Amruta Kulkarni,
USC ID: 6914970827
Email: arkulkar@usc.edu
Date: 1 Oct 2015

EE 569 HOMEWORK #3

1. Problem #1: Geometrical Modification

1.1 Swirl Effect

1.1.1 Motivation

Digital Images sometimes need to be manipulated either to make corrections to the image or for creative purposes. An image can be transformed to a whole new image with very interesting effects. One such technique is the swirl effect. After giving this type of effect to an image, we get a twirled and rotated image. In this problem, we will learn the implementation of this effect.

1.1.2 Approach and Implementation

- To start with, we have a 24 bits RGB image of dimensions 512 × 512.
- Images are stored according to the image co-ordinates that have their origin at the top-left corner of the image.
- The horizontal co-ordinates (i.e. u) increase from left to right whereas the vertical co-ordinates (i.e. v) increase from top to bottom.
- We first need to convert these image co-ordinates to Cartesian co-ordinates so that we can do some geometric transformations to our image.
- Now, our image has ‘u’ from 0 to 511 and ‘v’ from 0 to 511 as well.
- In Cartesian co-ordinate system, we want the origin of the image to be at the center.
- So we basically want to place the image in the x-y plane such that the center pixel of the image lies at 0,0.

- To do this, we first calculate the midpoint of the image that, in our case, is 256,256.
- Now, in programming convention, row corresponds to the height (so y) and column corresponds to the width (so x).
- Thus, to get Cartesian y of the image, we subtract the row (u) from the midpoint and to get Cartesian x of the image we subtract the midpoint from the column (v).
- Thus, now the top-left corner of the image lies in the second quadrant, top-right in the first quadrant, bottom left in the third quadrant and bottom-right in the fourth quadrant.
- After we get the Cartesian co-ordinates, we now need to convert them into polar co-ordinates where we gat the magnitude (‘r’) and the angle (‘θ’) of every location with respect to the origin.
- This conversion to polar is done using the following 2 formulae:

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

- Now, once we obtain the angle and the magnitude of our pixel Cartesian co-ordinate, we need to decide the amount of rotation.
- This rotation will be magnitude dependent, meaning that the rotation angle will be decided on the basis of the distance of that pixel from the origin (i.e. ‘r’).
- The formula that we have used in this case is:

  Rotation angle = θ – (r / scaling factor) + (90 × π / 180)

- Scaling factor in our case is 160 but it can be varied depending upon how much swirling we want to create.
- Thus, we have replaced the original θ with a new angle (say Φ).
- Now, we want to go back to the Cartesian domain from the current polar domain.

- To do this we use the following two formulae to get 'X' and 'Y':

$$X = r \times \cos(\Phi)$$
$$Y = r \times \sin(\Phi)$$

- Now, we want to go back to the image co-ordinates because that is where we had started.
- To do this, we simply add the midpoint values of u and v to the new Cartesian values X and Y.
- Thus, we are back to the u-v plane.
- It might be the case that our u-v values are not integers and we know that pixel locations are always integers. Thus, to get the pixel value at such non-integer positions, we can use bilinear interpolation.
- The following figure depicts how to calculate pixel values at locations that aren't integer values:
-

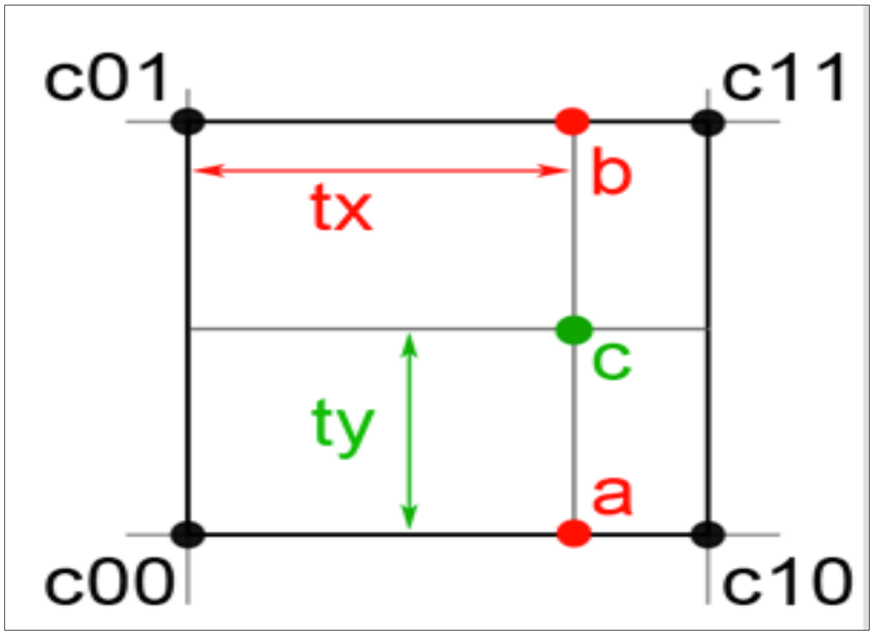


Figure 1.1 Bilinear Interpolation

- Now, we want to obtain the intensity value at 'C' by using the intensity values at its nearest 4 neighbors 'C00', 'C10', ' C11' and 'C01'.

- This is obtained by using the following formula:

$$\boxed{C = (1-tx)(1-ty)C00 + (1-tx)(ty)C01 + (tx)(1-ty)C10 + (tx)(ty)C11}$$

- In this manner, we have successfully created the swirling effect but there is one last thing that we need to consider.
- After we got the final u, v values, it might not be the case that they still lay in between 0 and 511.
- Thus, the pixel values whose location is out of this bound will be set to zero.
- By doing this, we are ensuring that the image is fit properly into the same dimensions as before.
- Thus, we now place pixels at their corresponding new u and v values and this new image has the desired swirling effect.
- This method has been implemented in C++.

1.1.3 Results

The original image and the image with swirling effect are as shown below:

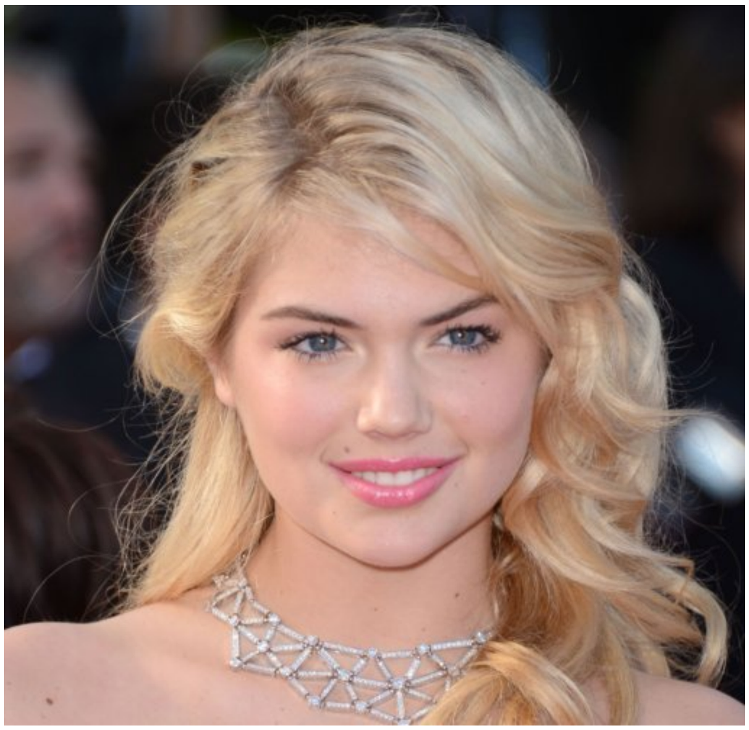


Figure 1.2(a) Original image

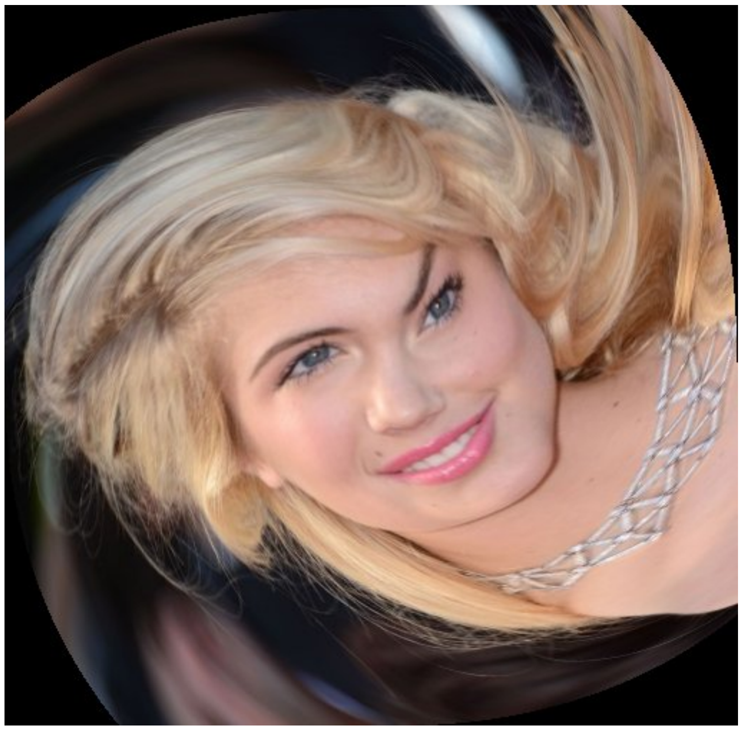


Figure 1.2(b) Image with the swirling effect

1.1.4 Discussion

From the above result we can observe the swirl effect on the original image. Thus, we can conclude that we can make geometric modifications to our image so as to get some interesting effects. To be able to do that, we always need to convert the image co-ordinates domain to the Cartesian co-ordinates domain. We can then use simple geometric formulae to rotate, scale or shift the image pixels.

1.2 Perspective Transformation and Imaging Geometry

1.2.1 Motivation

Images usually have 2 dimensions, whereas the object, whose image is captured using a camera, has 3 dimensions. Now, if given an object, we need to have some knowledge about the way we can capture this object using a camera to create its image. This problem addresses this procedure of converting a scene in 3 dimensions to an image in 2 dimensions. Concepts of world co-ordinates and camera co-ordinates come in handy in performing this process.

1.2.2 Approach and Implementation

1.2.2.1 Pre-Processing

- We are provided with a set of 5 different images and now we want to place them on a cube to create our object.
- Out of the 6 faces of the cube, only 5 can be seen.
- We have a camera placed at (5,5,5) over the cube in the 3D world geometry.
- The following figure shows the cube and the camera:

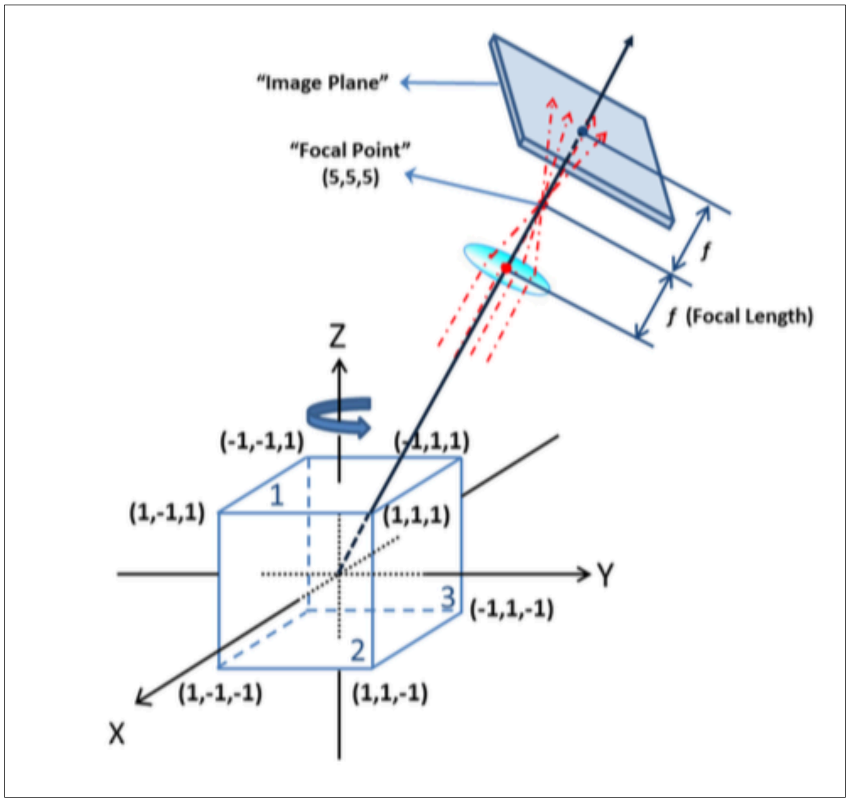


Figure 1.3 3D cube in world geometry

- Now, we want to place the 5 images on each side of the cube.
- The side marked as 1 will have an image of a baby, side 2 will have an image of a baby cat, side 3 will have an image of a baby dog, (side 4 cannot be seen from outside) side 5, that is opposite to side 3, will have an image of a baby panda and side 6, that is opposite to side 1, will have an image of a baby bear.
- Now, all these 5 images have dimensions of 200 × 200 and they are in the image co-ordinates system (u-v plane) i.e. their origin is at the top-left corner.
- We first need to convert the image co-ordinates into Cartesian co-ordinates.
- To do this, we will just subtract the midpoint of u and v (i.e. 200/2) from each row and column number of the 200 × 200 matrix.
- By converting the image co-ordinates into Cartesian co-ordinates, we are just shifting the origin of every image at the center of the image (100,100).
- Thus, now the top-left corner of each image will lie in the 2nd quadrant, the top-right corner will lie in the 1st quadrant, the bottom-

left corner will lie in the 3rd quadrant and the bottom-right corner will lie in the 4th quadrant.

- Now, we want to go from 2 dimensional Cartesian co-ordinates system to a 3 dimensional world co-ordinates system.
- Image 1 will have its world X and world Y co-ordinates in the range of -1 to 1 whereas the world Z co-ordinate will be equal to 1.
- Image 2 will have its world Y and world Z co-ordinates in the range of -1 to 1 and the world X co-ordinate will be equal to 1.
- Similarly we need to figure out the world X, world Y, world Z co-ordinates of the remaining 3 images.
- Thus, in the world co-ordinates system, each image will have 2 co-ordinates in the range of -1 to 1 and the third co-ordinate will be equal to 1.
- Thus, we just have to perform rescaling of the Cartesian X and Y co-ordinates so that they now lie in the range of -1 to 1 instead of -100 to 100.
- After rescaling the X and Y co-ordinates of every image, we need to assign them to the appropriate world co-ordinates.
- For e.g. if we consider image 3, we know that its world X and world Z co-ordinates are from -1 to 1 whereas the world Y co-ordinate is equal to 1. Thus, we just allocate Cartesian X to world X and Cartesian Y to world Z co-ordinates.
- Similarly, we get the world co-ordinates of the remaining images.
- We have now placed the 5 images on the 5 sides of the cube.
- The world X, world Y and world Z co-ordinates of every image are stored in 5 different 3 dimensional 200×200 matrix.
- This is how we go from the image co-ordinates to the world co-ordinates.
- Thus we have the location information of all the 5 images.
- We can also store the RGB content of each image by using a 6 dimensional 200×200 matrix instead of 3 dimensions.

1.2.2.2 Capturing 3D scene

- From figure 1.3, we can conclude that the camera can see only 3 faces of the cube that correspond to the first 3 images (baby, baby cat and baby dog ☺).
- So in this part, we will consider only those 3 images.

- Now, we want to go from the world co-ordinates (obtained from above) to the camera co-ordinates and from there back to our image co-ordinates.
- Thus, we want to get back our Cartesian X and Y from world X, world Y and world Z co-ordinates with the help of camera co-ordinates.
- The following formula can now be used:

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

- The first matrix is an intrinsic camera matrix denoted by K whereas the second matrix is the extrinsic camera matrix denoted by [R|t].
- X,Y,Z are the world co-ordinates and they are augmented by 1.
- w is just the scaling factor which is equal to z that we will come across later.
- We will first apply the extrinsic matrix on the world co-ordinates of the 3 images separately and store the camera co-ordinates in a new 3 dimensional 200× 200 matrix.
- Extrinsic matrix is used for conversion of world co-ordinates into camera co-ordinates.
- Now, we already have pre-defined parameters as follows:

$$[R|t] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} = \begin{bmatrix} X_c^X & X_c^Y & X_c^Z & -\mathbf{r}\cdot\mathbf{X_c} \\ Y_c^X & Y_c^Y & Y_c^Z & -\mathbf{r}\cdot\mathbf{Y_c} \\ Z_c^X & Z_c^Y & Z_c^Z & -\mathbf{r}\cdot\mathbf{Z_c} \end{bmatrix}$$

Where $\mathbf{X_c} = (-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0)^T$, $\mathbf{Y_c} = (\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}, -\frac{2}{\sqrt{6}})^T$

$\mathbf{Z_c} = (-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}})^T$

and

with r = 5 as the camera is placed at (5,5,5) w.r.t. the origin i.e. (0,0,0).

- Now, all we do is simply multiply this matrix to the 3 world-co-ordinates belonging to every location of all the 200 × 200 pixels of every image.
- This will result into new co-ordinates X_c, Y_c and Z_c.
- We do the same operation to all the pixel locations of all the 3 images and thus get the new matrix of camera co-ordinates of the 3 images.
- Now, we need to consider the intrinsic matrix and again we have a set of pre-defined parameters as given below:

$$\omega \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix}$$

where X_c , Y_c and Z_c are the camera co-ordinates (obtained by multiplication of the extrinsic matrix and the world co-or dinates).

F is equal to $\sqrt{3}$ and C_x and C_y are equal to 0.

- Intrinsic matrix is used to go from the camera co-ordinates to the image plane co-ordinates.
- Now, we will multiply this intrinsic matrix to all the camera co-ordinates of the 200 × 200 pixels of all the 3 images.
- After multiplication, we get x, y and z. But we are now trying to go back to 2 dimensions and thus we want to get rid of z.
- Thus, we multiple x and y by w which is equal to 1/z. So now, we just have x and y (z is now equal to 1) for every pair of X_c, Y_c and Z_c.
- After obtaining the new x and y of all the three images, we can say that we have reached back to the Cartesian domain.
- The last step is to go back to the image co-ordinates where we can store the pixel information that is present at x and y.
- To do this, we first calculate the minimum and maximum values of x and y of all the 3 images combined.

- Thus the range of x is from min_x to max_x and range of y is from min_y to max_y.
- Now, we need to set the resolution i.e. the pixel density of our new image.
- Let the dimensions of this new image be 300 × 300.
- We can set the new range of x and y from 20 to 280.
- Again by using rescaling, we change the values of x and y so that they now lie in the new range.
- Now, these new values are nothing but u and v co-ordinates of the pixels.
- Thus, once we are into the u-v plane, we simply go to that particular location and place its corresponding RGB content at that location.
- In short, all the 3 images will now be seen in a single image and thus the u-v locations of the 3 images together are the pixel co-ordinates in this new image of 300 × 300.
- We already have the RGB content of the 3 images and we will now place that content at its corresponding location in the new image.
- Thus we have captured the 3D cube in our camera and have obtained the 2D image of this scene.
- This approach has been implemented using C++.

1.2.3 Results

The 2D image of the 3D cube that gets captured by the camera is as shown below:

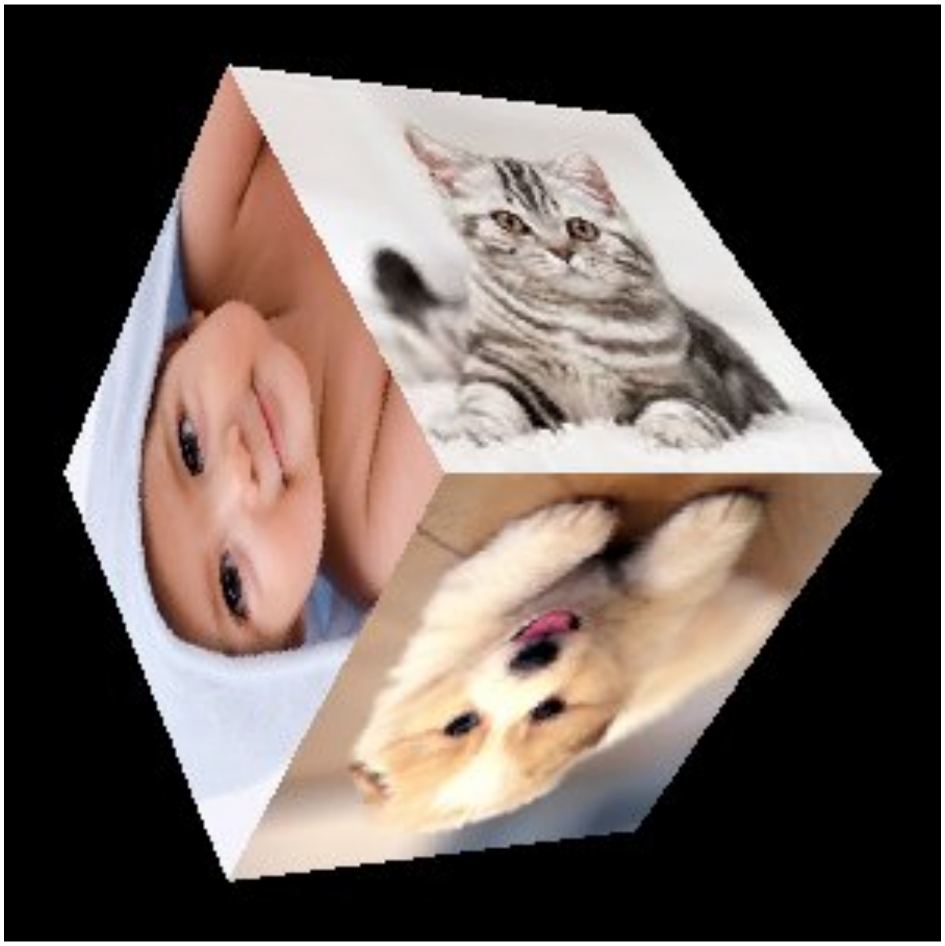


Figure 1.4 2D image of the 3D cube captured by the camera

1.2.4 Discussion

We have set the resolution of this image to 300 × 300. Moreover, we have filled in pixels in between 20 and 280 along u and v direction. We can see the 3 sides of the cube very clearly. Now, if we increase the pixel density by increasing the resolution to 800 × 800 and changing the new range of x and y to 100 to 700, we get the following image:

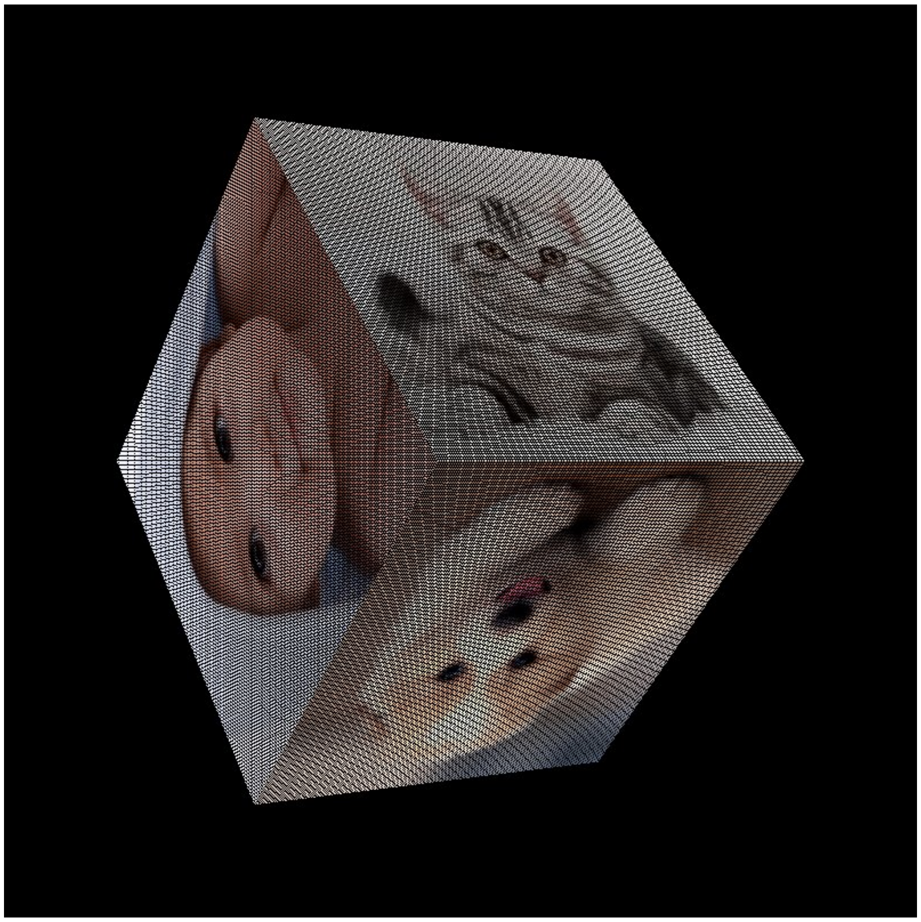


Figure 1.5 2D image of 3D cube with increased range

Thus, we can see that image contains curvy lines passing through them and also it is darker than before. Thus, we got some artifacts in this increased dimension image. Thus, the pixel density and hence the resolution of the image need to be chosen properly. The parameter that was chosen to be the best is the range of 20 to 280 i.e. all pixel lying between this range. This is because when the range was increase to 100 to 700, we got some undesirable artifacts in the image. Thus, in order to get a clear image with no artifacts, we simply need to use the proper pixel density value.

2. Problem #2: Digital Halftoning

2.1 Dithering

2.1.1 Motivation

When viewing an image on screen, we can use a large range of colors or gray levels but when it comes to printing an image on paper, we need to use only a limited variety of colored ink. So, in short, the actual colors are represented on paper by using a small set of colors. We do not see much of a difference between the image on paper and that on screen because the human eye does not really come to know about the optical illusion. Using halftoning basically creates this illusion. In this problem, we will perform halftoning using dithering which is nothing but converting a grayscale image to a black and white image in such a way that the different densities of black dots look like different gray levels.

2.1.2 Approach and Implementation

- We have a gray scale image that has pixel intensities ranging from 0 to 255.
- Our aim is to be able to represent this image using just two intensities of 0 and 255, without much loss of image content.
- We will perform dithering with the help of a Bayer's index matrix that indicates how likely a dot will be turned on.
- We will use two index matrices of 4 × 4 and 8 × 8.
- The formula for creating an index matrix is as follows:

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

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

- By using the above we can get $I_4(i,j)\ and\ I_8(i,j)$.

- Now, we will first normalize our image so that the pixel intensities now lie between 0 and 1 instead of 0 and 255. Dividing every value by 255 can do this.
- By using the index matrices $I_4(i,j)$ and $I_8(i,j)$ we now need to generate threshold matrices of the same dimensions respectively.
- This can be done by using the formula given below:

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

- We will get two separate threshold matrices with N = 4 (I_4) and N = 8 (I_8) respectively.
- Now, every normalized pixel value will be compared with the threshold value so that it gets assigned a value of 1 or 0 depending on the result of comparison.
- The same threshold value is repeated throughout the image and the comparison is done as follows:

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

  Where F(i,j) is the current normalized pixel value and G(i,j) is the new halftoned value.
- Thus, depending upon whether the normalized pixel value is less than or greater than the threshold, we assign 0 or 1 respectively.
- In order to visualize the image, we simply change 1 to 255 and keep 0 valued pixels unchanged.
- In this way, we have converted the gray colored image to a black and white image with just 2 levels i.e. 0 and 255.
- Now, let us try to repeat the above process in order to convert the image into four gray levels of 0, 85, 170 and 255.
- So, basically instead of 2 levels we want 4 levels to represent the original gray level image with 256 intensity levels.
- The process is same as above with a few small changes.
- We will use the same 2 threshold matrices that were obtained above but now our comparison process will be different than before.
- Now, we will compare F(i,j) with three different thresholds and assign values accordingly as follows:

G(i,j) =

$$\begin{cases} 0 \; if \; F(i,j) < 0.5 \times T\;(i\; modN, j\; modN) \\ 85 \; if \; F(i,j) \geq 0.5 \times T\;(i\; modN, j\; modN) \; and \; F(i,j) < T\;(i\; modN, j\; modN) \\ 170 \; if \; F(i,j) \geq \; T\;(i\; modN, j\; modN) \; and \; F(i,j) < 1.5 \times T\;(i\; modN, j\; modN) \\ 255 \; otherwise \end{cases}$$

- Thus, we are now comparing our normalized pixel values with three thresholds and depending upon the range in which they lie, we assign them one of the four values.
- So, this is how we can convert our original image with 256 levels into a halftoned image with 4 intensity levels.
- The above implementation has been done in C++.

2.1.3 Results

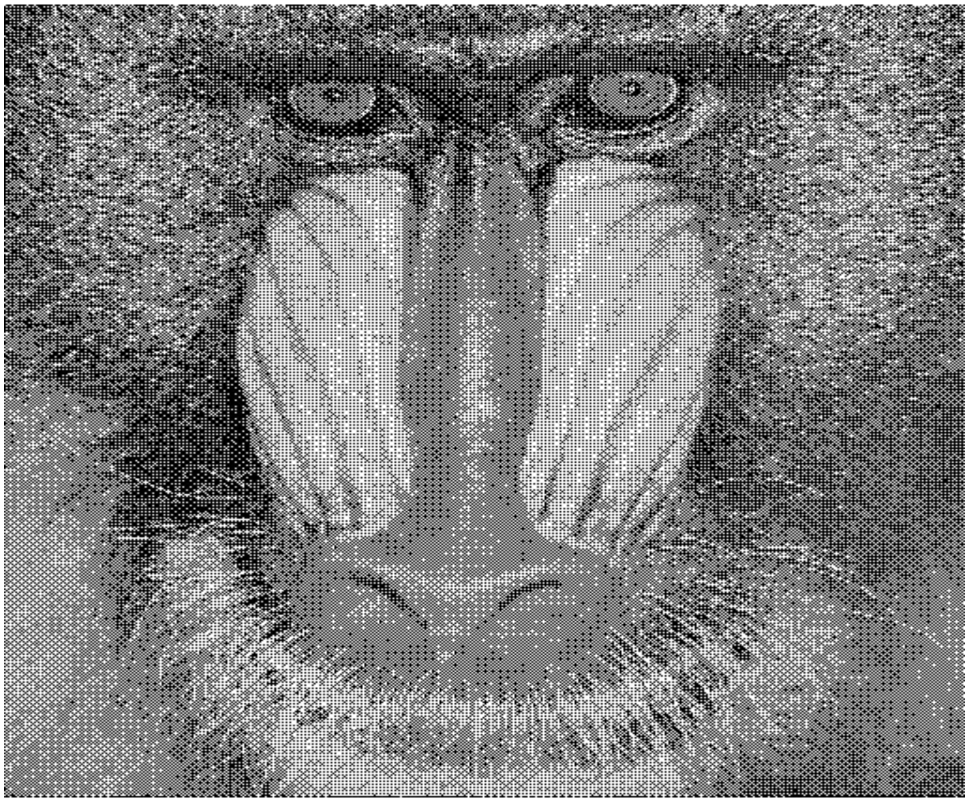


Figure 2.1(a) Dithering with N = 4 (I4)

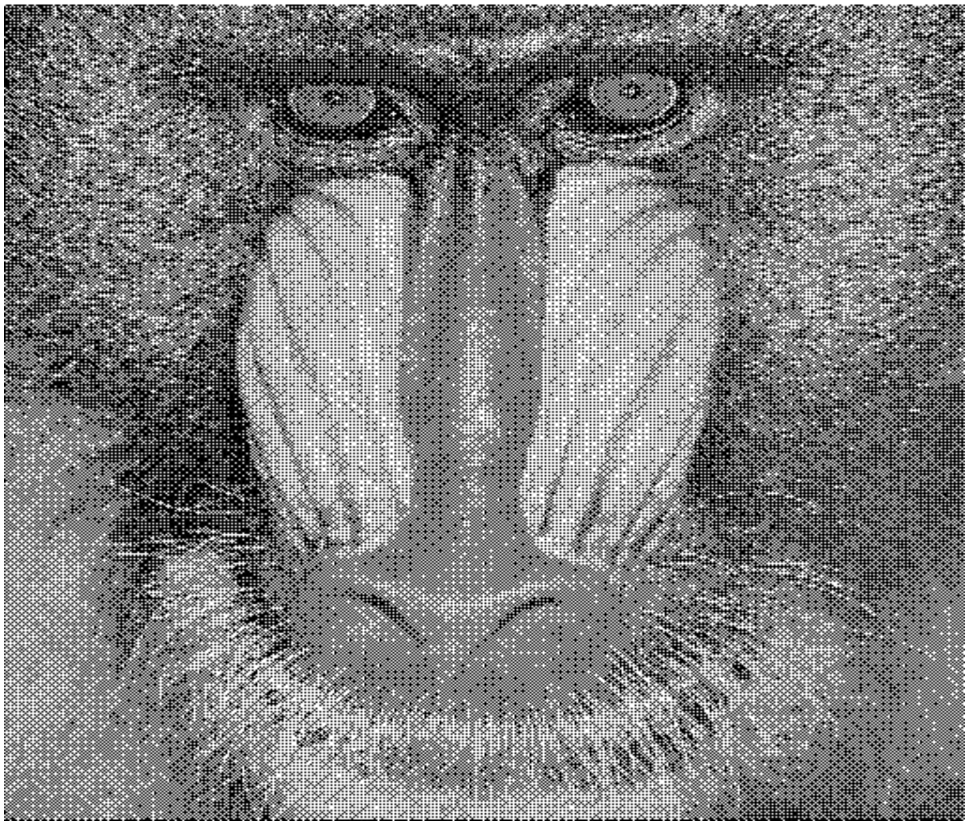


Figure 2.1(b) Dithering with N = 8 (I8)

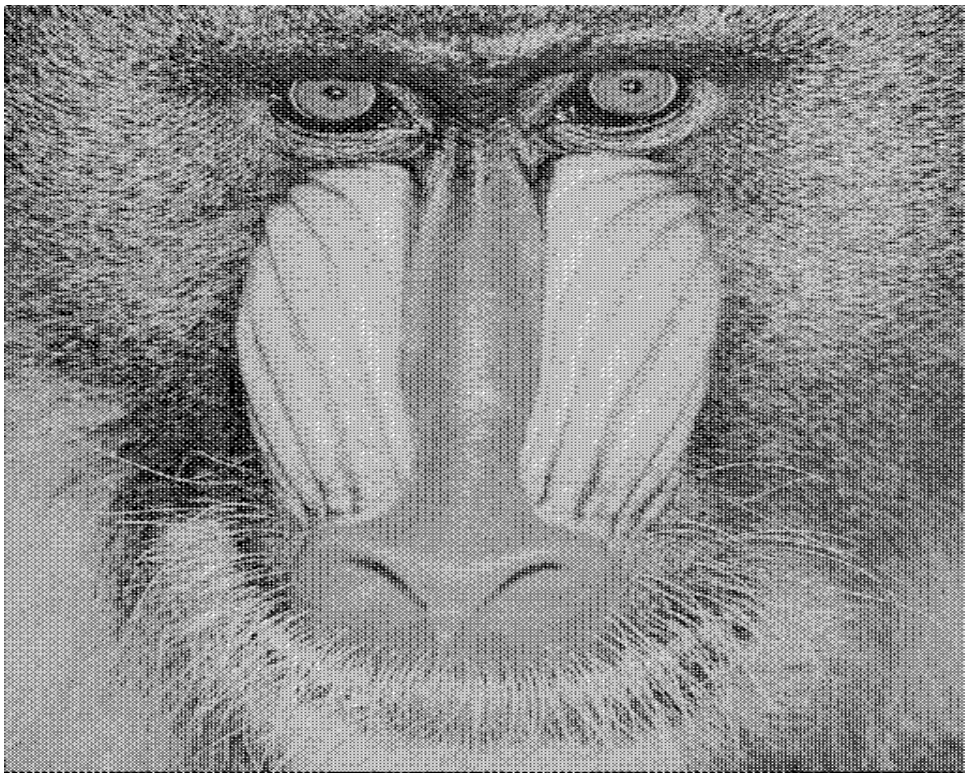


Figure 2.1(c) Dithering with 4 levels and N = 4 (I4)

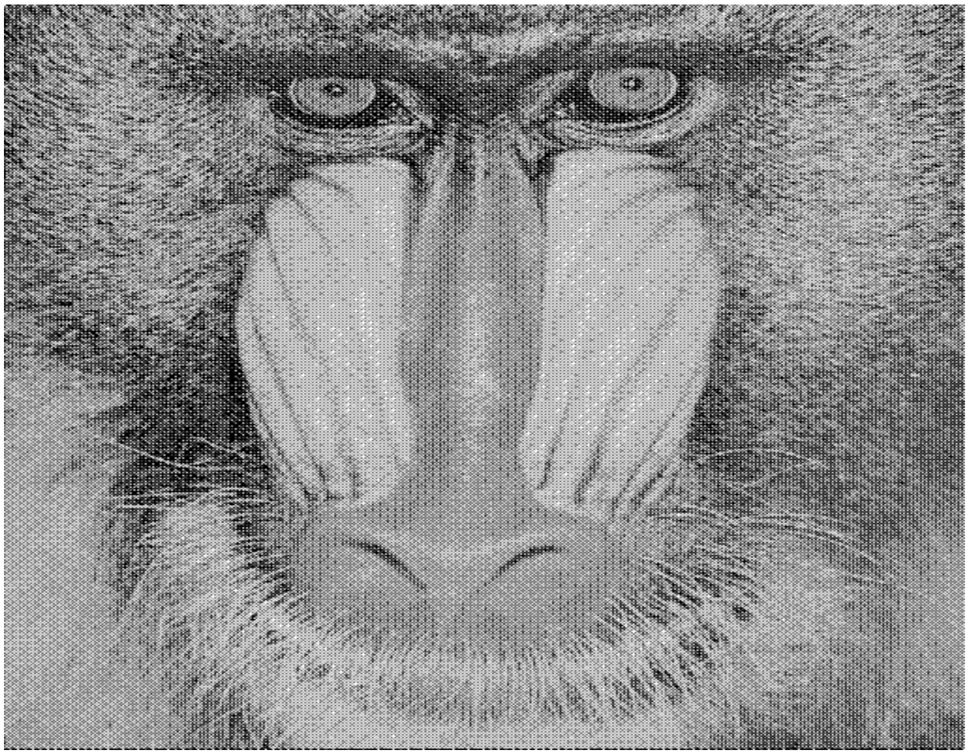


Figure 2.1(d) Dithering with 4 levels and N = 8 (I8)

2.1.4 Discussion

The above results show how using a reduced gray color set with 2 intensity levels and 4 intensity levels can represent the image. We can observe that the images look quite similar for both the index matrices as we cannot find considerable difference between the images in Figure 2.1(a) and Figure 2.1(b). Now, after we convert the original image into an image with 4 intensity levels we can see that the number of black pixels is now lesser than that in the image with 2 intensity levels. Also, the images look smoother. Thus, we can conclude that dithering can be used to perform halftoning on a gray scale image.

2.2 Error Diffusion

2.2.1 Motivation

Error diffusion is another type of halftoning that calculates error of quantization and then distributes this error to the neighboring pixels. Just like dithering, this method is used to convert an image with multiple intensity levels into a binary image. In this problem, we will perform error diffusion with the help of 3 different error diffusion matrices namely Floyd-Steinberg's error diffusion, Jarvis, Judice and Ninke error diffusion and Stucki error diffusion.

2.2.2 Approach and Implementation

- The original image is an 8 bits gray scale image with intensity values between 0 and 255.
- We will first normalize the pixel values so that they lie between 0 and 1. Dividing the pixel intensity by 255 does this.
- Normally, we traverse the matrix row wise and from top to bottom. But in this method we will use a different scanning method.
- We start from the top left pixel of the image of the first row and keep moving right till we reach the last column of the first row.
- Now, for the second row we will move from this last column to left towards the first column and so on.
- In programming convention, the row and column go from 0 up to dimension – 1. Thus, we can say that if the row number is even, traverse from left to right whereas if the row number is odd, traverse from right to left.
- This is how we will scan the original matrix.
- Now, again we will use thresholding to assign 2 levels of 0 or 1.
- The threshold is 0.5, meaning that if the pixel value is greater than 0.5 then it will be assigned 1, otherwise it is assigned a 0.
- Now, when we perform this type of quantization, we are meant to get the quantization error which indicates how much different is the new value from the old one.

- For e.g. if the pixel was 0.75 before thresholding and became 1 afterwards (because 0.75 > 0.5), then it gave a quantization error of 0.25.
- In error diffusion, this quantization error affects the nearby pixels because we diffuse this error to them.
- We are going to consider 3 different matrices for error diffusion. They are as shown below:

Floyd Steinberg's matrix:

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

Jarvis, Judice and Ninke (JJN) matrix:

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

Stucki matrix:

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

- Now, after we get the quantization error for the current pixel, we diffuse this error to its future pixels by using one of the three matrices above.
- Basically, we place the center of the error diffusion matrix at the current pixel.

- After using thresholding to assign a new value (either 0 or 1), we get the quantization error. Depending upon the location of non-zero elements of the diffusion matrix, the corresponding neighboring pixels are considered for diffusion.
- So if we are using the Floyd Steinberg's matrix, 4 pixels will be considered for diffusing the error of the current pixel, namely the pixel to the right, the pixel at the bottom-left, the pixel at the bottom and the pixel at the bottom-right of the current pixel.
- So while diffusing the error to the pixel at the right of the current pixel, the error value is multiplied with the corresponding weight given in the matrix, which in this case is 7/16, and this is added to the actual intensity value of that pixel.
- Thus, the changes that are made at a certain pixel affect the neighboring pixels.
- The pixels considered for diffusion are always the future pixels, meaning that their values are yet to be thresholded.
- We have used these 3 matrices for error diffusion.
- JJN and Stucki matrices consider 12 neighboring pixels to diffuse error whereas Floyd-Steinberg's matrix considers 4 neighbors.
- Thus, the technique is very simple to implement as we are just thresholding the pixels, calculating the quantization error and diffusing this error to the appropriate pixels.
- After we perform error diffusion using these three matrices separately, we will get 3 halftoned binary images.
- This approach has been implemented in C++.

2.2.3 Results

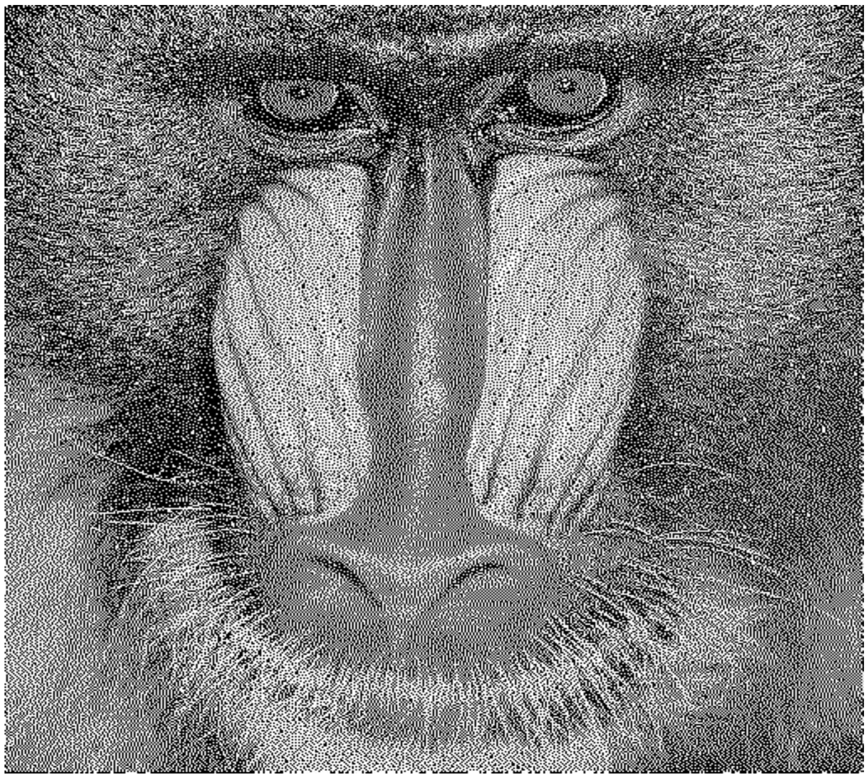


Figure 2.2(a) Floyd-Steinberg's error diffusion

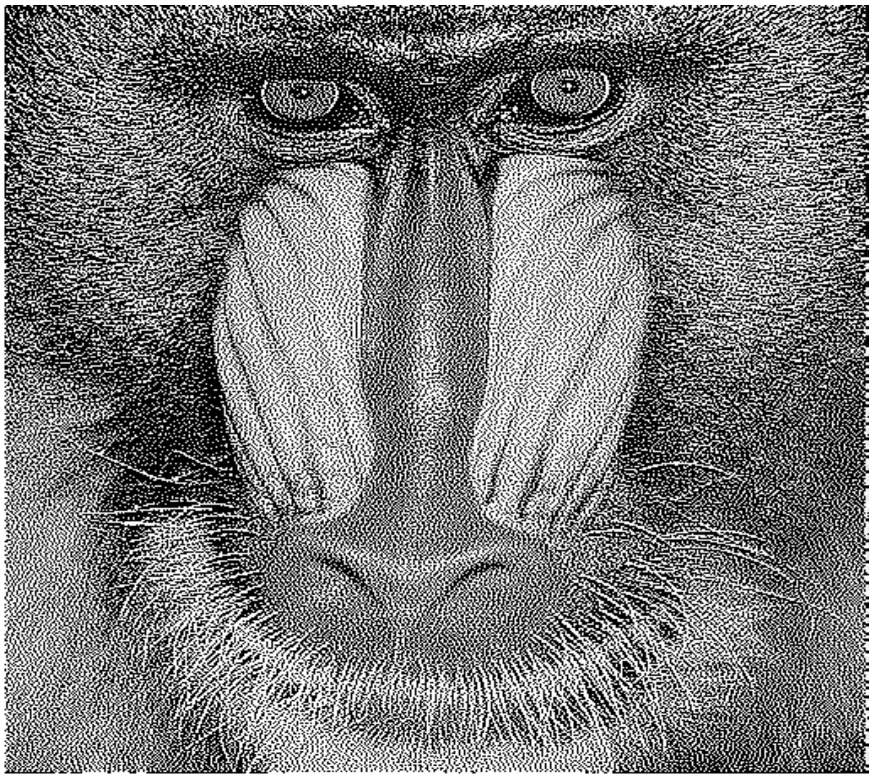


Figure 2.2(b) Jarvis, Judice and Ninke error diffusion

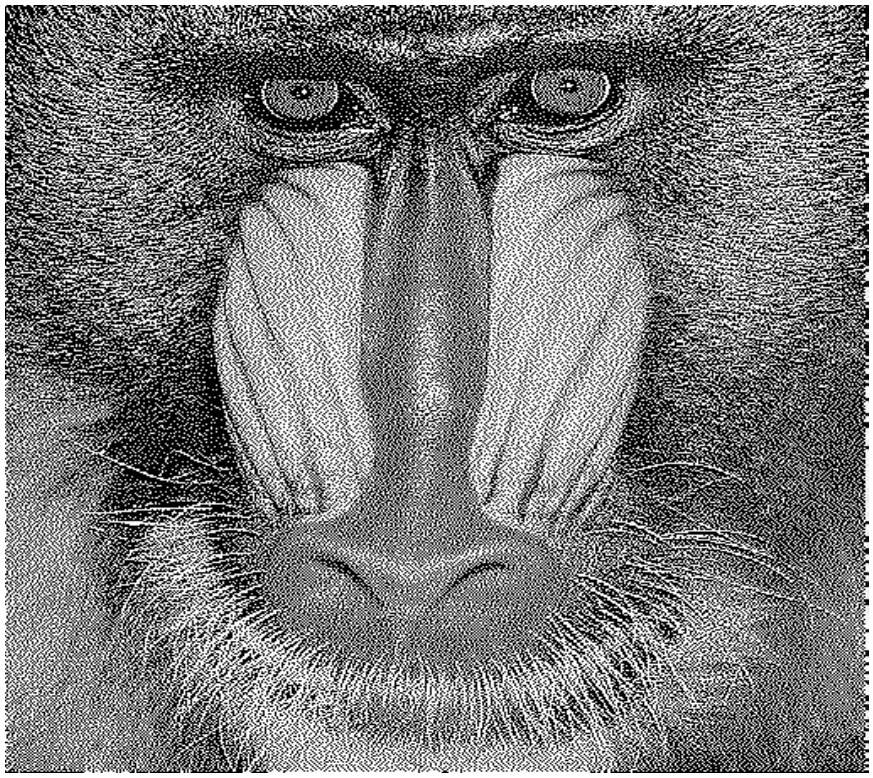


Figure 2.2(c) Stucki error diffusion

2.2.4 Discussion

From the above results, we can see that the images in Figure 2.2(b) and Figure 2.2(c) look better than the image in figure 2.2(a). Thus, JJN and Stucki error diffusion matrices give better results than the Floyd-Steinberg's error diffusion. This is because they consider a larger neighborhood matrix for diffusing error around the current pixel. Moreover, Stucki gives the best halftoned image amongst the three methods. We can observe the image quality in Figure 2.2(c). The image appears to be smooth and the noise content is minimal. We have looked at three different error diffusion methods that result into a binary halftoned image. We can see that all three images are not very uniform or completely noise free. We can also use other dithering methods like Sierra dithering or Burkes dithering to get slightly improved reults.

2.3 Scalar Color Halftoning

2.3.1 Motivation

Halftoning is mainly done to reduce the large color set of an image to a finite color set in order to represent the image on paper. In the above problem, we implemented halftoning on a gray scale image and converted it into a binary image with two intensity values of 0 and 255. Now, we can extend the same procedure for a color image of 24 bits. We can simply apply the error diffusion matrices on the three channels (Red, Green and Blue) separately and then combine them to get a halftoned color image. In this problem, we will use the Floyd-Steinberg's error diffusion for halftoning.

2.3.2 Approach and Implementation

- To get started, we first need to go from RGB domain to CMY domain.
- Cyan is the complementary color to Red, Magenta is the complementary color to Green and Yellow is complementary to Blue.
- Thus, to get cyan from red, we just subtract the red intensity value from 255. Similarly we obtain magenta and yellow from green and blue respectively.
- Now, the error diffusion matrix that we are going to use is shown below:

  Floyd Steinberg's matrix:

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

- As we want to implement Scalar halftoning, we will perform error diffusion on the cyan, magenta and yellow channels separately.

- Thus, we can just store the three channels of our color image into three separate matrices and then perform the error diffusion on them.
- As we learnt earlier, we just place the center of Floyd-Steinberg's error diffusion matrix at our current pixel, and we continue doing this for all the pixels of the image.
- Now, at the current pixel, we again compare the pixel intensity with the threshold value. In this case, the threshold is 128.
- If the pixel value is less than 128, then it is assigned a zero value whereas if it is greater than 128 then it is changed to 255.
- Every time we quantize a pixel, we will get a quantization error that is nothing but the difference between the old value and the new value of the pixel.
- With the help of the diffusion matrix, we diffuse this error to the future pixels of the original image that has not been thresholded yet.
- The details of this procedure have already been explained in the previous problem. Also the method of scanning the image has been described earlier. This same method needs to be used for the 3 matrixes of Cyan, Magenta and Yellow.
- We repeat this process thrice i.e. for the entire matrix of every channel separately.
- Thus, we have converted the Cyan, Magenta and Yellow channels into binary images with intensity values of 0 or 255.
- Now, we will convert CMY to RGB by subtracting pixels from 255 and then we will combine the three matrices together to form a color halftoned image.
- We have now converted our original 24 bits RGB image into a halftoned color image where each channel has only 2 intensity levels instead of 256.
- This approach has been implemented in C++.

2.3.3 Results

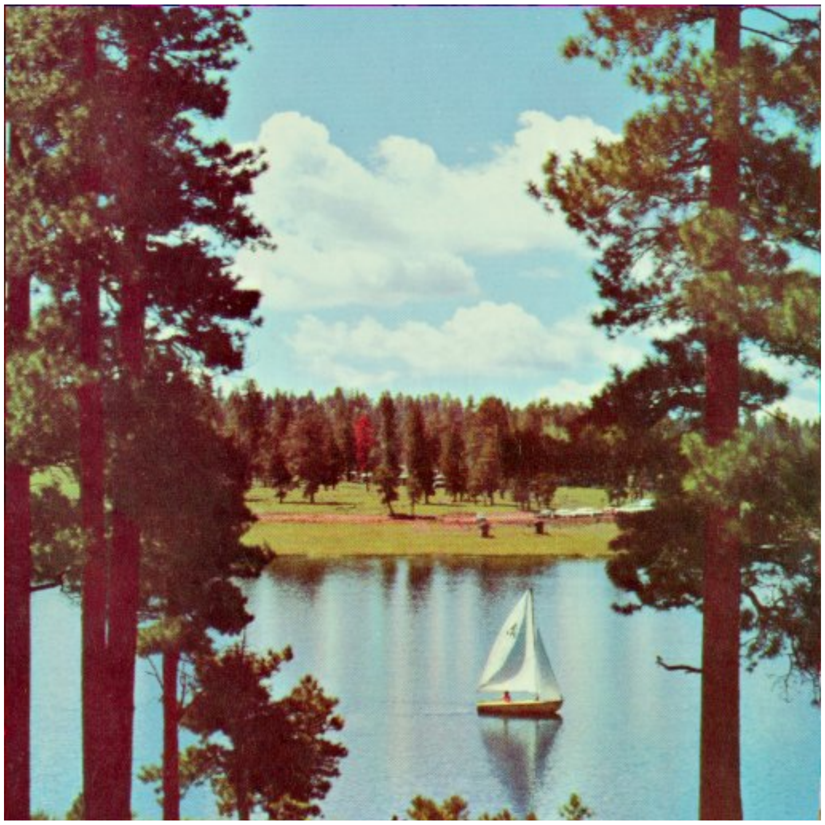
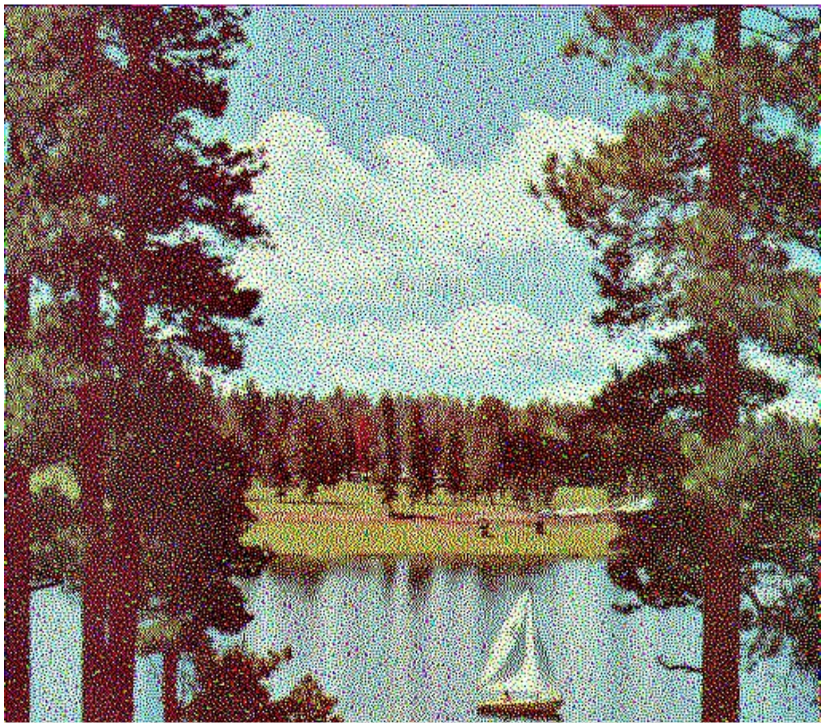


Figure 2.3 (a) Original color image



2.3.4 Discussion

We can see from the above results that it is possible to represent a color image with a reduced color set. The original image had 8 bits each for red, green and blue whereas the halftoned image has just a single bit each for red, green and blue. In spite of using a very small set of colors, we have still retained the dominant colors of the image. But, if we observe closely, we can see some irregularities in the halftoned image. The image looks noisy and it has a lot of dots spread all over the image. This is the main drawback of this technique. It results into a noisy image that is not smooth.

2.4 Vector Color Halftoning

2.4.1 Motivation

In the above problem, we learnt how to implement the scalar color halftoning. We simply performed the error diffusion halftoning on the three channels separately and the combined the results to get the halftoned color image. The halftoned image obtained by this process was not very good as it contained a lot of noise. In this problem, we will again perform error diffusion but on the three channels together. We will use MBVQ quantization and then do error diffusion by using the joint vector of the three channels.

2.4.2 Approach and Implementation

- We first convert the RGB image into CMY by subtracting all the pixel values from 255 for all the three channels.
- We then perform thresholding i.e. set the pixel value to 255 (i.e. 1) if it is greater than 128 and to 0 (i.e. 0) if its less than 128.
- From this point onwards, we will consider the color vector of each pixel (i.e. the C, M, Y intensities).
- We will traverse the image from left to right for even rows and from right to left for odd rows, just like we learnt in the earlier problems.
- Also, we will store the accumulated error of each pixel in a separate matrix (that is of the same dimensions as that of the image) at the corresponding pixel location.
- The accumulated error for the first pixel will be (0,0,0) and will go on increasing as we traverse through the image.
- The cube that we will follow to get MBVQ is shown in Figure 2.4.

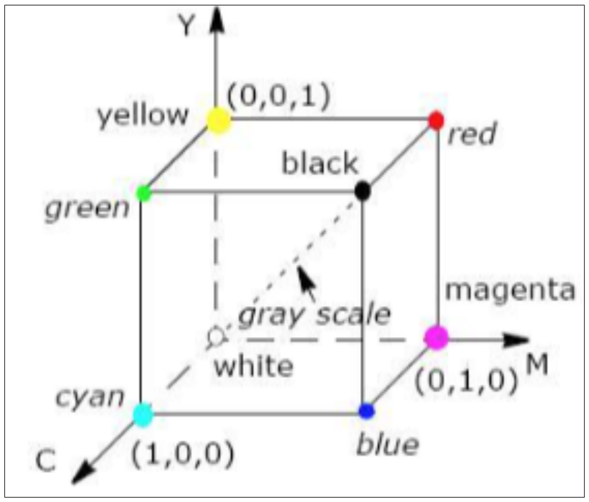


Figure 2.4 The CMY cube

- Here, C is (1,0,0), M is (0,1,0), Y is (0,0,1), R is (0,1,1), G is (1,0,1), B is (1,1,0), K is (1,1,1) and W is (0,0,0).
- For each pixel i.e. the C, M, Y vector, we get the MBVQ quadruple that is nothing but the set of vertices of the tetrahedron in which the color vector is located.
- This is done by using the following algorithm given in [2]

```
Pyramid MBVQ (BYTE C, BYTE M, BYTE Y)
{
    if ((C + M) > 255)
        if ((M + Y) > 255)
            if ((C + M + Y) > 510)          return RGBK;
            else                            return GBMR;
        else                                return CMGB;
    else
        if (!((M + Y) > 255))
            if ( !((C + M + Y) > 255))      return WCMY;
            else                            return CMYG;
        else                                return RGMY;
}
```

- Thus, depending upon the intensity values of C, M and Y of each pixel, we will get the MBVQ tetrahedron for that pixel.
- After we get the MBVQ tetrahedron, we need to find the vertex of this tetrahedron that is closest to the sum of our pixel and its accumulated error.

- The accumulated error of that pixel is already stored in the error matrix.
- The closest vertex is found out by using the Euclidian distance between each of the four vertices and the pixel with error.
- Once we get the closest vertex, this vertex's intensity values are now assigned to the current pixel of the image.
- We now compute the quantization error of this pixel as:
  Quantization error = CMY(i,j) + error(i,j) – vertex
- This quantization error is now diffused to the future pixels by using the Floyd-Steinberg's error diffusion matrix.
- This process is continued till we finish traversing all the pixels in the image.
- Thus, we go on replacing the pixel values with its closest vertex intensities.
- In the end, to get the final image, we go back to RGB by subtracting the pixel values from 255.
- The quantization error that we obtain for each pixel is actually diffused to the error buckets (in our matrix for error) of the future pixels.
- We can obtain the vector color halftoned image by following this procedure.
- This approach has been implemented in C++.

2.4.3 Results

The image for vector color halftoning is shown below:

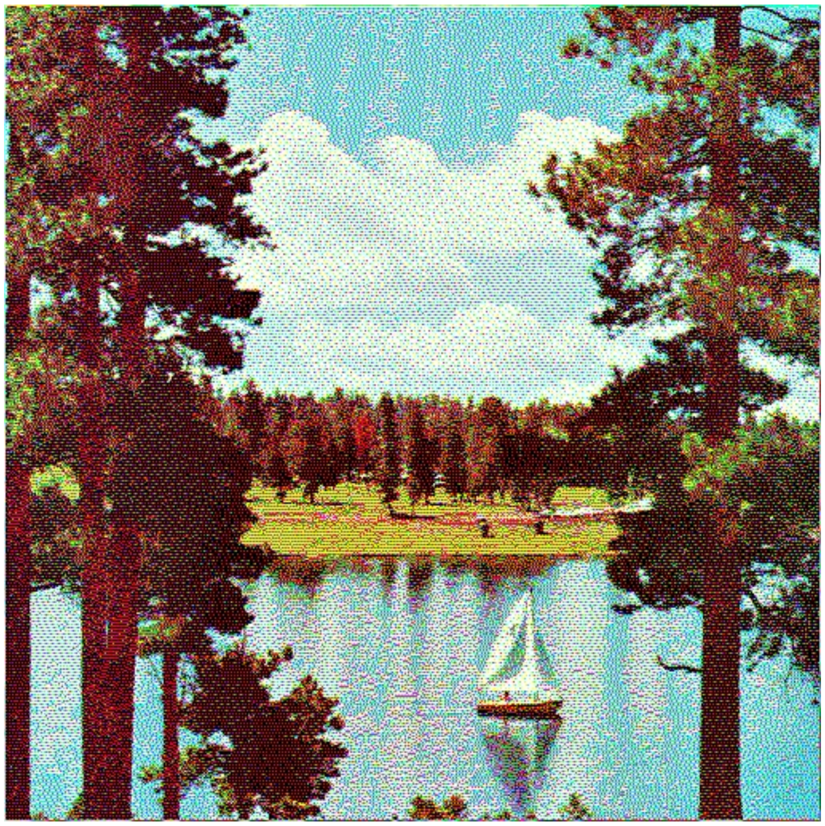


Figure 2.5 Vector color halftoned image

2.4.4 Discussion

In this method, we consider the three channels together instead of treating them separately. Moreover, we use MBVQ tetrahedron of every pixel and then change the value of that pixel to its nearest vertex, by inclusing its accumulated error. By comparing the images in Figure 2.5 and Figure 2.3(b), we can see that the vector color halfoning gives a better image. We canconclude this because the color quality of the vector halftoned is much better than the scalar halftoned image. Moreover, the noise is also very less and the image looks smoother. Thus, we can say that vector color halftoning is better than scalar color halftoning.

3. Problem #3: Morphological Processing

3.1 Shrinking

3.1.1 Motivation

Shrinking is a type of morphological operation that is mainly performed to count the number of objects in the image. In this problem we will perform shrinking on the horseshoe image to count the number of nails and holes in the image. Moreover, we will also count all the white objects in our image by removing the black holes in the horseshoe.

3.1.2 Approach and Implementation

- To perform shrinking on the horseshoe image we are going to use the conditional and unconditional mark patterns for shrinking given in [1].
- Shrinking is performed in two stages.
- In stage one, the pixels that are likely to get deleted are marked as 1 by using the conditional mark patterns. The new matrix is M.
- Now, this matrix M is passed on to stage 2 where only those values that satisfy the unconditional mark patterns are kept unchanged and the others are equated to zero.

- The following steps describe the shrinking process:

  Step 1: Take the image in matrix named 'I' and convert it into a binary valued image containing 1 for 255 and 0 for 0.
  Step 2: Create a matrix of the same dimensions as that of the image and name it 'S1' and equate it to zero.
  Step 3: Consider the 3 × 3 neighborhood of the current pixel in I.
  Step 4: Calculate the bond value of that pixel by using 8-connectivity.
  Step 5: Depending upon the bond value, apply the corresponding conditional mark patterns on the current pixel and its neighborhood of the Image I.
  Step 6: If the neighborhood is equal to the pattern, then set that corresponding pixel in S1 to 1.
  Step 7: Do this till we reach the end of the image. We now have a matrix S1 that contains the locations of pixels that are likely to get deleted
  Step 8: Now, create another matrix of the same dimensions as that of the image and name it S2 and equate it to 1 − S1.
  Step 9: Apply unconditional mark patters on the current pixel and its 3 × 3 neighborhood in the S1 matrix.
  Step 10: If the neighborhood and the pattern match, then equate the corresponding value in S2 to 1.
  Step 11: If there is no match, then equate that value in S2 to 0.
  Step 12: We have now marked all the pixels that we want to delete by 0 in S2.
  Step 13: Create a copy of I in a matrix named 'copy'
  Step 14: Multiply S2 with I and now this is the new I. Thus, I = S2 × I.
  Step 15: Check if I is equal to copy.
  Step 16: If not they aren't the same, then using the new I, repeat steps 2 to 15.
  Step 17: If they are the same, we conclude that we have reached convergence and the new I is the final shrunk image.

- After getting the shrunk image, we now need to count the number of nails and holes in the image.
- To count the number of nails, we simply traverse through the image to find white pixels with all the 8-connected neighboring pixels equal to 0. The number of nails in the shrunk image is equal to 6.

- To count the number of holes, we label the black pixels that are inside the circular regions at the top of the horseshoe image. We need to assign 3 unique labels to each of these regions. And this can be achieved by using segment labeling in our image.
- Thus, the number of holes in the shrunk horseshoe image is 3.
- Now, we also want to calculate the number of white objects in the original horseshoe image.
- We first want to fill the holes of the horseshoe image.
- To do this, we just find the black pixels that are 8-connecetd to all white pixels, and then equate that back pixel to 255.
- After we fill in the holes, we now need to perform shrinking on this hole-filled image.
- Thus, we can get the shrunk image by performing the steps given above.
- Now, we can again count the nails as we did earlier.
- To label the horseshoe boundary, we need to track the white pixels and then consider them to form one single object.
- Tracking the white pixels by using iterations can do this.
- Thus, the number of white objects in the shrunk image is 7.

3.1.3 Results

The original horseshoe image is as shown below:

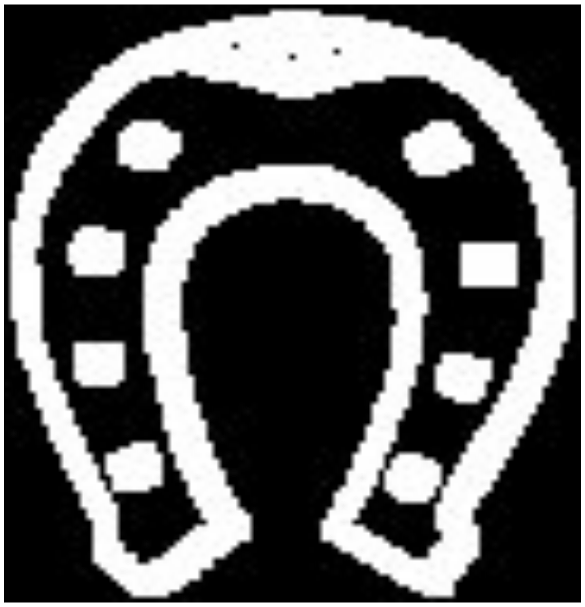


Figure 3.1 (a) The original Horseshoe image

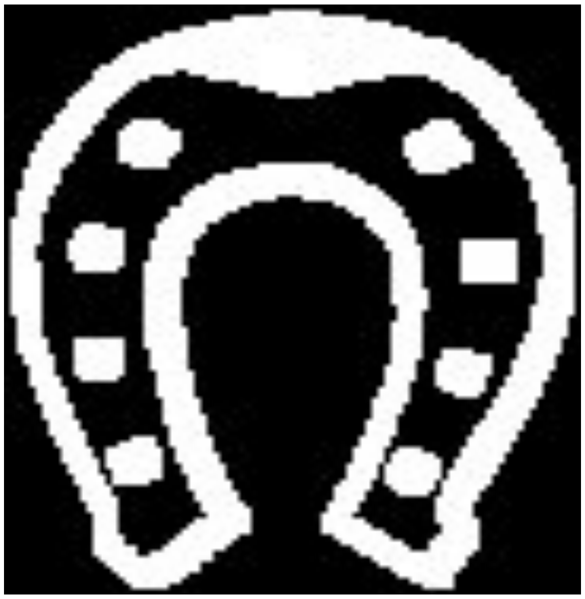


Figure 3.1 (b) The hole-filled horseshoe image

The images obtained after shrinking the original horseshoe image and the hole-filled horseshoe image are shown below:

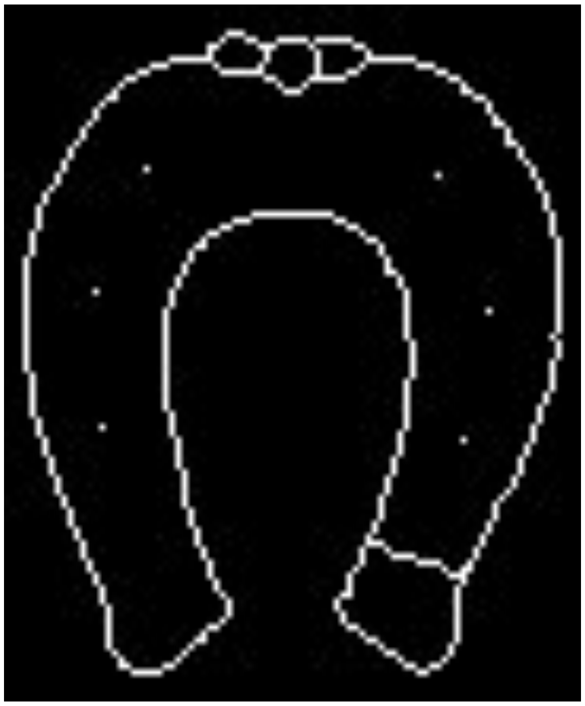


Figure 3.1 (c) Shrinking of the original horseshoe image

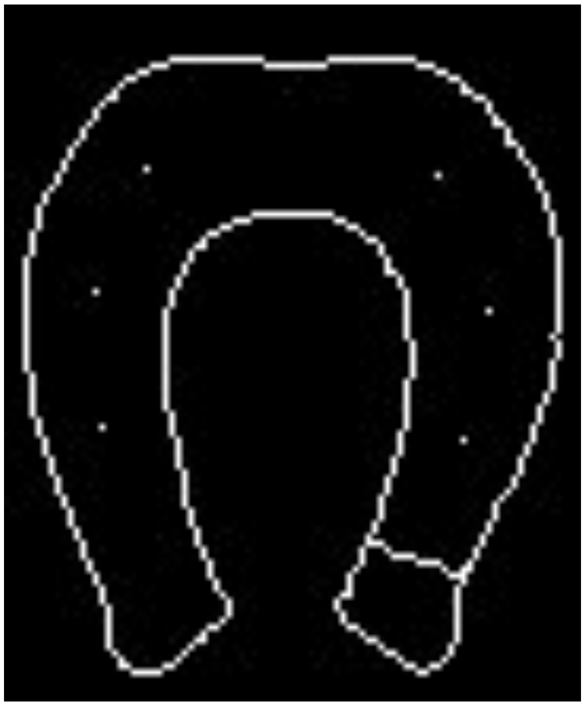


Figure 3.1(d) Shrinking of the hole-filled Horseshoe image

3.1.4 Discussion

If we try to count the number of nails and holes manually, then we would say that there are total 8 nails and 3 holes. Similarly, we would say that there are total 9 white objects in the image. But, if we observe the original image closely, then we can see that the bottom-right nail is connected to the horseshoe at two ends whereas the bottom-left nail is connected to the horseshoe at one end. Thus, the bottom-left nail is completely shrunk whereas the bottom-right nail gets shrunk to a line. Thus, now if we count the number of nails, we get 6 and for white objects we get 7. The number of holes we get is 3. Thus, the program gives different counts, of nails and white objects, than we would expect from manual counting.

3.2 Thinning and Skeletonizing

3.2.1 Motivation

Morphological processing is done for the analysis of the geometrical structures of digital images. The main idea is to compare the binary image elements with different pre-defined shapes and then extract some features from the image. In this problem, we will implement thinning and skeletonizing on a binary image of a horse.

3.2.2 Approach and Implementation

- This process also has two stages of operation just like we saw in the previous problem.
- First, we will assign 1 to pixels that are white (i.e.255) and 0 to pixels that are black (0 already).
- Now, we need to pass this binary image to stage 1 of thinning.
- But if we observe the original image, we can see some holes in the horse. We do not want them as they will not result into the true thinned horse.
- Moreover, we can see that the horse's edges are not smooth so wee also need to do boundary smoothing.
- Thus, as pre-processing, we first need to apply hole-filling and boundary smoothing to the original horse image.

3.2.2.1 Pre-processing

3.2.2.1.1 Hole-filling

- When we want to shrink an image, we usually use an operation with 2 stages as we saw earlier.
- The first stage uses a set of conditional mark patterns that predict how likely a pixel will get deleted.

- If we pass our original horse image through this first stage, we will get the pixels that lie at the boundary of various regions.
- In short, passing the image through the shrinking conditional mark patterns result into a new image containing edges.
- Now, this edge image will have the true edges of the horse as well as edges of the holes inside the horse.
- The original horse image and the images containing all the edges are as shown below:

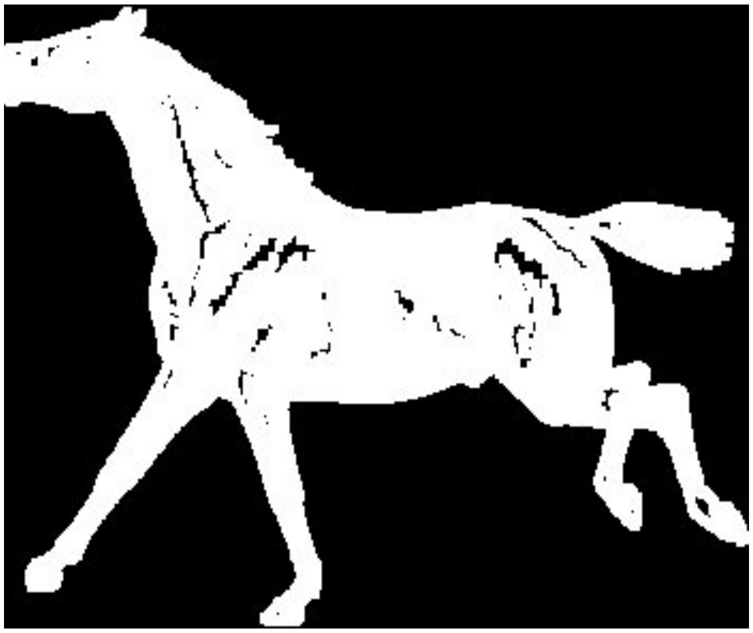


Figure 3.2 Original Horse image

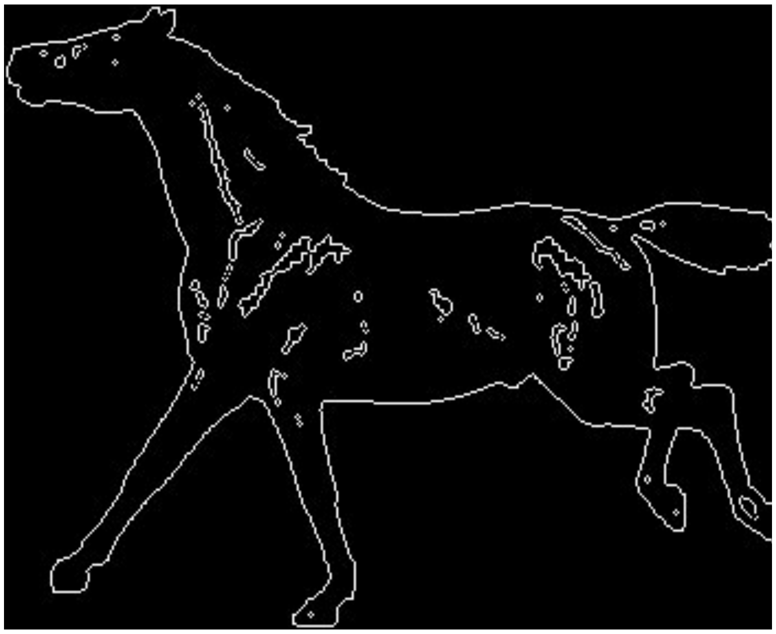


Figure 3.3 Edges of the original horse image

- Now, we want to determine only the true edge of the horse and get rid of the edges of the holes.
- This can be done by using the following steps:

  (<u>Note</u>: Every white pixel is 8-connected to only 2 white pixels in its 3 × 3 neighborhood and the boundary needs to be completely closed)

  Step 1: Create another image (T) that will store the true horse edge by creating a matrix of the same dimension as that of horse, and with all pixel values equal to zero.

Step 2: Traverse the edge image of the original horse (F) from the top left corner till you get the first white pixel and then store its row and column number

Step 3: Now, check the pixel to the right, bottom-right and bottom of this white pixel. If either of them is white, then change the current pixel in F to zero and in T to 1. The current pixel now becomes this connected white pixel and we need to store the new row and column number.

Step 4: Track the next white pixel connected to the current pixel by using 8-connectivity and then when a white pixel is found, just replace the current pixel in F by 0 and in T by 1 and then store the row and column number of the new current pixel.

Step 5: Repeat step 4 till we reach a pixel that does not have even a single white neighboring pixel (8-connected).

- Thus, by following these steps we have detected the true boundary of our horse image and this is as shown below:

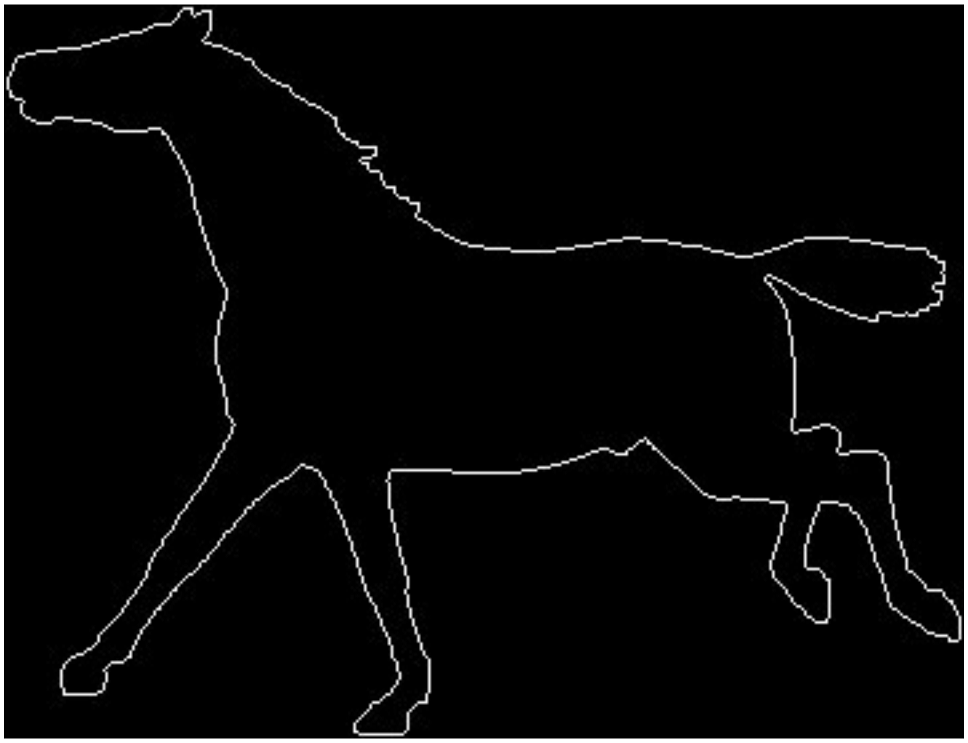
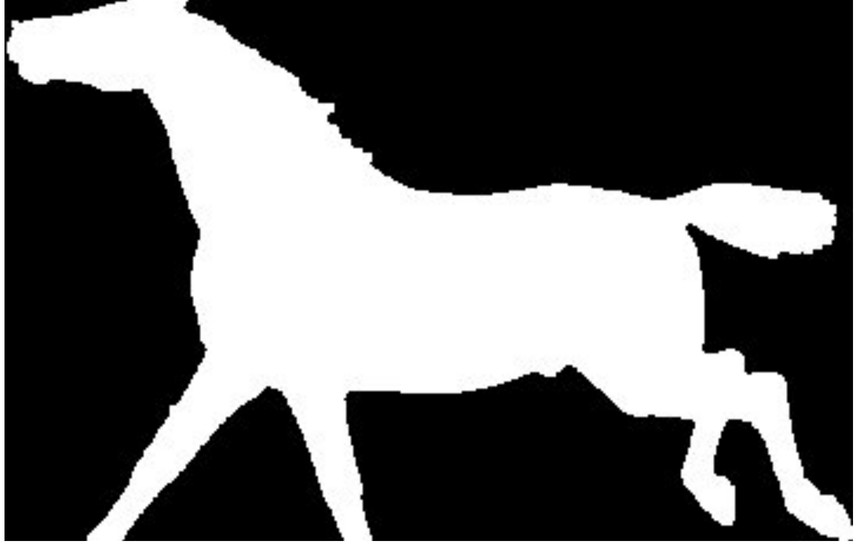


Figure 3.4 True edge of the Horse image

- Now, we have the original horse image (H) containing holes, the edge image (F) that has edges of horse as well as holes and the true edge image (T) containing only the true horse boundary.
- Now, we can compare the pixels in T and F and then make changes to H as follows:
    - ➢ For each white pixel in F, check if the corresponding pixel in T is also white. If it is white, then keep everything unchanged.
    - ➢ If it is not white, it means that it is the boundary of a hole.
    - ➢ Now, take a 3×3 neighborhood around this pixel in H and change all its 8-connected pixels to white.
    - ➢ Perform this for all the pixels that are white in F but black in T.
    - ➢ Thus, we are now finding out the holed pixels of the horse and filling the holes by replacing the black pixels by white pixels.
    - ➢ After we perform this, we get a horse image with less number of holes than before any holes.
    - ➢ Now, we again traverse H and after we come across a black pixel, we take the 5 × 5 neighborhood of this pixel and count the number of white pixels in this neighborhood.
    - ➢ If the number is greater than 15 then it indicates that it is a hole pixel in the horse and not a background pixel and then we change this black pixel to white.
    - ➢ We do this for the entire image and thus convert all the black pixels of the holes into white pixels of the horse.
    - ➢ We have thus performed hole-filling for the horse image.
- The hole-filled horse image is as follows:



3.2.2.1.2 Boundary Smoothing

- First applying dilation to the hole-filled image and then applying erosion to the dilated image can perform boundary smoothing.
- In dilation, we consider each black pixel and then convert it into white if at least one of its 8-connected neighbor is white.
- After dilating the entire image we perform erosion on this dilated image.
- In erosion, we consider each white pixel and convert it into black if at least one of its 8-connected neighbor is black.
- After we perform dilation and erosion on the hole-filled horse image, we get the final pre-processed image with smooth boundaries as shown below:

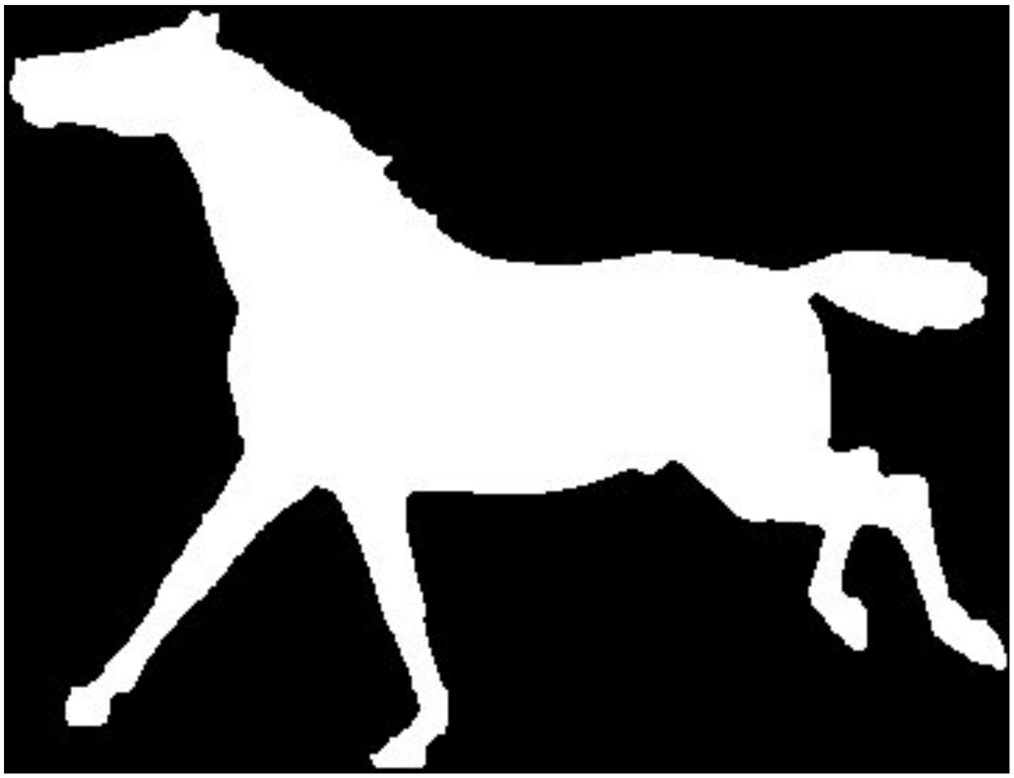


Figure 3.6 Pre-processed Horse image

3.2.2.2 Thinning

- Just like shrinking, thinning is a two-stage process.
- Thus, in stage 1, we will use all the conditional mark patterns for thinning given in [1].
- In stage 2, we will use all the unconditional mark patterns for thinning given in [1].
- To get started, the pre-processed image is used as an input to the first stage.
- The output of the first stage is a matrix (M) that marks all the boundary pixels likely to get deleted.
- This M is used as an input to the second stage.
- Now, with the help of unconditional mark patterns of thinning, only those pixels, that satisfy the criteria, are deleted from the image.
- This image is again sent to stage 1 and stage 2.
- Thus, the process is repeated till convergence i.e. till the image at the output of stage 2 is equal to the image at the input of stage 1.
- The obtained image is the thinned image of Horse.

3.2.2.3 Skeletonizing

- The same process described above is used for skeletonizing.
- The only difference is the conditional and unconditional mark patters.
- In this part, the skeletonizing conditional mark patterns and the skeletonizing unconditional mark patterns given in [1] are used in place of the thinning patterns that were used above.
- After convergence, the image that we get is the skeletonized horse Image.

3.2.3 Results

The above approach has been implemented in C++. The thinned and skeletonized images of Horse with pre-processing are as follows:

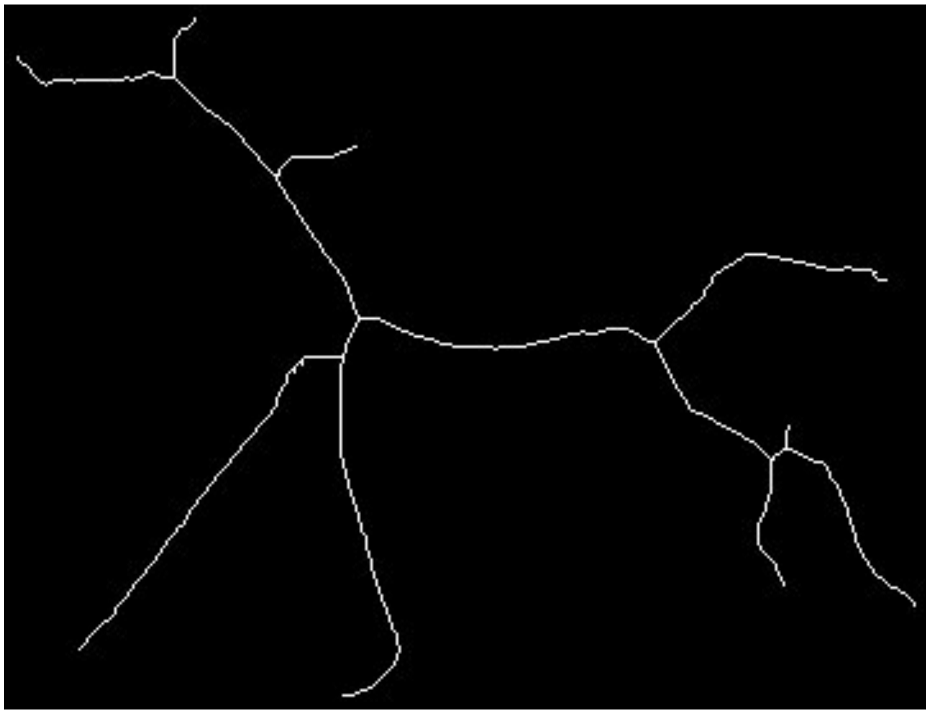
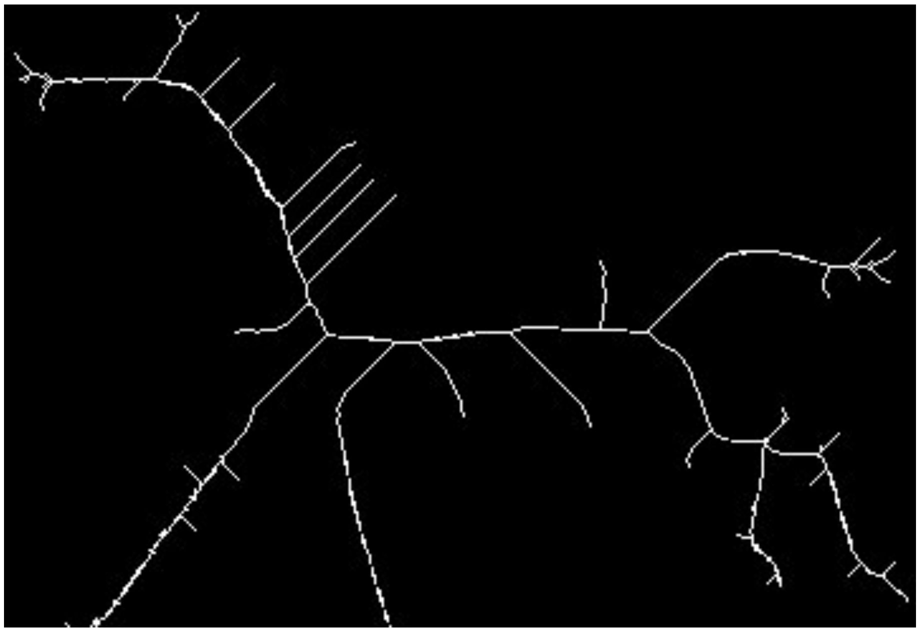


Figure 3.7 (a) Thinning with pre-processed Horse image



3.2.4 Discussion

We have seen the results of thinning and skeletonizing after we implement it on the pre-processed image of horse. In pre-processing we filled the holes in the horse and also implemented boundary smoothing by using dilation and erosion. If we do not perform these pre-processing steps, then the results that we get are as shown below:

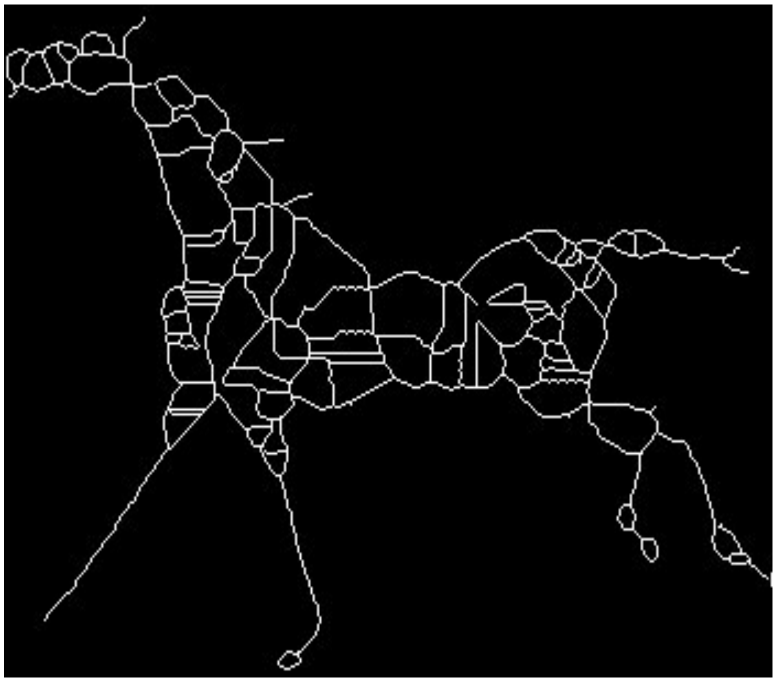


Figure 3.8 (a) Thinning without pre-processing

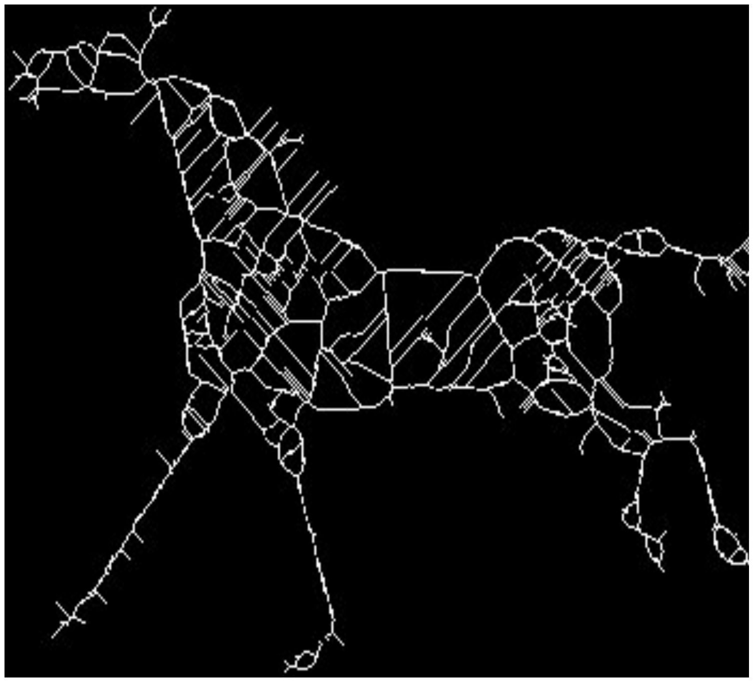


Figure 3.8 (b)4 Skeletonizing without pre-processing

Thus, by comparing the thinning and skeletonizing results obtained before and after pre-processing, we can see how important the pre-processing step is. Hole filling is more crucial than the boundary smoothing, because the presence of holes inside the object does not allow the proper usage of the mark patterns on the image. Hence, if we do not perform pre-processing, we cannot get the true thinned and skeletonized versions of the Horse image.

4. References

[1] Pattern Tables in the textbook by William K. Pratt: Digital Image Processing

[2] Color Diffusion: Error Diffusion for color halftones by Doron Shaked et. Al.