result.

For the step of creating pattern tables, all the masks were hard coded to make sure that any mask is not missed while comparing. In case of D= 0 or 1(don't cares), A or B or C =1, all the combinations were considered to create the masks. For example: if a mask contained 3 D's, the pattern was created for all combinations of D, which is 8 (= 2^3) in number.

Counting the number of nails & holes B To count the number of nails and holes in Figure 3.2.1(i), shrinking was used which brings the pixels down to one pixel but preserving the shape. So, we can count the number of isolated pixels in an image which would provide us with the total number of that particular identity available in the image. To find the number of nails we calculate the number of isolated pixels in the image after we apply shrinking. On the other hand, to find the number of holes, we can invert the image (0's turning into 1's and 1's into 0's) and look for the isolated black pixels which are the holes.

Thinning and Skeletonizing We apply thinning and skeletonizing on the image in Figure 3.2.1(ii). At one time, thinning and skeletonizing is performed without any pre-processing steps. At another step, thinning and skeletonizing is performed with pre-processing steps, such as: hole-filling and boundary smoothing.

3.2.4 Connected Component Labeling

Connected component labeling is used to uniquely label subset of connected components. Connected component labeling should not be confusion with image segmentation. Image segmentation provides us the result of different segment regardless of pixel values whereas connected component labeling is done on a binary image to find unique segments or partitions.

The algorithm used in this problem to count the number of white segments or objects in Figure 3.2.1(i) is inspired from a faster scanning algorithm developed by Lifeng *et al.* [5]. The implementation of the algorithm follows two-pass method:

First pass:

1) We raster-scan through each pixel of the image.
2) If the current pixel is not 0 i.e. 1 in our case
   i) We get the neighboring pixels of the current pixel.
   ii) If there are no neighboring pixels, the pixel is uniquely labeled.
   iii) Otherwise, we find neighboring pixels with the smallest label and denote the current pixel by that label.
   iv) We also store the equivalence among neighboring pixels.

Second pass:

1) We raster-scan through each pixel of the image again.

2) If the current pixel is not 0
   i) We re-label the pixel with the lowest equivalent label.
   ii) Otherwise, the label stays on the current pixel.

Counting the number of white objects At first, hole-filling filter was applied as a pre-processing step. The above mentioned algorithm was implemented to segment the horseshoe image (Figure 3.2.1(i)) to obtain the number of white objects or segments present in the image. Although it is easy to determine with naked eye the number of white objects, the importance of algorithm is evident while applying on a horseshoe image with multiple nails or an image with a lot of stars. A color labeling was done on the image to verify the two-pass algorithm.

3.3 Experimental Results

The number of nails and holes obtained from the implementation of algorithm is shown in **Figure 3.3.1.**

```
Number of nails:   8
Number of holes:   3
```

Figure 3.3.1: Results obtained for Problem 3.a.1

The number of white objects obtained in the image using two-pass scanning algorithm is shown in **Figure 3.3.2.**

```
Number of white objects:   9
```

Figure 3.3.2: Results obtained for Problem 3.a.2

The segmentation of white objects obtained from two-pass algorithm of connected component labeling can be shown using an interactive image of color labels for each white objects labeled. It is shown in **Figure 3.3.3**. Different colors are used to represent different segments. If we count the color labels, it is found to be nine (9) in number which is the output obtained from the algorithm.

Figure 3.3.3: Color labeled image using connected component labeling

The results obtained from Thinning and Shrinking without pre-processing steps (hole-filling & boundary smoothing filter) are shown in **Figure 3.3.4** & with pre-processing steps are shown in **Figure 3.3.5.**

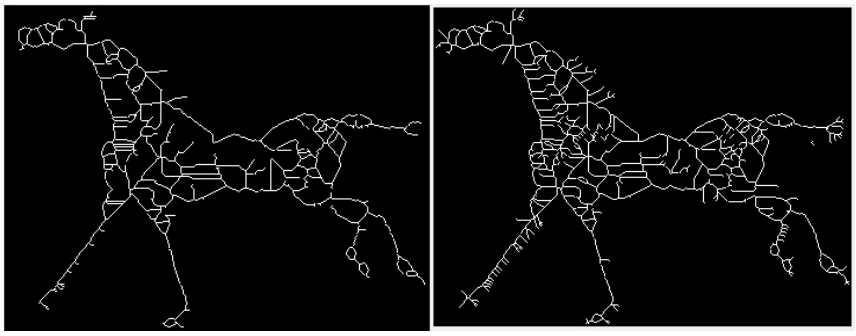


Figure 3.3.4: (Left to Right): Morphological Operations without pre-processing i) Thinning; ii) Skeletonizing

Figure 3.3.5: (Left to Right): Morphological Operations with pre-processing i) Thinning; ii) Skeletonizing

3.4 Discussion

It is observed in case of horseshoe.raw that shrinking operation worked fine to provide us with the count of nails and holes. If we count the number of nails and holes manually, it is verified that the numbers are correct. Although it is very hard to appreciate the counting ability achieved due to shrinking in our example, there are cases where shrinking has proved its ability in counting the number of stars or number of pathways in a PCB circuit bringing the pixels down to one component. To appreciate the least, it can be observed from the shrunk image that isolated pixels were properly created through shrinking morphology as shown in **Figure 3.4.1.**

Figure 3.4.1: Shrunk image of horseshoe.raw

To count the number of holes, the inversion of image was performed based on the assumptions that holes are black isolated pixels in the image. Hence, inverting the image gives us more flexibility to count the number of white isolated pixels function that was developed was counting the number of nails. An inverted shrunk image with a zoomed-in view of the holes are provided in **Figure 3.4.2.**

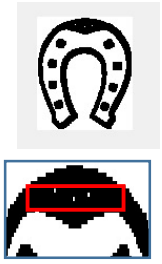


Figure 3.4.2: Inverted image of shrunk horseshoe.raw

The number of white objects in horseshoe.raw is also verified using color labeling as shown in Figure 3.3.3. While considering neighboring pixels in the two-pass algorithm, we do not consider the diagonal pixels since they may create a conflict in labeling for other regions. This is taken care in the second pass. It is enough to consider only north and east pixels. However, in this problem, I considered north, south, east and west pixels to check for the results. The result was similar in both cases.

It is to be mentioned that the number of iterations and the way the image is eroded and dilated makes a difference in finding the result. It was observed that without any dilation and erosion, it required almost 150 iterations to get to a proper shrunk image. However, with right ordering of dilation and erosion, it was observed that number of iterations came down to 15. In **Figure 3.3.3**, a series of shrunk images are shown at different iterations. As we decrease the number of iterations, we decrease the time it requires the program to generate shrunk images. However, to be on the safe side, the image was shrunk for 40 iterations.

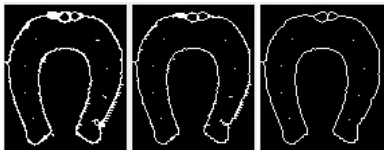


Figure 3.4.3:(From Left to Right): Shrunk images at iterations of 4 times, 6 times and 15 times respectively

For skeletonizing and thinning, it is evident from the results shown in Figure 3.3.4 and Figure 3.3.5 that pre-processing makes a difference in finding a better result. The principle reason can be found from the image in Figure 3.2.1 (ii). It is seen from the image there are interior holes in the body of horse which are not filled. Hence, we use hole filling filer to fill those holes of the body without adding excess pixels in the border unlike erosion. Next, we apply a boundary smoothing filter to smoothen the contour of the structure. The boundary smoothing filter also helps us in getting a better result in removing spurs from the final skeletonized and thinned images. The thinning results for various iterations starting from 20 to 70 are shown till we reach our final result in **Figure 3.4.4**. The results for shrinking are also shown from 20 to 70 iterations in **Figure 3.4.5**. These results are obtained with pre-processing steps.

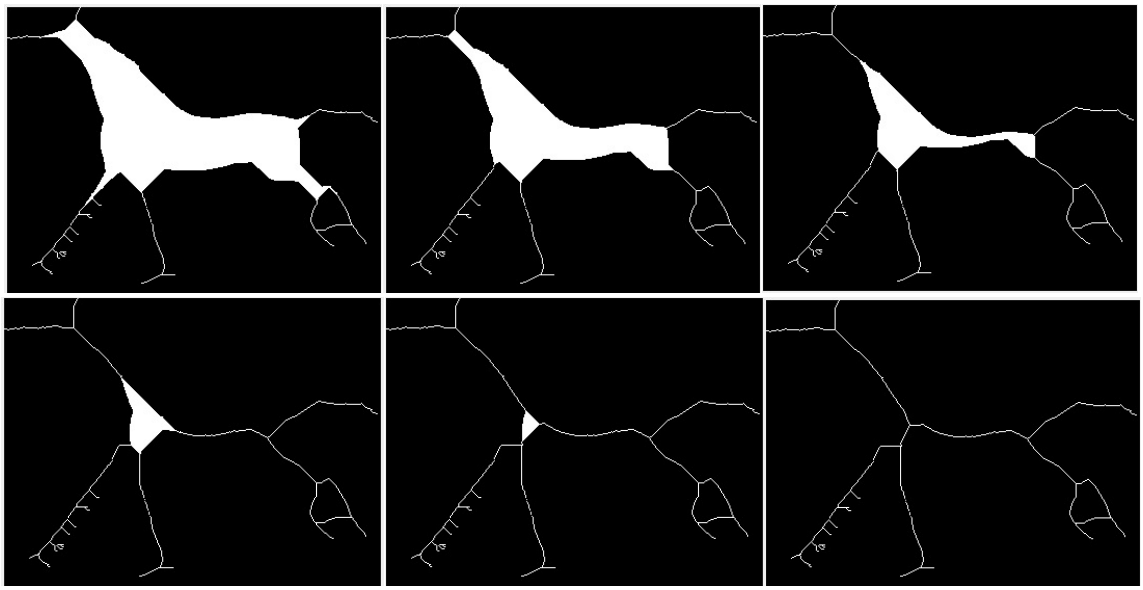


Figure 3.4.4: (From Left to Right): Thinning results for iterations of 20,30,40;50,60,70

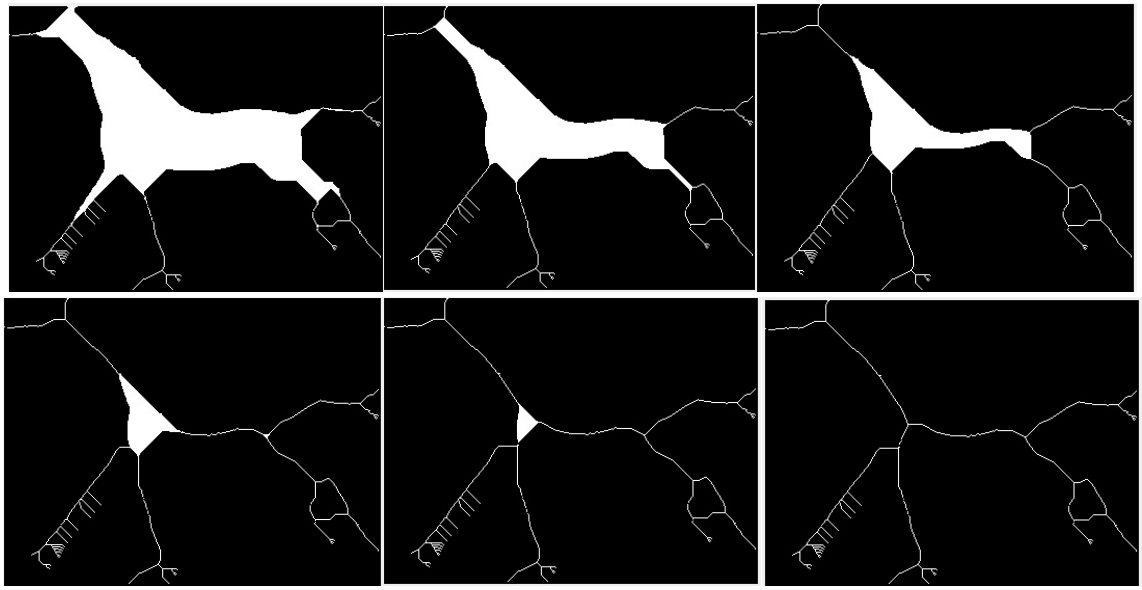


Figure 3.4.5: (From Left to Right): Skeletoning results for iterations of 20,30,40;50,60,70

The results are almost comparable to MATLAB's built-in function found using **bwmorph**. However, MATLAB's functions are faster than my implementation and uses a different algorithm which are considered to be state of the art and uses lesser conditions to compare for the mask image we obtained. One such implementation can be found in [6] which uses lesser patterns and lesser iterations required for the implementation. MATLAB's implementation are attached with the pre-processing steps added

in **Figure 3.4.6**. We can see that MATLAB's implementation provides more morphological structured with lesser spurs than our implementation. However, our morphological processing

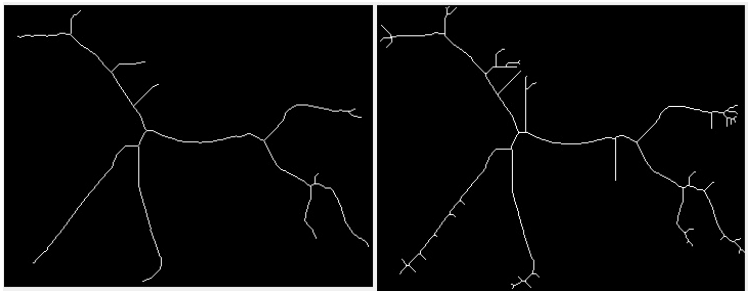


Figure 3.4.6: (From Left to Right): MATLAB's implementation of i) Thinning; ii) Skeletoning

Although the results obtained from the implementation using pattern tables are satisfactory, there are certain places where errors can originate from. One is thresholding the boundary smoothened image to a binary image. The threshold used in this case can vary from image to image. Hence, hard thresholding is not the right way to approach the problem. Careful steps should be taken in terms of applying pre-processing steps. Overdoing pre-processing may lead to unexpected results. **Figure 3.4.7** shows two examples where overdoing of pre-processing steps led to discontinuity in the morphological shape. The last image includes an example of how absence of boundary smoothing filter can make a difference. Hence, boundary smoothing filter plays an important role in removing spurs. Hence, precaution must be taken in regard to pre-processing steps.

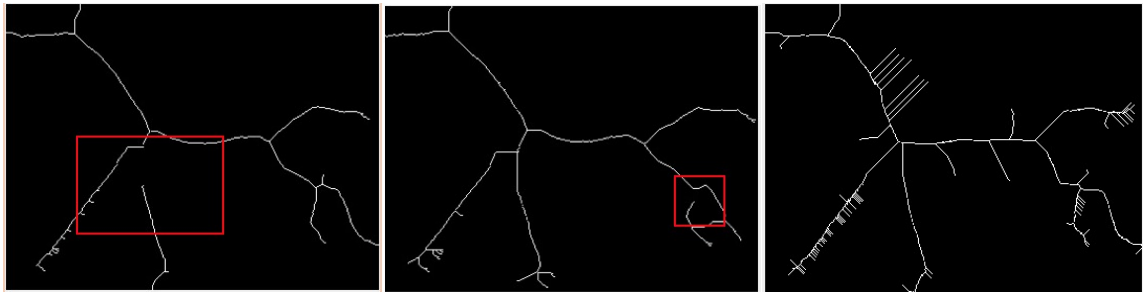


Figure 3.4.7: (Left to Right): i) Overdoing of pre-processing on thinning; ii) Overdoing of pre-processing on Skeletonizing; iii) Effect of not using boundary smoothing filter

References

[1] B. E. Bayer, "An optimum method for two-level rendition of continuous-tone pictures," SPIE MILE- STONE SERIES MS, vol. 154, pp. 139–143, 1999.

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

[3] S. Chang Lee, Y. Tae Kim, Y. Cho and Y. Ha, 'Improved Vector Error Diffusion for Reduction of Smear Artifact in the Boundary Regions', *Proc. SPIE 5008, Color Imaging VIII: Processing*, 2003.

[4] Academia.edu, 'Morphological Image Processing', 2015. [Online]. Available: http://www.academia.edu/3482702/Morphological_Image_Processing.

[5] Lifeng He, Yuyan Chao and K. Suzuki, 'A Run-Based Two-Scan Labeling Algorithm', *IEEE Transactions on Image Processing*, vol. 17, no. 5, pp. 749-756, 2008.

[6] L. Lam, S. Lee and C. Suen, 'Thinning methodologies-a comprehensive survey', *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 9, pp. 869-885, 1992.

[7] http://sipi.usc.edu/database/

[8] Discussion Notes from EE 569, Fall 2015.