COVER PAGE:

General Info:

Name: Poornima Manjunath

Assignment no: 3

Assignment Due: 04/05/2018

**Objective:**

The aim of this assignment is to explore the spatial domain convolution while utilizing the provided simulated cake image to determine the efficiency of the famous “Sobel” edge detection algorithm. We first develop a program to perform the Sobel edge detection and test it using the MRI image and then use it with the provided simulated cake image, with edges and added noise in it, so that you can measure how good//bad the Sobel edge detection performs with noise.

**Background:**

Filtering is a technique for modifying or enhancing an image. Spatial domain operation or filtering (the processed value for the current pixel processed value for the current pixel depends on both itself and surrounding pixels). Hence Filtering is a neighborhood operation, in which the value of any given pixel in the output image is determined by applying some algorithm to the values of the pixels in the neighborhood of the corresponding input pixel. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel.

Spatial filtering is a form of finite impulse response (FIR) filtering. The filter is actually a mask of weights arranged in a rectangular pattern. The process is one of sliding the mask along the image and performing a multiply and accumulate operation on the pixels covered by the mask.

For obtaining the horizontal edges, Sobel’s mask {{-1,-2,-1},{0,0,0},{1,2,1}} is used. To obtain the vertical edges the Sobel’s mask {{-1,0,1},{-2,0,2},{1,0,1}} is used. The edge image is obtained by adding the normalized version of the absolute values of the horizontal and vertical filtered edge images.

Thresholding:

Image thresholding is a simple, yet effective, way of partitioning an image into a foreground and background. This [image analysis](https://www.mathworks.com/discovery/image-analysis.html) technique is a type of image segmentation that isolates objects by converting grayscale images into binary images. Image thresholding is most effective in images with high levels of contrast.

In the simplest implementation, the output is a [binary image](https://homepages.inf.ed.ac.uk/rbf/HIPR2/binimage.htm) representing the segmentation. Black pixels correspond to background and white pixels correspond to foreground (or vice versa). In simple implementations, the segmentation is determined by a single parameter known as the intensity threshold. In a single pass, each pixel in the image is compared with this threshold. If the [pixel's intensity](https://homepages.inf.ed.ac.uk/rbf/HIPR2/value.htm) is higher than the threshold, the pixel is set to, say, white in the output. If it is less than the threshold, it is set to black.

Sobel’s Edge detection with noisy images:

In the second part of the assignment, noise is added to the image and then the Sobel’s edge filtering is done to see how bad the Sobel’s edge detection performs with noise. We can see that the noise has increased during the edge detection and it is no longer possible to find a threshold which removes all noise pixels and at the same time retains the edges of the objects.

The object in the previous example contains sharp edges and its surface is rather smooth. Therefore, we could (in the noise-free case) easily detect the boundary of the object without getting any erroneous pixels. because it contains many fine depth variations (i.e. resulting in intensity changes in the image) on its surface.

Sobel’s filter produces thick , non-precise edges.

We can see that the intensity of many pixels on the surface is as high as along the actual edges. One reason is that the output of many edge pixels is greater than the maximum pixel value and therefore they are `cut off' at 255.

To avoid this overflow we smoothen the image prior to the edge detection and then [normalize](http://homepages.inf.ed.ac.uk/rbf/HIPR2/stretch.htm) the output, using an averaging filter.

Although the result has improved significantly, we still cannot find a threshold so that a closed line along the boundary remains and all the noise disappears.

**Algorithm**:

Part1:

1. Read an image file and put into a 2D array of 256X256 dimension.
2. To create padding, take a 258X258 2D array and make all elements set to zero.
3. Then starting indexes from i=1 to i<257 and j=1 to j<257, set the elements equal to the pixel values of the image array. This will make a 2D array of image pixels surrounded by a padding of zeros on all the four sides.
4. Then take the Sobel’s mask for obtaining the horizontal edges. {{-1,-2,-1},{0,0,0},{1,2,1}}. Multiply the pixels of the 3X3 subarray of the padded array with the pixels of the mask such that the pixel of the original image is multiplied with the center of the mask.
5. Repeat step4 till all the pixels of the original image are processed with the Sobel’s mask.
6. Take absolute of the horizontal edge filtered image and normalize it. Put it to an output image file to view it.
7. Take Sobel’s mask {{-1,0,1},{-2,0,2},{-1,0,1}} to obtain the vertical edges.Multiply the pixels of the 3X3 subarray of the padded array with the pixels of the mask such that the pixel of the original image is multiplied with the center of the mask.
8. Repeat step7 till all the pixels of the original image are processed with the Sobel’s mask.
9. Take absolute of the vertical edge filtered image and normalize it. Put it to an output image file to view it.
10. Add the horizontal and vertical edge filtered images and normalize it. Put it to an output image file to view it.
11. Take an optimal threshold value. Compare each pixel of the edge image with the threshold. If it is greater than or equal to 255, make it 255(i.e a foreground ‘white’ pixel) .Else make it 0 ,a background ‘black’ pixel.
12. However, selecting an optimal threshold to create a distinct binary edge image is not possible. Because it is difficult to say if it is an edge or the high variation of intenity on the surface of the image.

Part 2.

1. Repeat the steps mentioned in Part1 with the simulated cake image. This will yield the binary edge image of the cake.
2. Add noise to the cake image. Call rand() function. It gives a random value say X between [0,RANDMAX]. Reduce the range to [-1,1] by using the formula

Y = -1+((2\*X)/RANDMAX]

Y is the random noise in the range [-N,N] to be added to each pixel of the cake image.Different random noise is added to each pixel.

3. Repeat the steps of obtaining the binary edge image with the only difference of using the noisy image instead of original image. This will give the binary noisy edge image.

4. Take the appropriate portion of the binary edge image and the binary noisy edge image which covers the boundary of the cake. I started it from (index i=9 to i=246) and (j=9 to j=246).

5. Compare the pixels of this particular portion of the binary edge image and the binary noisy edge image. Keep count of matching pixels.

6. Obtain the accuracy using the formula Accuracy = Number of Matching pixels/Total number of pixels.

7. Increase the threshold from 1 to 255 in step of 1. Measure the corresponding accuracy.Find the optimal threshold which gives the highest accuracy.

8. Also, vary the noise level ( I varied it from 1 to 50 in step of 5). Plot the graph of accuracy versus noise level used.

9. Now apply the averaging filter i.e (1/9){{1,1,1},{1,1,1},{1,1,1}} to the noisy image. This will make every pixel of the noisy image to be multiplied by 1 and then divide by 9. This will reduce the noise by smoothing the noisy image to give a noise-filtered image.

10. Repeat the steps of obtaining the binary edge image on the noise-filtered image. Compare the accuracy results to know the effect of noise on the Sobel-based edge detection algorithm.

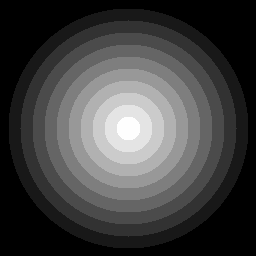
The results would have been significantly improved after noise-filtering.

11. To calculate accuracy, I compared the boundary portion of the binary edge image of noisy image and the original image without noise (I used indices i=10 to i<250 and j=10 to j<250 as boundary) and increased the count(which was initially 0) and accuracy = count/total number of pixels in the selected portion.

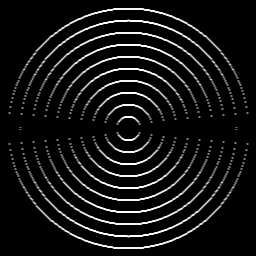
12.Same way, I calculated the accuracy of the noise filtered binary edge image.

**Results:**

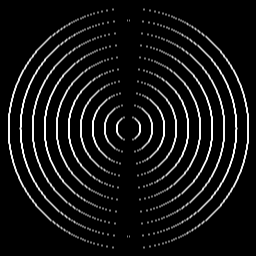
Original Cake:



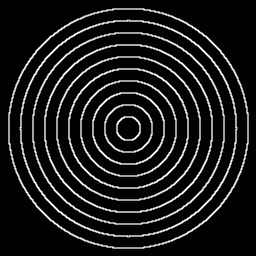
Horizontal edge image



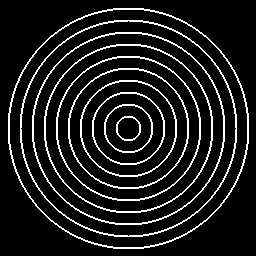
Vertical edge Image



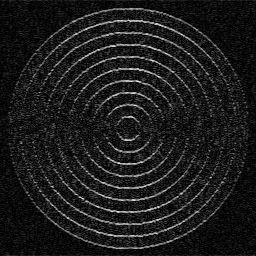
Normalized edge image



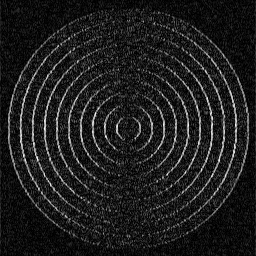
Binary edge image



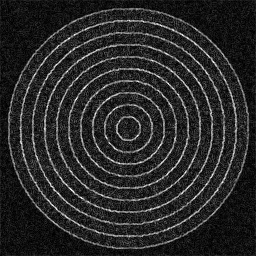
Horizontal noise image



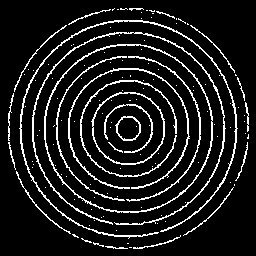
Vertical noise image



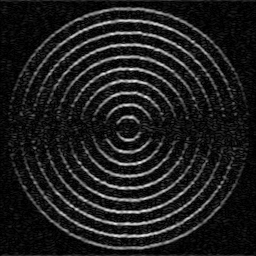
Noisy edge image



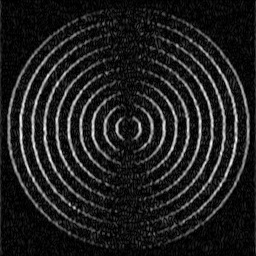
Binary Noisy edge image



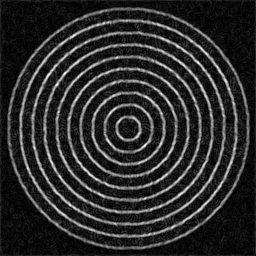
Horizontal noise filtered edge image



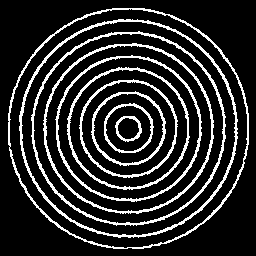
Vertical noise filtered edge image



Noise filtered edge image

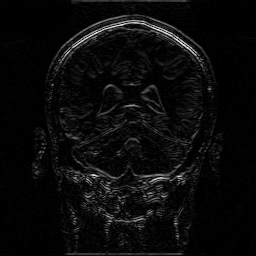


Noise filtered Binary edge image

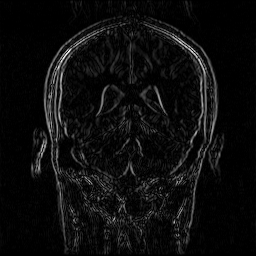


MRI images

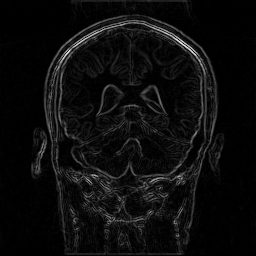
Original Mri

Horizontal Edge Image

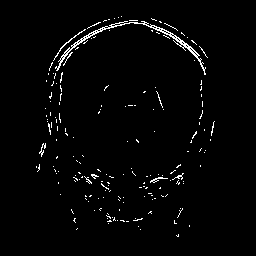
Vertical Edge image



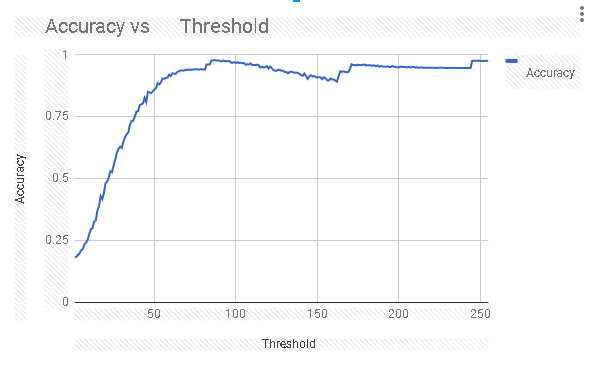
Edge image

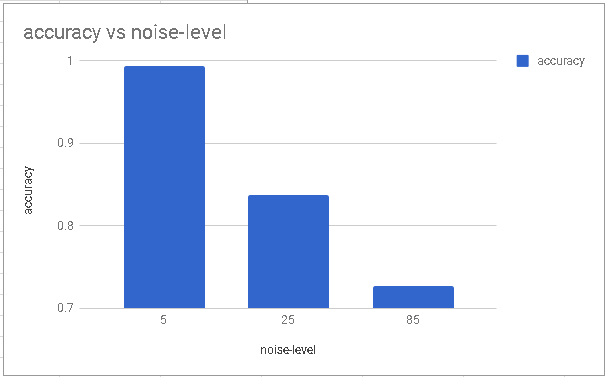


Binary Edge image



Graphs:





* The accuracy (as per my knowledge based on theory) should increase after noise-filtering.
* However, in my code because of some issue which I was not able to debug, I am getting lower accuracy after noise filtering.

**Observations:**

Answers to some of the questions asked in the assignment:

1.Taking care of boundary conditions while filtering:

I did padding of one layer of zeros around the image before convolving it with the mask. i.e I took a 258X258 array and made all elements zero. After that, from i=1 to i<257 and from j=1 to j<257, I appended the elements of my original image to the new image.

2.Finding the optimal threshold in part1:

In the first part, to obtain the binary edge image it is not possible to have an optimal threshold. Because of imprecise presence of edges. There is always a chance of losing some of the pixels depending on the threshold selected. Since the intensity function of a digital image is only known at discrete points, derivatives of this function cannot be defined unless we assume that there is an underlying continuous intensity function which has been sampled at the image points. As per my research on this issue, I found some papers related to Gradient thresholding in Sobel filtering which might be applicable here.

1. Optimal threshold in noisy image edge filtering.

For my experiment, I got maximum accuracy at 89.

1. Comparing accuracy of binary noisy edge image with binary noise-filtered edge image.

Noise affects the Sobel filter. Because it becomes difficult to differentiate between the high frequency intensity of the noise present on the surface of the image and the high frequency intensities on the edges. So noise filtering significantly improves the edge detection in Sobel filtering.

**Conclusion**:

Sobel filter is a derivate mask and is used for edge detection.

The horizontal mask will prominent the horizontal edges in an image. It also works on the principle of above mask and calculates difference among the pixel intensities of a particular edge. As the center row of mask is consisting of zeros so it does not include the original values of edge in the image but rather it calculates the difference of above and below pixel intensities of the particular edge. Thus increasing the sudden change of intensities and making the edge more visible. Similarly, the vertical mask will highlight the vertical edges.

By thresholding, you can nicely separate the edges (high frequencies) from the background. I think it will work in case of range images where you will be sure that there are no high frequencies (variation of frequencies) present on the surface.

Noise filtering significantly improves the edge detection in Sobel filtering.

Codes:

MRI filtering

/\*\*\*\*\*\*\*

\* Program to perform spatial filtering of the Image

\* For Computer Vision by Poornima Manjunath using sobel's mask

\*/

#include <math.h>

#include <stdlib.h>

#include <stdio.h>

/\*function to calculate the min of the array\*/

float calculate\_min(float image[256][256])

{

float min = image[0][0];

int i,j;

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(min>image[i][j])

{

min = image[i][j];

}

}

}

return min;

}

/\*function to calculate the max of the array\*/

float calculate\_max(float image[256][256])

{

float max = image[0][0];

int i,j;

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(max<image[i][j])

{

max = image[i][j];

}

}

}

return max;

}

/\*function to normalise the array\*/

float normalise(float magnitude,float min,float max)

{

float pixel;

float intermed;

intermed = ((magnitude-min)/(max-min));

pixel = (255\*intermed);

return pixel;

}

/\*function to filter using the given mask\*/

void filter(int in\_array[258][258], int filter[3][3],float out\_array[256][256])

{

int i,j,i\_new,j\_new;

i\_new = 0;

for(i =1;i<257;i++)

{

j\_new=0;

for(j =1;j<257;j++)

{

out\_array[i\_new][j\_new] = ((in\_array[i-1][j-1]\*filter[0][0])+(in\_array[i-1][j]\*filter[0][1])+(in\_array[i-1][j+1]\*filter[0][2])+ (in\_array[i][j-1]\*filter[1][0])+(in\_array[i][j]\*filter[1][1])+(in\_array[i][j+1]\*filter[1][2])+(in\_array[i+1][j-1]\*filter[2][0])+(in\_array[i+1][j]\*filter[2][1])+(in\_array[i+1][j+1]\*filter[2][2]));

j\_new++;

}

i\_new++;

}

}

/\*main function\*/

int main(int argc, char \*argv[]){

unsigned sizeX=256; //image width

unsigned sizeY=256; //image height

unsigned char image[sizeX][sizeY]; //image array

int new\_image[258][258]; //centered image array

unsigned levels;

int val;

unsigned char norm\_magnitude\_x[256][256];

unsigned char norm\_magnitude\_y[256][256];

//read image file

FILE \*fp;

fp = fopen("mri.pgm","r");

int i,j,k;

//reading the image file and putting it to a 2 D array

if(fp==0) {

printf("Error in reading cake image\n");

return 1;

}

if(3!=fscanf(fp, "P5 %d %d %d ", &sizeX, &sizeY, &levels)) return 1;

fread(image,sizeof(unsigned char),sizeX\*sizeY,fp);

fclose(fp);

//create an array of zeros

for(i=0;i<258;i++)

{

for(j=0;j<258;j++)

{

new\_image[i][j]= 0;

}

}

//add elements of the original image to the current array

//keeping one layer of zeros around the elements of the original array

int i\_old,j\_old;

i\_old =0;

for(i=1;i<257;i++)

{

j\_old =0;

for(j=1;j<257;j++)

{

new\_image[i][j] = image[i\_old][j\_old];

j\_old++;

}

i\_old++;

}

//take the 3X3 mask to obtain horizontal derivative

int mask[3][3] = {{-1,-2,-1},{0,0,0},{1,2,1}};

float output\_x[256][256] = {0,};

filter(new\_image,mask,output\_x); //perform horizontal derivation

//take absolute values of the horizontal filter output

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_x[i][j] = fabs(output\_x[i][j]);

}

}

//take min and max values of the horizontal filter output

float min = calculate\_min(output\_x);

float max = calculate\_max(output\_x);

//normalise the horizontal filter output array elements

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_magnitude\_x[i][j] = (unsigned char) normalise(output\_x[i][j],min,max);

}

}

//write normalised horizontal filter output to image file

FILE \*iFile = fopen("horizontal.pgm","w");

if(iFile==0) return 1; //error handling

fprintf(iFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_magnitude\_x,sizeof(unsigned char),sizeX\*sizeY,iFile);//write binary image

fclose(iFile);

//take the mask for vertical filter output

int mask2[3][3] = {{-1,0,1},{-2,0,2},{-1,0,1}};

float output\_y[256][256] = {0,};

filter(new\_image,mask2,output\_y); //perform vertical derivation

//take absolute of the vertical filter output array

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_y[i][j] = fabs(output\_y[i][j]);

}

}

min = calculate\_min(output\_y);

max = calculate\_max(output\_y);

//normalise the vertical filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_magnitude\_y[i][j] = (unsigned char) normalise(output\_y[i][j],min,max);

}

}

//write normalised vertical o/p array to image file

FILE \*vFile = fopen("vertical.pgm","w");

if(vFile==0) return 1; //error handling

fprintf(vFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_magnitude\_y,sizeof(unsigned char),sizeX\*sizeY,vFile);//write binary image

fclose(vFile);

//add the horizontal and the vertical edge filter o/ps

float edge\_filter[256][256] = {0,};

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

edge\_filter[i][j] = norm\_magnitude\_x[i][j]+norm\_magnitude\_y[i][j];

}

}

min = calculate\_min(edge\_filter);

max = calculate\_max(edge\_filter);

unsigned char norm\_edge[256][256];

//normalise the edge filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_edge[i][j] = (unsigned char) normalise(edge\_filter[i][j],min,max);

}

}

//write normalised vertical o/p array to image file

FILE \*oFile = fopen("norm\_edge\_filter.pgm","w");

if(oFile==0) return 1; //error handling

fprintf(oFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_edge,sizeof(unsigned char),sizeX\*sizeY,oFile);//write binary image

fclose(oFile);

//create a binay image using thresholding

int threshold = 80;

unsigned char threshold\_image[256][256];

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(norm\_edge[i][j] >= threshold)

{

threshold\_image[i][j] = 255;

}

else{

threshold\_image[i][j] = 0;

}

/\*\*\*\*\*\*\*

\* Program to perform spatial filtering of the Image

\* For Computer Vision by Poornima Manjunath using sobel's mask

\*/

#include <math.h>

#include <stdlib.h>

#include <stdio.h>

/\*function to calculate the min of the array\*/

float calculate\_min(float image[256][256])

{

float min = image[0][0];

int i,j;

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(min>image[i][j])

{

min = image[i][j];

}

}

}

return min;

}

/\*function to calculate the max of the array\*/

float calculate\_max(float image[256][256])

{

float max = image[0][0];

int i,j;

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(max<image[i][j])

{

max = image[i][j];

}

}

}

return max;

}

/\*function to normalise the array\*/

float normalise(float magnitude,float min,float max)

{

float pixel;

float intermed;

intermed = ((magnitude-min)/(max-min));

pixel = (255\*intermed);

return pixel;

}

/\*function to filter using the given mask\*/

void filter(int in\_array[258][258], int filter[3][3],float out\_array[256][256])

{

int i,j,i\_new,j\_new;

i\_new = 0;

for(i =1;i<257;i++)

{

j\_new=0;

for(j =1;j<257;j++)

{

out\_array[i\_new][j\_new] = ((in\_array[i-1][j-1]\*filter[0][0])+(in\_array[i-1][j]\*filter[0][1])+(in\_array[i-1][j+1]\*filter[0][2])+ (in\_array[i][j-1]\*filter[1][0])+(in\_array[i][j]\*filter[1][1])+(in\_array[i][j+1]\*filter[1][2])+(in\_array[i+1][j-1]\*filter[2][0])+(in\_array[i+1][j]\*filter[2][1])+(in\_array[i+1][j+1]\*filter[2][2]));

j\_new++;

}

i\_new++;

}

}

/\*main function\*/

int main(int argc, char \*argv[]){

unsigned sizeX=256; //image width

unsigned sizeY=256; //image height

unsigned char image[sizeX][sizeY]; //image array

int new\_image[258][258]; //centered image array

unsigned levels;

int val;

unsigned char norm\_magnitude\_x[256][256];

unsigned char norm\_magnitude\_y[256][256];

//read image file

FILE \*fp;

fp = fopen("mri.pgm","r");

int i,j,k;

//reading the image file and putting it to a 2 D array

if(fp==0) {

printf("Error in reading cake image\n");

return 1;

}

if(3!=fscanf(fp, "P5 %d %d %d ", &sizeX, &sizeY, &levels)) return 1;

fread(image,sizeof(unsigned char),sizeX\*sizeY,fp);

fclose(fp);

//create an array of zeros

for(i=0;i<258;i++)

{

for(j=0;j<258;j++)

{

new\_image[i][j]= 0;

}

}

//add elements of the original image to the current array

//keeping one layer of zeros around the elements of the original array

int i\_old,j\_old;

i\_old =0;

for(i=1;i<257;i++)

{

j\_old =0;

for(j=1;j<257;j++)

{

new\_image[i][j] = image[i\_old][j\_old];

j\_old++;

}

i\_old++;

}

//take the 3X3 mask to obtain horizontal derivative

int mask[3][3] = {{-1,-2,-1},{0,0,0},{1,2,1}};

float output\_x[256][256] = {0,};

filter(new\_image,mask,output\_x); //perform horizontal derivation

//take absolute values of the horizontal filter output

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_x[i][j] = fabs(output\_x[i][j]);

}

}

//take min and max values of the horizontal filter output

float min = calculate\_min(output\_x);

float max = calculate\_max(output\_x);

//normalise the horizontal filter output array elements

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_magnitude\_x[i][j] = (unsigned char) normalise(output\_x[i][j],min,max);

}

}

//write normalised horizontal filter output to image file

FILE \*iFile = fopen("horizontal.pgm","w");

if(iFile==0) return 1; //error handling

fprintf(iFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_magnitude\_x,sizeof(unsigned char),sizeX\*sizeY,iFile);//write binary image

fclose(iFile);

//take the mask for vertical filter output

int mask2[3][3] = {{-1,0,1},{-2,0,2},{-1,0,1}};

float output\_y[256][256] = {0,};

filter(new\_image,mask2,output\_y); //perform vertical derivation

//take absolute of the vertical filter output array

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_y[i][j] = fabs(output\_y[i][j]);

}

}

min = calculate\_min(output\_y);

max = calculate\_max(output\_y);

//normalise the vertical filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_magnitude\_y[i][j] = (unsigned char) normalise(output\_y[i][j],min,max);

}

}

//write normalised vertical o/p array to image file

FILE \*vFile = fopen("vertical.pgm","w");

if(vFile==0) return 1; //error handling

fprintf(vFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_magnitude\_y,sizeof(unsigned char),sizeX\*sizeY,vFile);//write binary image

fclose(vFile);

//add the horizontal and the vertical edge filter o/ps

float edge\_filter[256][256] = {0,};

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

edge\_filter[i][j] = norm\_magnitude\_x[i][j]+norm\_magnitude\_y[i][j];

}

}

min = calculate\_min(edge\_filter);

max = calculate\_max(edge\_filter);

unsigned char norm\_edge[256][256];

//normalise the edge filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_edge[i][j] = (unsigned char) normalise(edge\_filter[i][j],min,max);

}

}

//write normalised vertical o/p array to image file

FILE \*oFile = fopen("norm\_edge\_filter.pgm","w");

if(oFile==0) return 1; //error handling

fprintf(oFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_edge,sizeof(unsigned char),sizeX\*sizeY,oFile);//write binary image

fclose(oFile);

//create a binay image using thresholding

int threshold = 80;

unsigned char threshold\_image[256][256];

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(norm\_edge[i][j] >= threshold)

{

threshold\_image[i][j] = 255;

}

else{

threshold\_image[i][j] = 0;

}

}

}

//write thresholding output to image file

FILE \*output = fopen("threshold\_out.pgm","w");

if(output==0) return 1; //error handling

fprintf(output, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(threshold\_image,sizeof(unsigned char),sizeX\*sizeY,output);//write binary image

fclose(output);

return 0;

}

}

}

//write thresholding output to image file

FILE \*output = fopen("threshold\_out.pgm","w");

if(output==0) return 1; //error handling

fprintf(output, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(threshold\_image,sizeof(unsigned char),sizeX\*sizeY,output);//write binary image

fclose(output);

return 0;

}

Cake filter for noisy image

/\*\*\*\*\*\*\*

\* Program to perform spatial filtering of the Image

\* For Computer Vision by Poornima Manjunath using sobel's mask

\*/

#include <math.h>

#include <stdlib.h>

#include <stdio.h>

#include<time.h>

/\*function used to calculate min of all the pixels\*/

float calculate\_min(float image[256][256])

{

float min = image[0][0];

int i,j;

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(min>image[i][j])

{

min = image[i][j];

}

}

}

return min;

}

/\*function used to find max of all the pixels\*/

float calculate\_max(float image[256][256])

{

float max = image[0][0];

int i,j;

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(max<image[i][j])

{

max = image[i][j];

}

}

}

return max;

}

/\*function used for normalization\*/

float normalise(float magnitude,float min,float max)

{

float pixel;

float intermed;

intermed = ((magnitude-min)/(max-min));

pixel = (255\*intermed);

return pixel;

}

/\*function used for applying sobel filter\*/

void filter(float in\_array[258][258], int filter[3][3],float out\_array[256][256])

{

int i,j,i\_new,j\_new;

i\_new = 0;

for(i =1;i<257;i++)

{

j\_new=0;

for(j =1;j<257;j++)

{

out\_array[i\_new][j\_new] = ((in\_array[i-1][j-1]\*filter[0][0])+(in\_array[i-1][j]\*filter[0][1])+(in\_array[i-1][j+1]\*filter[0][2])+ (in\_array[i][j-1]\*filter[1][0])+(in\_array[i][j]\*filter[1][1])+(in\_array[i][j+1]\*filter[1][2])+(in\_array[i+1][j-1]\*filter[2][0])+(in\_array[i+1][j]\*filter[2][1])+(in\_array[i+1][j+1]\*filter[2][2]));

j\_new++;

}

i\_new++;

}

}

/\*main function\*/

int main(int argc, char \*argv[]){

unsigned sizeX=256; //image width

unsigned sizeY=256; //image height

unsigned char image[sizeX][sizeY]; //image array

float new\_image[258][258]; //centered image array

unsigned levels;

int val;

unsigned char norm\_magnitude\_x[256][256];

unsigned char norm\_magnitude\_y[256][256];

int noise\_level = 10; // desired noise level

//read image file

FILE \*fp;

fp = fopen("cake","r");

int i,j,k;

//reading the image file and putting it to a 2 D array

if(fp==0) {

printf("Error in reading cake image\n");

return 1;

}

if(3!=fscanf(fp, "P5 %d %d %d ", &sizeX, &sizeY, &levels)) return 1;

fread(image,sizeof(unsigned char),sizeX\*sizeY,fp);

fclose(fp);

int threshold = 89;

//print the image array to know the boundary of the cake image

//create an array of zeros

for(i=0;i<258;i++)

{

for(j=0;j<258;j++)

{

new\_image[i][j]= 0;

}

}

//add elements of the original image to the current array

//keeping one layer of zeros around the elements of the original array

int i\_old,j\_old;

i\_old =0;

for(i=1;i<257;i++)

{

j\_old =0;

for(j=1;j<257;j++)

{

new\_image[i][j] = image[i\_old][j\_old];

j\_old++;

}

i\_old++;

}

//take the 3X3 mask to obtain horizontal derivative

int mask[3][3] = {{-1,-2,-1},{0,0,0},{1,2,1}};

float output\_x[256][256] = {0,};

filter(new\_image,mask,output\_x); //perform horizontal derivation

//take absolute values of the horizontal filter output

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_x[i][j] = fabs(output\_x[i][j]);

}

}

//take min and max values of the horizontal filter output

float min = calculate\_min(output\_x);

float max = calculate\_max(output\_x);

//normalise the horizontal filter output array elements

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_magnitude\_x[i][j] = (unsigned char) normalise(output\_x[i][j],min,max);

}

}

//write normalised horizontal filter output to image file

FILE \*iFile = fopen("horizontal.pgm","w");

if(iFile==0) return 1; //error handling

fprintf(iFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_magnitude\_x,sizeof(unsigned char),sizeX\*sizeY,iFile);//write binary image

fclose(iFile);

//take the mask for vertical filter output

int mask2[3][3] = {{-1,0,1},{-2,0,2},{-1,0,1}};

float output\_y[256][256] = {0,};

filter(new\_image,mask2,output\_y); //perform vertical derivation

//take absolute of the vertical filter output array

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_y[i][j] = fabs(output\_y[i][j]);

}

}

min = calculate\_min(output\_y);

max = calculate\_max(output\_y);

//normalise the vertical filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_magnitude\_y[i][j] = (unsigned char) normalise(output\_y[i][j],min,max);

}

}

//write normalised vertical o/p array to image file

FILE \*vFile = fopen("vertical.pgm","w");

if(vFile==0) return 1; //error handling

fprintf(vFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_magnitude\_y,sizeof(unsigned char),sizeX\*sizeY,vFile);//write binary image

fclose(vFile);

//add the horizontal and the vertical edge filter o/ps

float edge\_filter[256][256] = {0,};

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

edge\_filter[i][j] = norm\_magnitude\_x[i][j]+norm\_magnitude\_y[i][j];

}

}

min = calculate\_min(edge\_filter);

max = calculate\_max(edge\_filter);

unsigned char norm\_edge[256][256] = {0,};

//normalise the edge filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_edge[i][j] = (unsigned char) normalise(edge\_filter[i][j],min,max);

}

}

//write normalised o/p array to image file

FILE \*oFile = fopen("edge\_filter.pgm","w");

if(oFile==0) return 1; //error handling

fprintf(oFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_edge,sizeof(unsigned char),sizeX\*sizeY,oFile);//write binary image

fclose(oFile);

//create a binay image using thresholding

unsigned char threshold\_image[256][256];

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(norm\_edge[i][j] >= threshold)

{

threshold\_image[i][j] = 255;

}

else{

threshold\_image[i][j] = 0;

}

}

}

//write thresholding output to image file

FILE \*output = fopen("threshold\_out.pgm","w");

if(output==0) return 1; //error handling

fprintf(output, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(threshold\_image,sizeof(unsigned char),sizeX\*sizeY,output);//write binary image

fclose(output);

//create a noisy image by adding noise in the below part of the code

float noise\_init,noise\_value,noisy\_image[256][256];

// srand(time(0));

for(i=0;i<256;i++) //apply it to the original image pixel values to get the noisy image

{

for(j=0;j<256;j++)

{

noise\_init = (float)rand(); //the range of the noise\_init will be [0,RAND\_MAX]

noise\_value = -1+ ((2\*noise\_init)/(float)RAND\_MAX); //the range of noise\_value will be [-1,1]

noise\_value = noise\_value\*noise\_level; //multiply it with desired noise level to get the range [-N,N]

noisy\_image[i][j] = (image[i][j]+noise\_value);

}

}

//repeat the steps of applying horizontal filter and vertical filter and obtaining the edge filtered image

//create an array of zeros

for(i=0;i<258;i++)

{

for(j=0;j<258;j++)

{

new\_image[i][j]= 0;

}

}

i\_old =0;

for(i=1;i<257;i++)

{

j\_old =0;

for(j=1;j<257;j++)

{

new\_image[i][j] = fabs(noisy\_image[i\_old][j\_old]);

j\_old++;

}

i\_old++;

}

float output\_noisex[256][256] = {0,};

filter(new\_image,mask,output\_noisex); //perform horizontal derivation

//take absolute values of the horizontal filter output

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_noisex[i][j] = fabs(output\_noisex[i][j]);

}

}

//take min and max values of the horizontal filter output

min = calculate\_min(output\_noisex);

max = calculate\_max(output\_noisex);

unsigned char norm\_x[256][256] = {0,};

//normalise the horizontal filter output array elements

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_x[i][j] = (unsigned char) normalise(output\_noisex[i][j],min,max);

}

}

//write normalised horizontal filter output to image file

FILE \*inFile = fopen("horizontal\_noise.pgm","w");

if(inFile==0) return 1; //error handling

fprintf(inFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_x,sizeof(unsigned char),sizeX\*sizeY,inFile);//write binary image

fclose(inFile);

float output\_noisey[256][256] = {0,};

filter(new\_image,mask2,output\_noisey); //perform vertical derivation

//take absolute of the vertical filter output array

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_noisey[i][j] = fabs(output\_noisey[i][j]);

}

}

min = calculate\_min(output\_noisey);

max = calculate\_max(output\_noisey);

unsigned char norm\_y[256][256]= {0,};

//normalise the vertical filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_y[i][j] = (unsigned char) normalise(output\_noisey[i][j],min,max);

}

}

//write normalised vertical o/p array to image file

FILE \*vnFile = fopen("vertical\_noise.pgm","w");

if(vnFile==0) return 1; //error handling

fprintf(vnFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_y,sizeof(unsigned char),sizeX\*sizeY,vnFile);//write binary image

fclose(vnFile);

//add the horizontal and the vertical edge filter o/ps

float edge\_filter\_noise[256][256] = {0,};

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

edge\_filter\_noise[i][j] = norm\_x[i][j]+norm\_y[i][j];

}

}

min = calculate\_min(edge\_filter\_noise);

max = calculate\_max(edge\_filter\_noise);

unsigned char norm\_noise[256][256] ={0,};

//normalise the edge filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_noise[i][j] = (unsigned char) normalise(edge\_filter\_noise[i][j],min,max);

}

}

//write normalised o/p array to image file

FILE \*onFile = fopen("edge\_filter\_noise.pgm","w");

if(onFile==0) return 1; //error handling

fprintf(onFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_noise,sizeof(unsigned char),sizeX\*sizeY,onFile);//write binary image

fclose(onFile);

//create a binay noise edge image using thresholding

unsigned char binary\_noise\_image[256][256];

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(norm\_noise[i][j] >= threshold)

{

binary\_noise\_image[i][j] = 255;

}

else{

binary\_noise\_image[i][j] = 0;

}

}

}

//write to binary noisy edge image to output file

FILE \*output\_noise = fopen("threshold\_noise\_out.pgm","w");

if(output\_noise==0) return 1; //error handling

fprintf(output\_noise, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(binary\_noise\_image,sizeof(unsigned char),sizeX\*sizeY,output\_noise);//write binary image

fclose(output\_noise);

//find the accuracy by comparing the appropriately selected part of binary edge image with noise and without noise

float accuracy;

int total = 0; //total number of pixels

int match = 0; //total number of matching pixels

for(i=10;i<250;i++)

{

for(j=10;j<250;j++)

{

if(threshold\_image[i][j] == binary\_noise\_image[i][j])

{

match++;

}

total++;

}

//printf("total's value is now %d\n",total);

}

//printf("Number of matching pixels is %d and the total number of pixels is %d\n",match,total);

accuracy = ((float)match/(float)total);

printf("The threshold is %d and the accuracy is %f \n",threshold,accuracy);

return 0;

}

Code 3:

/\*Code for noise filtering\*/

/\*\*\*\*\*\*\*

\* Program to perform spatial filtering of the Image

\* For Computer Vision by Poornima Manjunath using sobel's mask

\*/

#include <math.h>

#include <stdlib.h>

#include <stdio.h>

/\*function to calculate min value\*/

float calculate\_min(float image[256][256])

{

float min = image[0][0];

int i,j;

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(min>image[i][j])

{

min = image[i][j];

}

}

}

return min;

}

/\*function to calculate max value\*/

float calculate\_max(float image[256][256])

{

float max = image[0][0];

int i,j;

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(max<image[i][j])

{

max = image[i][j];

}

}

}

return max;

}

/\*function used to normalise the values\*/

float normalise(float magnitude,float min,float max)

{

float pixel;

float intermed;

intermed = ((magnitude-min)/(max-min));

pixel = (255\*intermed);

return pixel;

}

/\*function to filter the image using Sobel filter\*/

void filter(float in\_array[258][258], int filter[3][3],float out\_array[256][256])

{

int i,j,i\_new,j\_new;

i\_new = 0;

for(i =1;i<257;i++)

{

j\_new=0;

for(j =1;j<257;j++)

{

out\_array[i\_new][j\_new] = ((in\_array[i-1][j-1]\*filter[0][0])+(in\_array[i-1][j]\*filter[0][1])+(in\_array[i-1][j+1]\*filter[0][2])+ (in\_array[i][j-1]\*filter[1][0])+(in\_array[i][j]\*filter[1][1])+(in\_array[i][j+1]\*filter[1][2])+(in\_array[i+1][j-1]\*filter[2][0])+(in\_array[i+1][j]\*filter[2][1])+(in\_array[i+1][j+1]\*filter[2][2]));

j\_new++;

}

i\_new++;

}

}

/\*function used to averaging the noisy image\*/

void smooth\_filter(float in\_array[258][258],float filter2[3][3], float smooth\_arr[256][256])

{

int i,j,i\_new,j\_new;

i\_new = 0;

for(i =1;i<257;i++)

{

j\_new=0;

for(j =1;j<257;j++)

{

smooth\_arr[i\_new][j\_new] = (((in\_array[i-1][j-1]\*filter2[0][0])+(in\_array[i-1][j]\*filter2[0][1])+(in\_array[i-1][j+1]\*filter2[0][2])+(in\_array[i][j-1]\*filter2[1][0])+(in\_array[i][j]\*filter2[1][1])+(in\_array[i][j+1]\*filter2[1][2])+(in\_array[i+1][j-1]\*filter2[2][0])+(in\_array[i+1][j]\*filter2[2][1])+(in\_array[i+1][j+1]\*filter2[2][2]))/9);

j\_new++;

}

i\_new++;

}

}

int main(int argc, char \*argv[]){

unsigned sizeX=256; //image width

unsigned sizeY=256; //image height

unsigned char image[sizeX][sizeY]; //image array

float new\_image[258][258]; //centered image array

unsigned levels;

int val;

unsigned char norm\_magnitude\_x[256][256];

unsigned char norm\_magnitude\_y[256][256];

int noise\_level = 10; // desired noise level

//read image file

FILE \*fp;

fp = fopen("cake","r");

int i,j,k;

//reading the image file and putting it to a 2 D array

if(fp==0) {

printf("Error in reading cake image\n");

return 1;

}

if(3!=fscanf(fp, "P5 %d %d %d ", &sizeX, &sizeY, &levels)) return 1;

fread(image,sizeof(unsigned char),sizeX\*sizeY,fp);

fclose(fp);

//create an array of zeros

for(i=0;i<258;i++)

{

for(j=0;j<258;j++)

{

new\_image[i][j]= 0;

}

}

//add elements of the original image to the current array

//keeping one layer of zeros around the elements of the original array

int i\_old,j\_old;

i\_old =0;

for(i=1;i<257;i++)

{

j\_old =0;

for(j=1;j<257;j++)

{

new\_image[i][j] = image[i\_old][j\_old];

j\_old++;

}

i\_old++;

}

//take the 3X3 mask to obtain horizontal derivative

int mask[3][3] = {{-1,-2,-1},{0,0,0},{1,2,1}};

float output\_x[256][256] = {0,};

filter(new\_image,mask,output\_x); //perform horizontal derivation

//take absolute values of the horizontal filter output

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_x[i][j] = fabs(output\_x[i][j]);

}

}

//take min and max values of the horizontal filter output

float min = calculate\_min(output\_x);

float max = calculate\_max(output\_x);

//normalise the horizontal filter output array elements

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_magnitude\_x[i][j] = (unsigned char) normalise(output\_x[i][j],min,max);

}

}

//write normalised horizontal filter output to image file

FILE \*iFile = fopen("horizontal.pgm","w");

if(iFile==0) return 1; //error handling

fprintf(iFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_magnitude\_x,sizeof(unsigned char),sizeX\*sizeY,iFile);//write binary image

fclose(iFile);

//take the mask for vertical filter output

int mask2[3][3] = {{-1,0,1},{-2,0,2},{-1,0,1}};

float output\_y[256][256] = {0,};

filter(new\_image,mask2,output\_y); //perform vertical derivation

//take absolute of the vertical filter output array

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_y[i][j] = fabs(output\_y[i][j]);

}

}

min = calculate\_min(output\_y);

max = calculate\_max(output\_y);

//normalise the vertical filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_magnitude\_y[i][j] = (unsigned char) normalise(output\_y[i][j],min,max);

}

}

//write normalised vertical o/p array to image file

FILE \*vFile = fopen("vertical.pgm","w");

if(vFile==0) return 1; //error handling

fprintf(vFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_magnitude\_y,sizeof(unsigned char),sizeX\*sizeY,vFile);//write binary image

fclose(vFile);

//add the horizontal and the vertical edge filter o/ps

float edge\_filter[256][256] = {0,};

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

edge\_filter[i][j] = norm\_magnitude\_x[i][j]+norm\_magnitude\_y[i][j];

}

}

min = calculate\_min(edge\_filter);

max = calculate\_max(edge\_filter);

unsigned char norm\_edge[256][256] = {0,};

//normalise the edge filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_edge[i][j] = (unsigned char) normalise(edge\_filter[i][j],min,max);

}

}

//write normalised o/p array to image file

FILE \*oFile = fopen("edge\_filter.pgm","w");

if(oFile==0) return 1; //error handling

fprintf(oFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_edge,sizeof(unsigned char),sizeX\*sizeY,oFile);//write binary image

fclose(oFile);

//create a binay image using thresholding

int threshold = 89;

unsigned char threshold\_image[256][256];

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(norm\_edge[i][j] >= threshold)

{

threshold\_image[i][j] = 255;

}

else{

threshold\_image[i][j] = 0;

}

}

}

//write thresholding output to image file

FILE \*output = fopen("threshold\_out.pgm","w");

if(output==0) return 1; //error handling

fprintf(output, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(threshold\_image,sizeof(unsigned char),sizeX\*sizeY,output);//write binary image

fclose(output);

//create a noisy image by adding noise in the below part of the code

float noise\_init,noise\_value,noisy\_image[256][256];

float smooth\_arr[256][256]={0,};

for(i=0;i<258;i++) //apply it to the original image pixel values to get the noisy image

{

for(j=0;j<258;j++)

{

noise\_init = (float)rand(); //the range of the noise\_init will be [0,RAND\_MAX]

noise\_value = -1+ (2\*(noise\_init/RAND\_MAX)); //the range of noise\_value will be [-1,1]

noise\_value = noise\_value\*noise\_level; //multiply it with desired noise level to get the range [-N,N]

noisy\_image[i][j] = (fabs)(image[i][j]+noise\_value);

}

}

//repeat the steps of applying horizontal filter and vertical filter and obtaining the edge filtered image

//create an array of zeros

for(i=0;i<258;i++)

{

for(j=0;j<258;j++)

{

new\_image[i][j]= 0;

}

}

i\_old =0;

for(i=1;i<257;i++)

{

j\_old =0;

for(j=1;j<257;j++)

{

new\_image[i][j] = (noisy\_image[i\_old][j\_old]);

j\_old++;

}

i\_old++;

}

//now apply averaging filter to smoothen the noisy image

float avg\_filter[3][3] = {{1,1,1},{1,1,1},{1,1,1}};

smooth\_filter(new\_image,avg\_filter,smooth\_arr); //apply smoothing filter to the noisy image

//take min and max values of the average filter output

min = calculate\_min(smooth\_arr);

max = calculate\_max(smooth\_arr);

unsigned char norm\_avg[256][256] = {0,};

//normalise the horizontal filter output array elements;but id denoising is done properly, an extra step of normalising is not necessary

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_avg[i][j] =(unsigned char)normalise(smooth\_arr[i][j],min,max);

}

}

//write normalised horizontal filter output to image file

FILE \*avgFile = fopen("avg\_noise.pgm","w");

if(avgFile==0) return 1; //error handling

fprintf(avgFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_avg,sizeof(unsigned char),sizeX\*sizeY,avgFile);//write binary image

fclose(avgFile);

//after obtaining normalised smoothened array, create a padded array that has to be edge filtered

//create an array of zeros

for(i=0;i<258;i++)

{

for(j=0;j<258;j++)

{

new\_image[i][j]= 0;

}

}

i\_old =0;

for(i=1;i<257;i++)

{

j\_old =0;

for(j=1;j<257;j++)

{

new\_image[i][j] = norm\_avg[i\_old][j\_old];

j\_old++;

}

i\_old++;

}

float output\_noisex[256][256] = {0,};

filter(new\_image,mask,output\_noisex); //perform horizontal derivation

//take absolute values of the horizontal filter output

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_noisex[i][j] = fabs(output\_noisex[i][j]);

}

}

//take min and max values of the horizontal filter output

min = calculate\_min(output\_noisex);

max = calculate\_max(output\_noisex);

unsigned char norm\_x[256][256] = {0,};

//normalise the horizontal filter output array elements

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_x[i][j] = (unsigned char) normalise(output\_noisex[i][j],min,max);

}

}

//write normalised horizontal filter output to image file

FILE \*inFile = fopen("horizontal\_avg\_noise.pgm","w");

if(inFile==0) return 1; //error handling

fprintf(inFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_x,sizeof(unsigned char),sizeX\*sizeY,inFile);//write binary image

fclose(inFile);

float output\_noisey[256][256] = {0,};

filter(new\_image,mask2,output\_noisey); //perform vertical derivation

//take absolute of the vertical filter output array

for(i = 0;i<256;i++)

{

for(j=0;j<256;j++)

{

output\_noisey[i][j] = fabs(output\_noisey[i][j]);

}

}

min = calculate\_min(output\_noisey);

max = calculate\_max(output\_noisey);

unsigned char norm\_y[256][256]= {0,};

//normalise the vertical filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_y[i][j] = (unsigned char) normalise(output\_noisey[i][j],min,max);

}

}

//write normalised vertical o/p array to image file

FILE \*vnFile = fopen("vertical\_avg\_noise.pgm","w");

if(vnFile==0) return 1; //error handling

fprintf(vnFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_y,sizeof(unsigned char),sizeX\*sizeY,vnFile);//write binary image

fclose(vnFile);

//add the horizontal and the vertical edge filter o/ps

float edge\_filter\_noise[256][256] = {0,};

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

edge\_filter\_noise[i][j] = norm\_x[i][j]+norm\_y[i][j];

}

}

min = calculate\_min(edge\_filter\_noise);

max = calculate\_max(edge\_filter\_noise);

unsigned char norm\_noise[256][256] ={0,};

//normalise the edge filter o/p array

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

norm\_noise[i][j] = (unsigned char) normalise(edge\_filter\_noise[i][j],min,max);

}

}

//write normalised o/p array to image file

FILE \*onFile = fopen("edge\_filter\_avg\_noise.pgm","w");

if(onFile==0) return 1; //error handling

fprintf(onFile, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(norm\_noise,sizeof(unsigned char),sizeX\*sizeY,onFile);//write binary image

fclose(onFile);

//create a binay noise edge image using thresholding

unsigned char binary\_noise\_image[256][256];

for(i=0;i<256;i++)

{

for(j=0;j<256;j++)

{

if(norm\_noise[i][j] >= threshold)

{

binary\_noise\_image[i][j] = 255;

}

else{

binary\_noise\_image[i][j] = 0;

}

}

}

//write to binary noisy edge image to output file

FILE \*output\_noise = fopen("threshold\_avg\_noise\_out.pgm","w");

if(output\_noise==0) return 1; //error handling

fprintf(output\_noise, "P5 %d %d %d ",sizeX,sizeY, 255);//write header

fwrite(binary\_noise\_image,sizeof(unsigned char),sizeX\*sizeY,output\_noise);//write binary image

fclose(output\_noise);

//find the accuracy by comparing the appropriately selected part of binary edge image with noise and without noise

float accuracy;

int total = 0; //total number of pixels

int match = 0; //total number of matching pixels

for(i=10;i<250;i++)

{

for(j=10;j<250;j++)

{

if(threshold\_image[i][j] == binary\_noise\_image[i][j])

{

match++;

}

total++;

}

//printf("total's value is now %d\n",total);

}

printf("Number of matching pixels is %d and the total number of pixels is %d\n",match,total);

accuracy = ((float)match/(float)total);

printf("The accuracy is %f \n",accuracy);

return 0;

}