Image Enhancement Toolkit

Project Report

Asif Uddin

Tim Demetriades

CPE 462 - Image Processing - Section A

May 16th, 2019

Pledge: *I pledge my honor that I have abided by the Stevens Honor System.*

Asif Uddin

Tim Demetriades

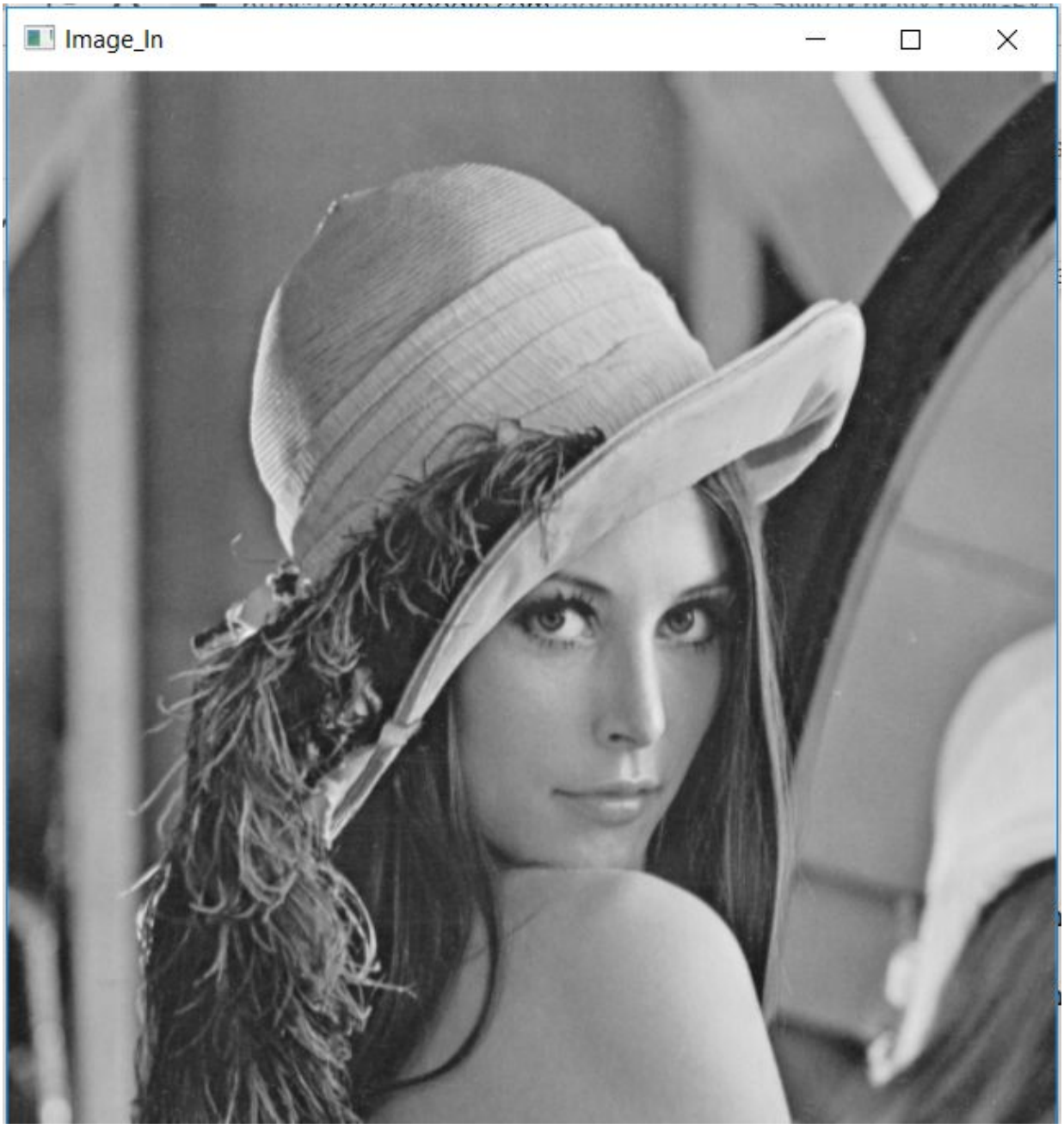
Introduction

For the image processing final project, the team chose to implement an image enhancement toolkit, applying many of the enhancement tools that we learned in class. The input image, output image file name, width, and height, are input by the user in the command line. Multiple enhancements can be done on the same image by using the output image from the previous run as the input image of the next. The list of the following tools is given below:

- Arithmetic operations
  - Negative (TD)
  - Arithmetic (TD)
- Intensity transformation
  - Logarithm (AU)
  - Inverse Logarithm (AU)
  - Power and root transform (AU)
  - Gamma correction (AU)
  - Contrast stretching (AU)
  - Histogram equalization
  - Gray level slicing (AU)
- Filters
  - Smoothing / Low pass filters (TD)
  - Smoothing / Averaging filters (TD)
  - Sharpening / High pass filters (TD)
  - Median filters (TD)
- Feature detection
  - Point detection (AU)
  - Line detection (AU)
  - Edge detection (AU)
- Image segmentation
  - Global thresholding (AU)
  - Adaptive thresholding (AU)
- Geometric Transformations
  - Translation (TD)

        - Right/Down
            - Black Background
            - White Background
        - Left/Up
            - Black Background
            - White Background
    - Rotation (TD)
        - 90 degree CW
        - 90 degree CCW
        - 180 degree
    - Reflection (TD)
        - Horizontally
        - Vertically
        - Diagonal
- Image Zooming (TD)
    - Top Left
    - Top Right
    - Bottom Left
    - Bottom Right
- Image Scaling (TD)
    - Down Scaling
        - Down Sampling (Center Pixel)
        - Down Sampling (Average)
- Adding Noise (TD)
    - Salt & Pepper Noise

In the code/results below, we will provide the code and the result for each image with an accompanying explanation.



Original Image

Code/Results

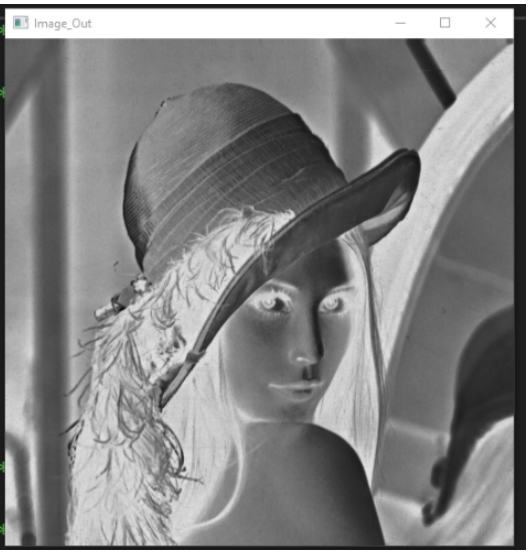
Arithmetic operations

The below functions perform basic arithmetic operations on the input image. They alter the pixel values of the image by simple addition or subtraction.

Negative

The function below creates the negative of the input image, which essentially is inverting the pixel values. This means that if the pixel was originally a low intensity dark pixel (pixel value 2 for example), then the pixel would be transformed into a high intensity light pixel (pixel value 253 for example). This is done by the simple formula below, with 255 being the max intensity a given pixel can have.

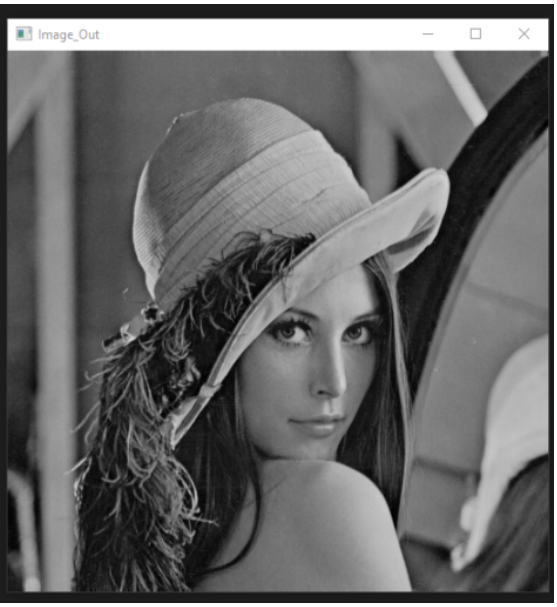
```
/*****************************************************
/* Image Processing begins
/*****************************************************



//Inverts Image

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            image_out[j][k] = 255 - image_in[j][k];
        }


/*****************************************************
/* Image Processing ends
/*****************************************************
```



Arithmetic

This is a simple arithmetic function that either makes the input image darker or lighter based on a given parameter. The example below shows the darkened image when using a value of - 25, which is making all the pixel values decrease by 25. They 'if' and "if else" statements provide an upper and lower limit so that none of the pixel values go above 255 or below 0.

```
//Arithmetic Operation

    int x = - 25;

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            if (image_in[j][k] + x > 255)
            {
                image_in[j][k] = 255;
            }
            else if (image_in[j][k] + x < 0)
            {
                image_out[j][k] = 0;
            }

            image_out[j][k] = image_in[j][k] + x;
        }
```

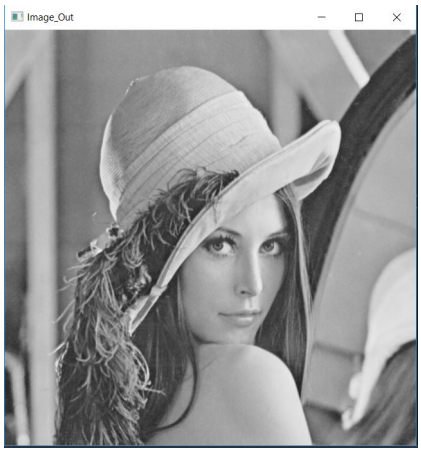


Intensity transformation

The below functions change the intensity of the pixel values based on a mathematical expression, such as logarithms and inverse logarithms.

Logarithm

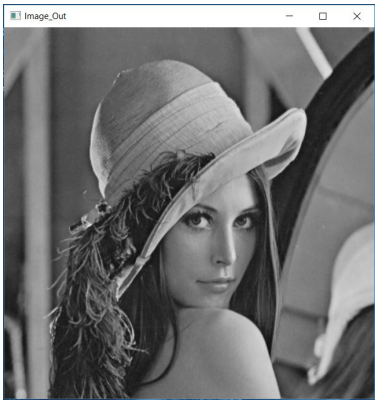
```
/*************************************************************/
/* Image Processing begins                                   */
/*************************************************************/

   // Logarithmic Identity Transform
   for (j = 0; j<height; j++)
       for (k = 0; k<width; k++)
       {
           image_out[j][k] = 256 * log2(1 + (float)image_in[j][k] / 256);
       }



/*************************************************************/
/* Image Processing ends                                     */
/*************************************************************/
```



This algorithm is an intensity logarithm transformation. The output image is 256 times the logarithm with base two of one plus the the value of the input image divided by 256. The value of the logarithm is between 0 inclusive and 1 exclusive. There are no parameters that the user can input and change. The logarithm transformation always lightens the image.

Inverse Logarithm

```
/*************************************************************/
/* Image Processing begins                                   */
/*************************************************************/

   // Inverse Logarithmic Identity Transform
   for (j = 0; j<height; j++)
       for (k = 0; k<width; k++)
       {
           image_out[j][k] = (pow(2.0, (float)image_in[j][k] / 256) - 1) * 256;
       }



/*************************************************************/
/* Image Processing ends                                     */
/*************************************************************/
```



This algorithm is an intensity inverse logarithm transformation. The output image is 256 times the power of base 2 to the exponent of the input image value divided by 256 minus one.

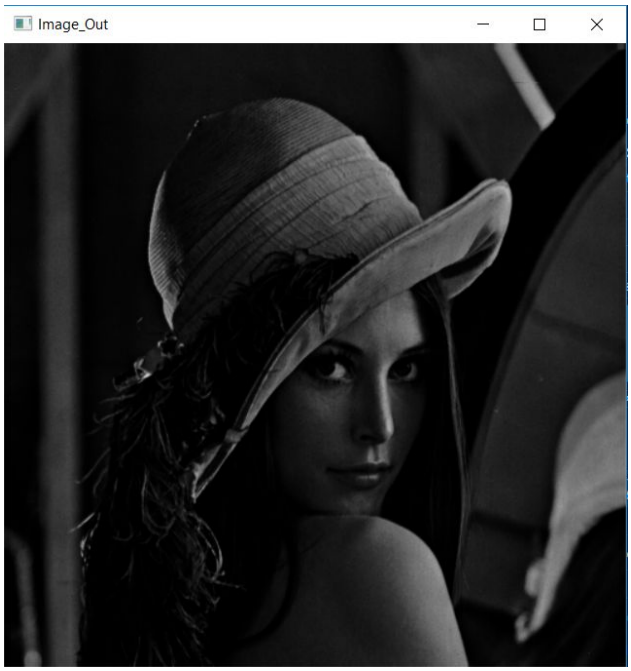
The value of the exponent is between 1 inclusive and 2 exclusive. There are no parameters that the user can input and change. The inverse logarithm transformation always darkens the image.

Power and root transform

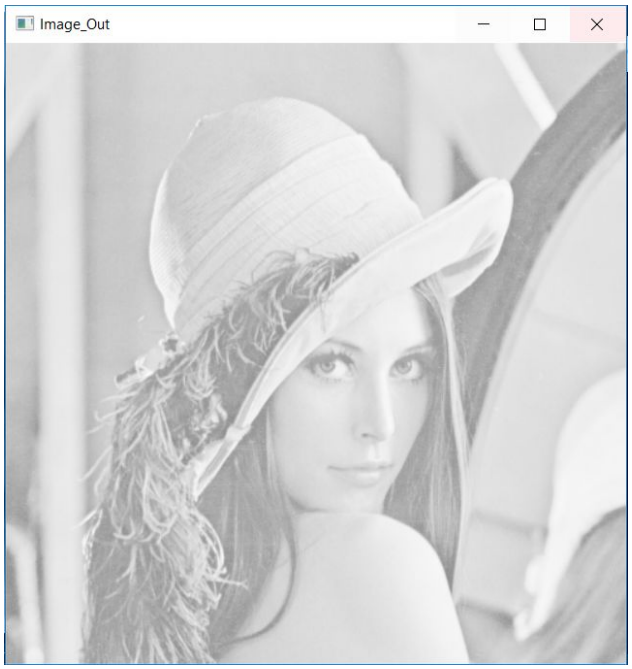
Code

```
// Power and Root Identity Transform

    float exp = 1.0/4;

    for (j = 0; j<height; j++)
        for (k = 0; k<width; k++)
        {
            image_out[j][k] = pow((float)image_in[j][k] / 256, exp) * 256;
        }
```

Output



Exp = 4



Exp = ¼

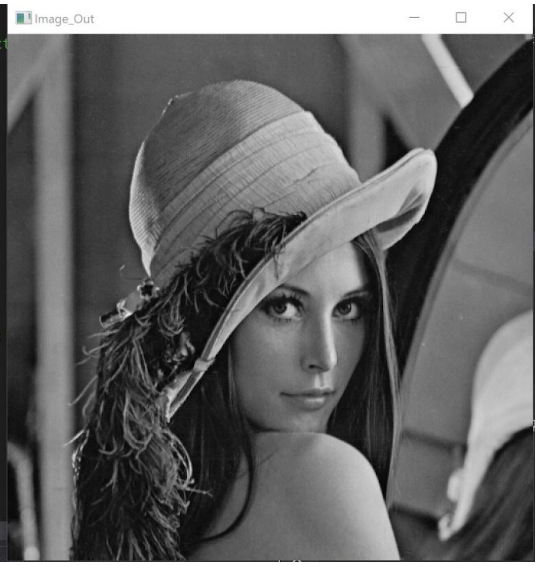
This algorithm is an intensity power and root transformation. The output image is equal to the power of the base of the input image pixel value divided by 256 to the exponent given by the user. Since the range of the input image pixel value is between 0 and 255, the base of the power is between 0 inclusive and 1 exclusive. The exponent is determined by the user but the value of the exponent determines whether it is a root transformation or a power transformation. The root transformation requires an input exponent value from 0 exclusive to 1 exclusive. The power transformation requires an input exponent value from 1 exclusive to infinity. These values also determine the range of the inputs for the exponent which is any positive real number. The root transformation lightens the image while the power transformation darkens the image.

Gamma correction

Gamma correction is used for accurately displaying images on a computer screen and controls the overall brightness of the image. It is similar to the power/root transforms. It functions by pre-distorting the image to make the displayed image look like the original. Here is an example with a gamma value of 1.5. The input range for the gamma value is 0 exclusive to infinity or any positive real number. The Constant value also has the range of any positive real number.

```
    CImgDisplay disp_in(image_disp,"Image_In",0);
    /* CImgDisplay display_name(image_displayed, "window title", normalization_fact

/*************************************************************/
/* Image Processing begins                                   */
/*************************************************************/

//Gamma Correction

    float Constant = 256;
    float Gamma = 1.5;

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            image_out[j][k] = Constant * pow((float)image_in[j][k] / 256 , Gamma);
        }

/*************************************************************/
/* Image Processing ends                                     */
/*************************************************************/

    /* display image_out */
    for (j=0; j<height; j++)
        for (k=0; k<width; k++)
```

No issues found



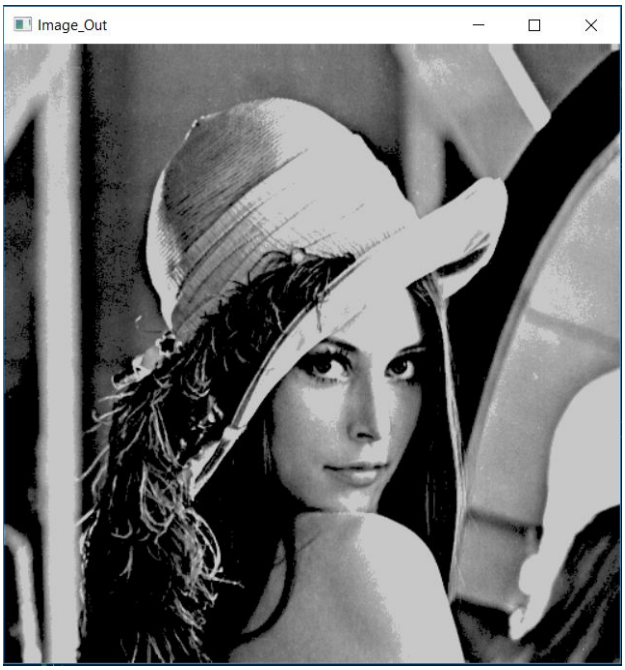
Contrast stretching

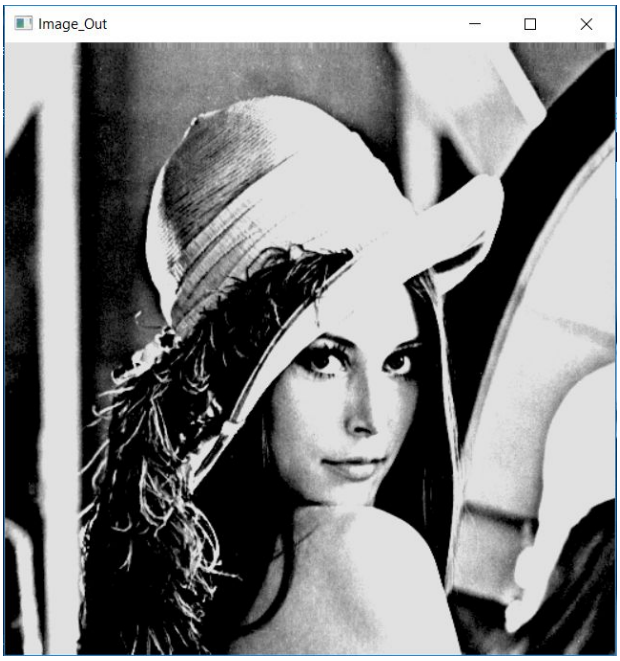
Contrast stretching is a method of ‘stretching’ the range of intensities of an image, which increases the contrast. This can be done using a histogram equalization.

Contrast Stretching Function

```
    int r1 = 96;
    int r2 = 160;
    int s1 = 50;
    int s2 = 200;
    float alpha = s1/r1;
    float beta = (s2 - s1) / (r2 - r1);
    float gamma = (255 -s2)/(255-r2);

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            if (image_in[j][k] < r1) {
                image_out[j][k] = round(alpha*image_in[j][k]);
            }
            else if (image_in[j][k] > r2) {
                image_out[j][k] = round(gamma*(image_in[j][k] - r2) + s2);
            }
            else
            {
                image_out[j][k] = round(beta*(image_in[j][k] - r1) + s1);
            }
        }
```

Output





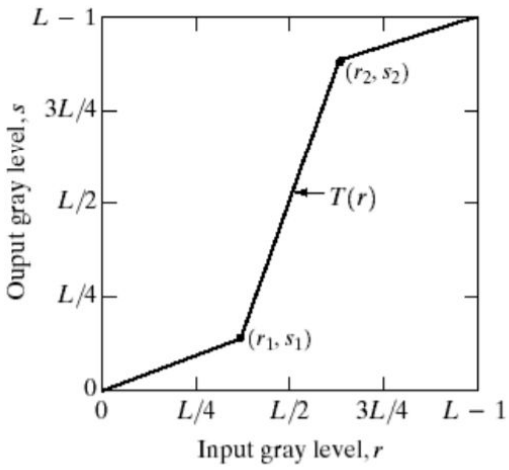
r1 = 96, r2 = 160

s1 = 50, s2 = 200

r1 = 100, r2 = 150

s1 = 25, s2 = 225

This algorithm is a contrast stretching algorithm. In this algorithm, the user has four inputs (r1, s1 & r2, s2) that divide the image into three regions. These input values have ranges between 0 and 255. Ideally, the user would like to create a piecewise function as the one on the right to create a higher contrast in the image. The input values below r1 and above r2 are typically multiplied by values less than 1 to squeeze the output values towards the ends. The output values between r1 and r2 and s1 and s2 are stretched which means more output values represent less input values. In the output image above, the difference between r1 and r2 is smaller and the

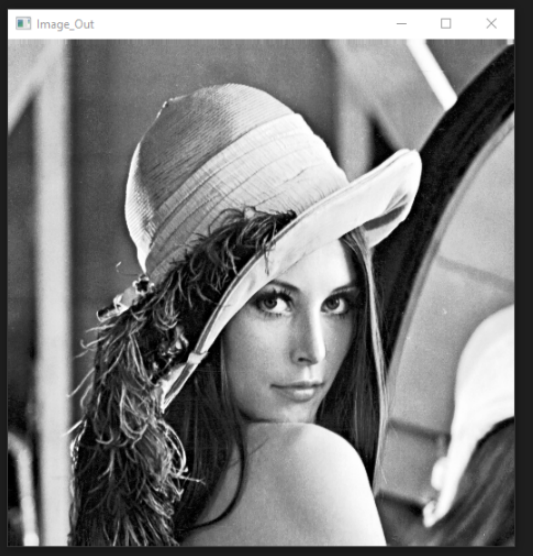


difference between s1 and s2 is bigger. This gives the image on the right a higher contrast than the image on the left.

Histogram equalization

This function applies a histogram equalization, which increases the contrast or difference between intensities of the image. It does this by first making a histogram of the input image, creating a distribution of pixel intensities in an array. Then, a mapping function is created based off of the histogram. Finally, the mapping function is applied to the original image.

```
//Histogram Equilization (fixes contrast)

    int pixelTotal = width * height;
    float pixelValue = 1 / ((float)pixelTotal);
    float hist[256] = { };  //initilize histagram array to 0
    float maphist[256] = { };  //initilize mapping array to 0

    //counting each intensity (making histogram)
    for (j = 0; j < height; j++) {
        for (k = 0; k < width; k++)  //scan entire image

            hist[image_in[j][k]] += pixelValue;
            // hist[i] = hist[i-1] + pixelValue

    }

    //mapping function of histagram
    for (j = 0; j < 256; j++) {
        for (k = 0; k < (j + 1); k++)

            maphist[j] += hist[k];  //summation of histogram
        maphist[j] = maphist[j] * (256 - 1);  //based on formula (L=256)
    }

    //apply mapping function to original image
    for (j = 0; j < height; j++) {
        for (k = 0; k < width; k++)

            image_out[j][k] = (int)maphist[image_in[j][k]];

    }
```



Gray level slicing

This function identifies a region of values to set a highlight on a particular area in the image with those values.

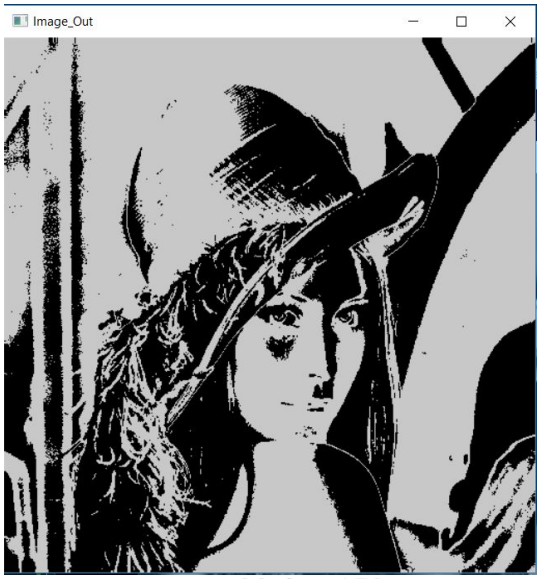
With Background

```
int a = 100;
int b = 130;
int gl = 150;

for (j = 0; j < height; j++)
    for (k = 0; k < width; k++)
    {
        if (image_in[j][k] < a) {
            image_out[j][k] = image_in[j][k];
        }
        else if (image_in[j][k] > b) {
            image_out[j][k] = image_in[j][k];
        }
        else
        {
            image_out[j][k] = gl;
        }
    }
```

Without Background

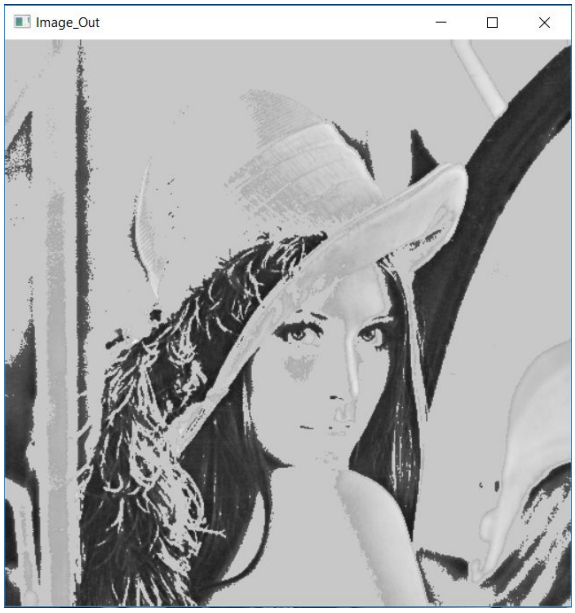
```
int a = 70;
int b = 130;
int gl = 150;
int ll = 0;

for (j = 0; j < height; j++)
    for (k = 0; k < width; k++)
    {
        if (image_in[j][k] < a) {
            image_out[j][k] = ll;
        }
        else if (image_in[j][k] > b) {
            image_out[j][k] = ll;
        }
        else
        {
            image_out[j][k] = gl;
        }
    }
```

Output



a = 90, b = 170

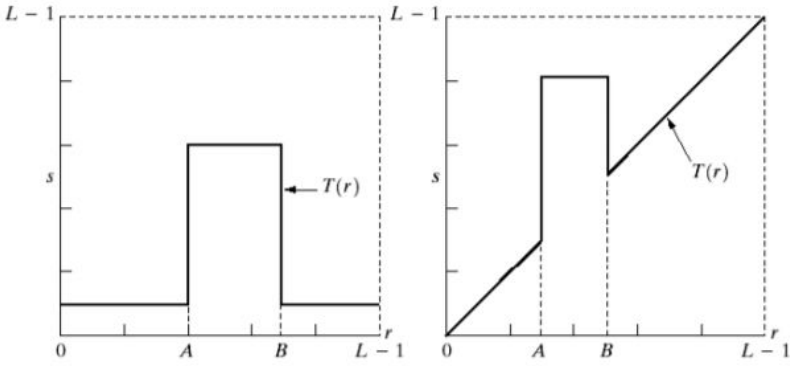
gl = 200, ll = 0



a = 90, b = 170

gl = 200

This algorithm is a gray slicing algorithm. In this algorithm, the user can choose between two values for the region of the slice (a and b) and choose what output value (gl) to set for the region inside the slice. There are two methods for this- the difference lies in how the region outside of the slice is handled. The first method changes the background and sets the output values to a lower level (ll) and the second method ignores the background and sets the output values to the same value as the one in the input image. The range is limited from 0 to 255 for all four input parameters. This algorithm is the most useful if you want to highlight a specific area within the image that has a different pixel value to the background.



Filters

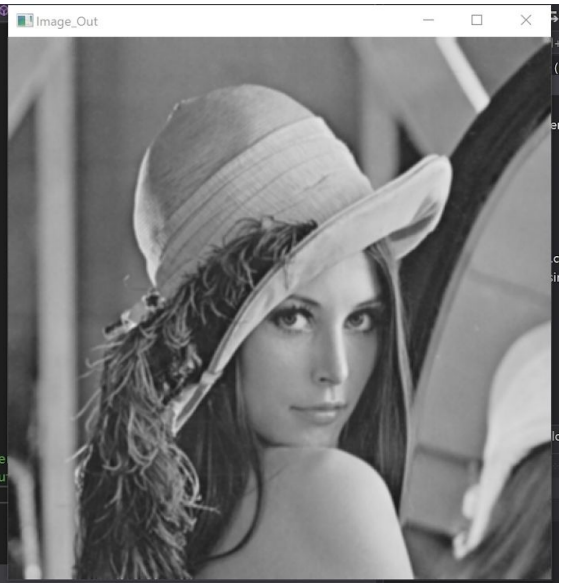
The below functions apply filters to the input image, altering its overall look.

Smoothing / Low pass filters

This function applies an averaging low pass filter filter using a 3 x 3 matrix with all 1's that is multiplied by 1/9. It produces a smoother image with less noise. Increasing the size of the mask will make the image smoother, which will eventually make the image exceedingly blurry. The example here uses the LPF matrix found below.

```
/* Image Processing begins                                          */
/********************************************************************/

   // Low Pass Filter (Smooths Image)

   float count = 0.0;
   float mask[3][3] = { {0.11, 0.11, 0.11},{0.11, 0.11, 0.11},{0.11, 0.11, 0.11} };
   float output = 0.0;

   for (j = 0; j < height; j++)
       for (k = 0; k < width; k++)
       {
           if (j == 0 || k == 0 || j == height - 1 || k == width - 1)
               image_out[j][k] = image_in[j][k];
           else {
               count = 0.0;
               for (int l = 0; l < 3; l++)
               {
                   for (int m = 0; m < 3; m++)
                   {
                       count += ((float)image_in[j + l - 1][k + m - 1] * mask[l][m]);
                   }
               }
               output = count;
               output = (output > 255 ? 255 : output); // output over 255 = 255 , unde
               output = (output < 0 ? 0 : output); // output under 0 = 0 , over 0 = ou
               image_out[j][k] = (int)output;
           }
       }

/********************************************************************/
```

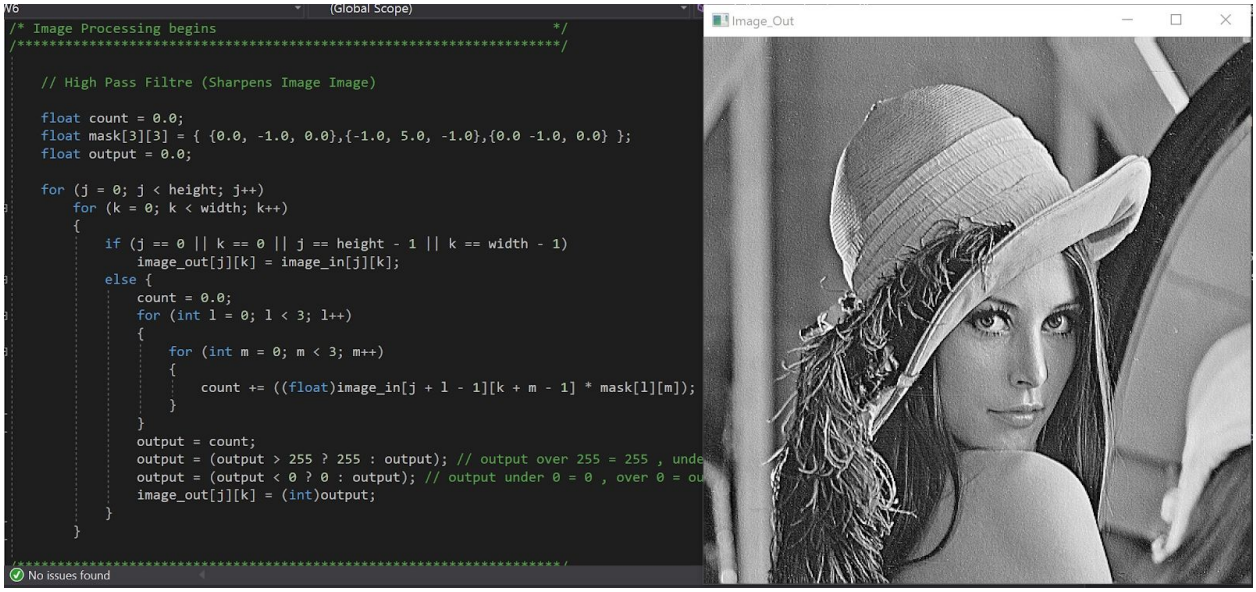
No issues found



$$\frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Sharpening / High pass filters

This filter is the opposite of the smoothing filter - it enhances the edges, which als enhances the noise. It does this by applying the composite Laplacian operator, found below.



```
/* Image Processing begins                                          */
/*******************************************************************/

    // High Pass Filtre (Sharpens Image Image)

    float count = 0.0;
    float mask[3][3] = { {0.0, -1.0, 0.0},{-1.0, 5.0, -1.0},{0.0 -1.0, 0.0} };
    float output = 0.0;

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            if (j == 0 || k == 0 || j == height - 1 || k == width - 1)
                image_out[j][k] = image_in[j][k];
            else {
                count = 0.0;
                for (int l = 0; l < 3; l++)
                {
                    for (int m = 0; m < 3; m++)
                    {
                        count += ((float)image_in[j + l - 1][k + m - 1] * mask[l][m]);
                    }
                }
                output = count;
                output = (output > 255 ? 255 : output); // output over 255 = 255 , unde
                output = (output < 0 ? 0 : output); // output under 0 = 0 , over 0 = ou
                image_out[j][k] = (int)output;
            }
        }

/*******************************************************************/
```

No issues found

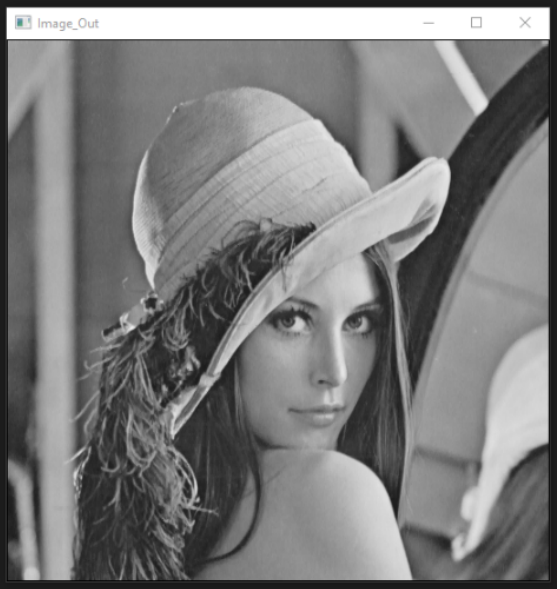
Image_Out

0	−1	0
−1	5	−1
0	−1	0

Median filters

The function of this filter is to smooth the input image and eliminate noise. In a sense it is a better low pass filter. It accomplishes this by going through all of the pixels in the image and taking the median pixel value of groups of three and applying it to the output image. You can notice in the output below there is now less noise.

```
// Median Filter

   int array[3] = { };

   int a;
   int b;
   int c;

   for (j = 1; j < height - 1; j++)
       for (k = 1; k < width - 1; k++)
       {
           int a = image_in[j - 1][k]; //a = first pixel
           int b = image_in[  j  ][k]; //b = second pixel
           int c = image_in[j + 1][k]; //c = third pixel

           //check for median b
           if ((a < b && b < c) || (c < b & b < a))
               image_out[j][k] = b;

           //check for median a
           else if ((b < a && a < c) || (c < a && a < b))
               image_out[j][k] = a;

           //median c
           else
               image_out[j][k] = c;

       }
```

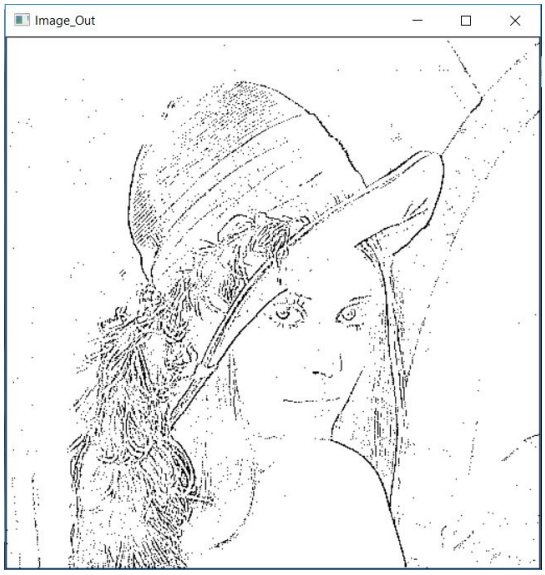


Feature detection

Point detection

```
/*  Point detection algorithm. Convolute the image with the
mask below. If the output is greater than the threshold
value, the pixel is white else black.
*/

/*  pt is a mask for point detection
*/

    int pt[3][3] = { { -1,-1,-1 },{ -1,8,-1 },{ -1,-1,-1 } };

    /*  a is the result of the convolution
    */

    int a;

    // t is the threshold value. User input should allow it to be changed

    int t = 200;

    for (j = 1; j< height - 1; j++)
        for (k = 1; k < width - 1; k++)
        {
            a = image_in[j - 1][k - 1] * pt[0][0]
                + image_in[j + 0][k - 1] * pt[1][0]
                + image_in[j + 1][k - 1] * pt[2][0]
                + image_in[j - 1][k + 0] * pt[0][1]
                + image_in[j + 0][k + 0] * pt[1][1]
                + image_in[j + 1][k + 0] * pt[2][1]
                + image_in[j - 1][k + 1] * pt[0][2]
                + image_in[j + 0][k + 1] * pt[1][2]
                + image_in[j + 1][k + 1] * pt[2][2];

            if (a > t) {
                image_out[j][k] = 255;
            }
            else
            {
                image_out[j][k] = 0;
            }
        }
```

Threshold Value: 200



Threshold Value: -100

This algorithm is a point detection algorithm. This algorithm applies convolution to a 3x3 mask through the entire image. The mask is shown in the 3x3 table to the right. This mask is initialized to the array pt in the code above. The result of

−1	−1	−1
−1	8	−1
−1	−1	−1

the convolution is equal to a for each pixel. This value is compared towards the threshold value. The threshold value is the only value that the user can change in this code. The range is any real number. The lower the threshold value, the more pixels become white. The higher the threshold value, the more pixels become black. In the two outputs on the right hand side of the previous page, the threshold value of 200 created an outline of the face with white pixels on a black background and the threshold value of -100 created an outline of the face with black pixels on a white background. Although the input value for the images are always between 0 and 255 inclusive, the threshold value can be any real number, including negative numbers and values greater than 255.

Line detection

Horizontal Line Detection

```
/*****************************************************************/
/* Image Processing begins                                       */
/*****************************************************************/

//Horizontal Line Detection

// hm is the horizontal mask that needs to be applied to do the convolution
// t is the threshold value that the output compares to
   int hm[3][3] = { { -1,-1,-1 },{ 2, 2, 2 },{ -1,-1,-1 } };
   int t = 70;
   int a = 0;

   for (j = 1; j< height - 1; j++)
       for (k = 1; k < width - 1; k++)
       {
           /*  a multiply the value in the vicinity and sum them
           convolution
           a is the result of the horizontal convolution
           */
           a = image_in[j - 1][k - 1] * hm[0][0]
               + image_in[j + 0][k - 1] * hm[1][0]
               + image_in[j + 1][k - 1] * hm[2][0]
               + image_in[j - 1][k + 0] * hm[0][1]
               + image_in[j + 0][k + 0] * hm[1][1]
               + image_in[j + 1][k + 0] * hm[2][1]
               + image_in[j - 1][k + 1] * hm[0][2]
               + image_in[j + 0][k + 1] * hm[1][2]
               + image_in[j + 1][k + 1] * hm[2][2];
           if (a > t) {
               image_out[j][k] = 255;
           }
           else
           {
               image_out[j][k] = 0;
           }
       }



/*****************************************************************/
/* Image Processing ends                                         */
```

Threshold Value: 70

Threshold Value: 30

This algorithm is a horizontal line detection algorithm. This algorithm applies convolution to a 3x3 mask through the entire image. The mask is shown in the 3x3 table to the right. This mask is initialized to the array hm in the code above. The result of the convolution is equal to a for each pixel. This value is compared towards the threshold value. The threshold value is the only value that the user can change in this code. The range is any real number. The lower the threshold value, the more pixels become white. The higher the threshold value, the more pixels become black. In the two outputs on the top of this page, the threshold value of 70 created a crude sketch of the face with white pixels on a black background while the threshold value of 30 created a more descriptive sketch of the picture which has the outline of the face and the hat. Although the input value for the images are always between 0 and 255 inclusive, the threshold value can be any real number, including negative numbers and values greater than 255. The threshold of 70 also shows

−1	−1	−1
2	2	2
−1	−1	−1

Horizontal

the nature of the horizontal line detection algorithm as many of the white pixels are in horizontal streaks.

Vertical Line Detection

Code

```
//Vertical Line Detection

// vm is the vertical mask that needs to be applied to do the convolution
// t is the threshold value that the output compares to. User should be able to change it
    int vm[3][3] = { { -1, 2,-1 },{ -1, 2,-1 },{ -1, 2,-1 } };
    int t = 70;
    int b = 0;

    for (j = 1; j< height - 1; j++)
        for (k = 1; k < width - 1; k++)
        {
            /*  b multiplies the value in the vicinity and sum them
            convolution
            b is the result of the vertical convolution
            */
            b = image_in[j - 1][k - 1] * vm[0][0]
                + image_in[j + 0][k - 1] * vm[1][0]
                + image_in[j + 1][k - 1] * vm[2][0]
                + image_in[j - 1][k + 0] * vm[0][1]
                + image_in[j + 0][k + 0] * vm[1][1]
                + image_in[j + 1][k + 0] * vm[2][1]
                + image_in[j - 1][k + 1] * vm[0][2]
                + image_in[j + 0][k + 1] * vm[1][2]
                + image_in[j + 1][k + 1] * vm[2][2];
            if (b > t) {
                image_out[j][k] = 255;
            }
            else
            {
                image_out[j][k] = 0;
            }
        }
```

Threshold Value : 70

Threshold Value: 150

This algorithm is a vertical line detection algorithm. This algorithm applies convolution to a 3x3 mask through the entire image. The mask is shown in the 3x3 table to the right. This mask is initialized to the array vm in the code above. The result of the convolution is equal to b for each pixel. This value is compared towards the threshold value. The threshold value is the only value that the user can change in this code. The range is any real number. The lower the threshold value, the more pixels become white. The higher the threshold value, the more pixels become black. In the two outputs on the top of this page, the threshold value of 70 creates a descriptive sketch of the face with white pixels on a black background while the threshold value of 150 creates a crude sketch with white pixels on a white background. The threshold of 150 also shows the nature of the vertical line detection algorithm as many of the white pixels are in vertical streaks.Although the

−1	2	−1
−1	2	−1
−1	2	−1

Vertical

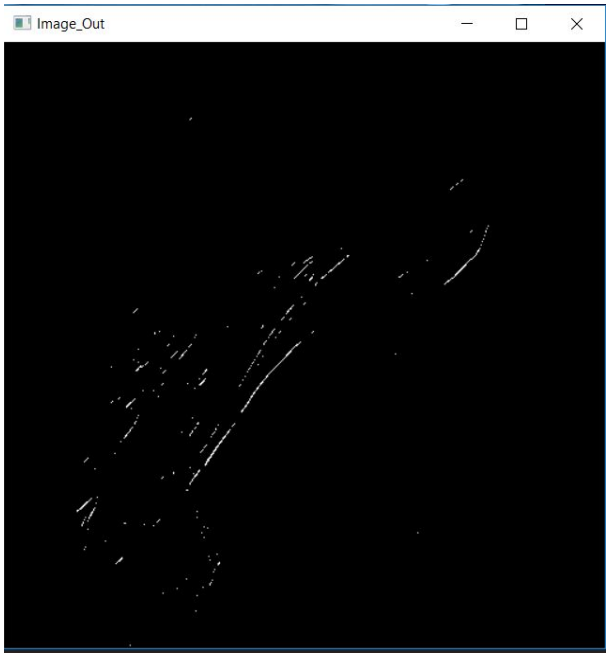
input value for the images are always between 0 and 255 inclusive, the threshold value can be any real number, including negative numbers and values greater than 255.

Positive 45 degree Detection

Code

```
//Positive 45 Degree Detection

// pos45 is the positive 45 degree mask that needs to be applied to do the convolution
// t is the threshold value that the output compares to. User should be able to change it
    int pos45[3][3] = { { -1,-1, 2 },{ -1, 2,-1 },{ 2,-1,-1 } };
    int t = 150;
    int c = 0;

    for (j = 1; j< height - 1; j++)
        for (k = 1; k < width - 1; k++)
        {
            /*  b multiplies the value in the vicinity and sum them
            convolution
            b is the result of the vertical convolution
            */
            c = image_in[j - 1][k - 1] * pos45[0][0]
                + image_in[j + 0][k - 1] * pos45[1][0]
                + image_in[j + 1][k - 1] * pos45[2][0]
                + image_in[j - 1][k + 0] * pos45[0][1]
                + image_in[j + 0][k + 0] * pos45[1][1]
                + image_in[j + 1][k + 0] * pos45[2][1]
                + image_in[j - 1][k + 1] * pos45[0][2]
                + image_in[j + 0][k + 1] * pos45[1][2]
                + image_in[j + 1][k + 1] * pos45[2][2];
            if (c > t) {
                image_out[j][k] = 255;
            }
            else
            {
                image_out[j][k] = 0;
            }
        }
```

Threshold Value: 70



Threshold Value : 150

This algorithm is a positive 45 degree line detection algorithm. This algorithm applies convolution to a 3x3 mask through the entire image. The mask is shown in the 3x3 table to the right. This mask is initialized to the array pos45 in the code above. The result of the convolution is equal to c for each pixel. This value is compared towards the threshold value. The threshold value is the only value that the user can change in this code. The range is any real number. The lower the threshold value, the more pixels become white. The higher the threshold value, the more pixels become black. In the two outputs on the top of this page, the threshold value of 70 creates a descriptive sketch of the face with white pixels on a black background while the threshold value of 150 creates a crude sketch with white pixels on a white background. The threshold of 150 also shows the nature of the positive 45 detection algorithm as many of the white pixels are streaks in this

−1	−1	2
−1	2	−1
2	−1	−1

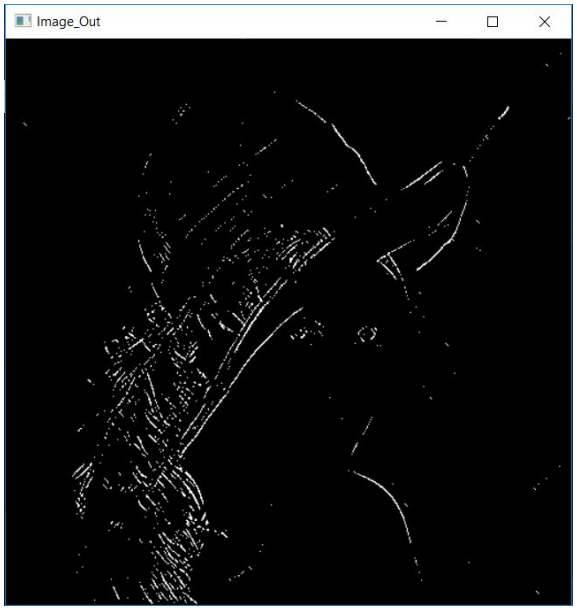
+45°

direction. Although the input value for the images are always between 0 and 255 inclusive, the threshold value can be any real number, including negative numbers and values greater than 255.

Negative 45 degree Detection

Code

```
//Negative 45 Degree Detection

// neg45 is the positive 45 degree mask that needs to be applied to do the convolution
// t is the threshold value that the output compares to. User should be able to change it
    int neg45[3][3] = { { 2,-1, -1 },{ -1, 2,-1 },{ -1,-1,2 } };
    int t = 150;
    int d = 0;

    for (j = 1; j< height - 1; j++)
        for (k = 1; k < width - 1; k++)
        {
            /*  d multiplies the value in the vicinity and sum them
            convolution
            d is the result of the vertical convolution
            */
            d = image_in[j - 1][k - 1] * neg45[0][0]
                + image_in[j + 0][k - 1] * neg45[1][0]
                + image_in[j + 1][k - 1] * neg45[2][0]
                + image_in[j - 1][k + 0] * neg45[0][1]
                + image_in[j + 0][k + 0] * neg45[1][1]
                + image_in[j + 1][k + 0] * neg45[2][1]
                + image_in[j - 1][k + 1] * neg45[0][2]
                + image_in[j + 0][k + 1] * neg45[1][2]
                + image_in[j + 1][k + 1] * neg45[2][2];
            if (d > t) {
                image_out[j][k] = 255;
            }
            else
            {
                image_out[j][k] = 0;
            }
        }
```



Threshold Value: 70

Threshold Value : 150

This algorithm is a negative 45 degree line detection algorithm. This algorithm applies convolution to a 3x3 mask through the entire image. The mask is shown in the 3x3 table to the right. This mask is initialized to the array neg45 in the code above. The result of the convolution is equal to d for each pixel. This value is compared towards the threshold value. The threshold value is the only value that the user can change in this code. The range is any real number. The lower the threshold value, the more pixels become white. The higher the threshold value, the more pixels become black. In the two outputs on the top of this page, the threshold value of 70 creates a descriptive sketch of the face with white pixels on a black background while the threshold value of 150 creates a crude sketch with white pixels on a white background. The threshold of 150 also shows the nature of the negative 45 detection algorithm as many of the white pixels are streaks in this direction. The streaks in the other directions are chopped up and

2	−1	−1
−1	2	−1
−1	−1	2

−45°

not smooth like the positive 45 degree detection algorithm. Although the input value for the images are always between 0 and 255 inclusive, the threshold value can be any real number, including negative numbers and values greater than 255.

Combined Line Detection

Code

The code is separated into three images because it was not possible to take one screenshot of the entire algorithm. Please read the code sequentially through the screenshots.

```
/*****************************************************************/
/* Image Processing begins                                       */
/*****************************************************************/

/*  This is the combined output for the line detection model.
This combines the horizontal, vertical, positive 45, and
negative 45 masks for a single composited image.
*/

/*  These four matrices are the masks for the line detection model.
hm is horizontal mask,
vm is vertical mask,
pos45 is the positive 45 degree mask, and
neg45 is the negative 45 degree mask
*/

int hm[3][3] = { { -1,-1,-1 },{ 2, 2, 2 },{ -1,-1,-1 } };
int vm[3][3] = { { -1, 2,-1 },{ -1, 2,-1 },{ -1, 2,-1 } };
int pos45[3][3] = { { -1,-1, 2 },{ -1, 2,-1 },{ 2,-1,-1 } };
int neg45[3][3] = { { 2,-1,-1 },{ -1, 2,-1 },{ -1,-1, 2 } };

/*  Result from convolution
a is the result of the horizontal convolution
b is the result of the vertical convolution
c is the result of the positive 45 degree convolution
d is the result of the negative 45 degree convolution
*/

int a;
int b;
int c;
int d;

/*  t is the threshold value for the model.
This value needs to be able to change based of user's input
*/

int t = 70;
```

```
for (j = 1; j< height - 1; j++)
    for (k = 1; k< width - 1; k++)
    {
        /*  a, b, c, d multiply the value in the vicinity and sum them (convolution)*/
    a = image_in[j - 1][k - 1] * hm[0][0]
        + image_in[j + 0][k - 1] * hm[1][0]
        + image_in[j + 1][k - 1] * hm[2][0]
        + image_in[j - 1][k + 0] * hm[0][1]
        + image_in[j + 0][k + 0] * hm[1][1]
        + image_in[j + 1][k + 0] * hm[2][1]
        + image_in[j - 1][k + 1] * hm[0][2]
        + image_in[j + 0][k + 1] * hm[1][2]
        + image_in[j + 1][k + 1] * hm[2][2];
    b = image_in[j - 1][k - 1] * vm[0][0]
        + image_in[j + 0][k - 1] * vm[1][0]
        + image_in[j + 1][k - 1] * vm[2][0]
        + image_in[j - 1][k + 0] * vm[0][1]
        + image_in[j + 0][k + 0] * vm[1][1]
        + image_in[j + 1][k + 0] * vm[2][1]
        + image_in[j - 1][k + 1] * vm[0][2]
        + image_in[j + 0][k + 1] * vm[1][2]
        + image_in[j + 1][k + 1] * vm[2][2];
    c = image_in[j - 1][k - 1] * pos45[0][0]
        + image_in[j + 0][k - 1] * pos45[1][0]
        + image_in[j + 1][k - 1] * pos45[2][0]
        + image_in[j - 1][k + 0] * pos45[0][1]
        + image_in[j + 0][k + 0] * pos45[1][1]
        + image_in[j + 1][k + 0] * pos45[2][1]
        + image_in[j - 1][k + 1] * pos45[0][2]
        + image_in[j + 0][k + 1] * pos45[1][2]
        + image_in[j + 1][k + 1] * pos45[2][2];
    d = image_in[j - 1][k - 1] * neg45[0][0]
        + image_in[j + 0][k - 1] * neg45[1][0]
        + image_in[j + 1][k - 1] * neg45[2][0]
        + image_in[j - 1][k + 0] * neg45[0][1]
        + image_in[j + 0][k + 0] * neg45[1][1]
        + image_in[j + 1][k + 0] * neg45[2][1]
        + image_in[j - 1][k + 1] * neg45[0][2]
        + image_in[j + 0][k + 1] * neg45[1][2]
        + image_in[j + 1][k + 1] * neg45[2][2];
    /*  this compares the a,b,c, and d value with the threshold
    value
    if one of these values are bigger than the threshold value, it exists
    out of this conditional statements and sets the value to black
    if all of these values are less than the threshold value, it
    sets the output value to white
    */
```

```
if (a > t) {
    image_out[j][k] = 255;
}
else  if (b > t)
{
    image_out[j][k] = 255;
}
else  if (c > t)
{
    image_out[j][k] = 255;
}
else  if (d > t)
{
    image_out[j][k] = 255;
}
else
{
    image_out[j][k] = 0;
}

}
```

Threshold Value: 70

Threshold Value: 150

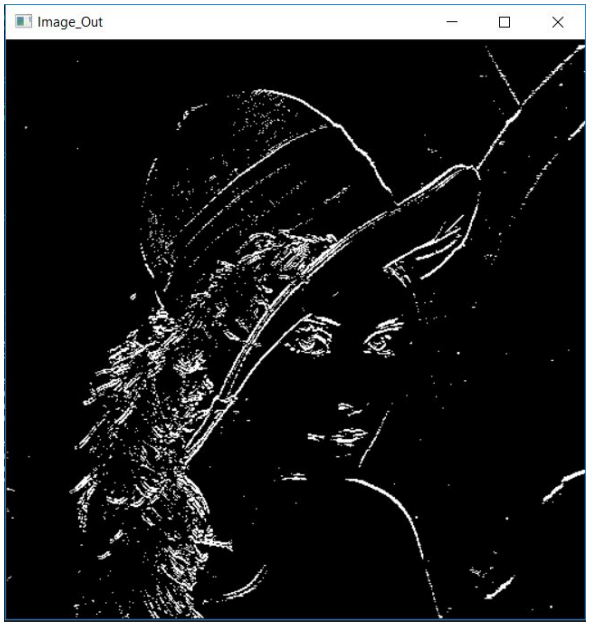
This algorithm is a combined line detection algorithm. This algorithm applies convolution to four 3x3 masks through the entire image- a horizontal line mask, a vertical line mask, a positive 45

degree line mask, and a negative 45 degree line mask. These masks are the same 2D arrays initialized in the previous four algorithms. The result of the convolution is equal to a, b, c, and d respectively for each pixel. This value is compared towards the threshold value. The threshold value is the only value that the user can change in this code. The range is any real number. The lower the threshold value, the more pixels become white. The higher the threshold value, the more pixels become black. In the two outputs on the top of this page, the threshold value of 70 creates a descriptive sketch of the face and hat with white pixels on a black background while the threshold value of 150 creates a crude outline with white pixels on a white background. The threshold of 150 also shows the nature of the combined algorithm as many of the white pixels are streaks in many different directions. Although the input value for the images are always between 0 and 255 inclusive, the threshold value can be any real number, including negative numbers and values greater than 255.

Edge detection

Robert Edge Detection

```
/*  Robert's edge detection algorithm. Convolute the image with the
mask below. Take the absolute value of the result for both of the
convolution. If the sum is greater than the threshold value, the
pixel is white else black.
*/

/*  gx and gy are both masks for the edge detection
*/
    int gx[2][2] = { { -1, 0 },{ 0,1 } };
    int gy[2][2] = { { 0,-1 },{ 1,0 } };

    /*  a and b are the result of the convolution
    */

    int a;
    int b;

    // t is the threshold value. User input should allow it to be changed

    int t = 30;

    for (j = 1; j< height; j++)
        for (k = 1; k < width; k++)
        {
            a = image_in[j - 1][k - 1] * gx[0][0]
                + image_in[j + 0][k - 1] * gx[1][0]
                + image_in[j - 1][k + 0] * gx[0][1]
                + image_in[j + 0][k + 0] * gx[1][1];

            b = image_in[j - 1][k - 1] * gy[0][0]
                + image_in[j + 0][k - 1] * gy[1][0]
                + image_in[j - 1][k + 0] * gy[0][1]
                + image_in[j + 0][k + 0] * gy[1][1];

            if ((abs(a + b)) > t) {
                image_out[j][k] = 255;
            }
            else
            {
                image_out[j][k] = 0;
            }
        }
```



Threshold Value: 30

Threshold Value: 10

This algorithm is a type of edge detection algorithm known as the Robert edge detection algorithm. This algorithm applies convolution to two 2x2 masks at the right through the entire image initialized as gx and gy in the code above. The edge detection algorithm works by approximating the gradient magnitude of the image values in the neighboring area. In other words, the gradient magnitude of the image values is the amount of change in the amplitude values. The result of the convolution is equal to a and b respectively for each pixel. The absolute value of the sum of a and b is compared towards the threshold value. The threshold value is the only value that the user can change in this code. Since the absolute value is a nonnegative value, the threshold range is any nonnegative real number. The lower the threshold value, the more pixels become white. The

−1	0	0	−1
0	1	1	0

Roberts

higher the threshold value, the more pixels become black. In the two outputs on the top of this page, the threshold value of 30 creates a descriptive outline of the face, hat, and some background with white pixels on a black background. The threshold value of 10 fails to create an accurate edge detection algorithm with many white dots. These dots form because of the change in amplitude with the different shades and gradients in the background.

Sobel Horizontal/Vertical and Diagonal Detection

Sobel Horizontal/Vertical Detection

```
/*  Sobels edge detection algorithm. Convolute the image with the
mask below. Take the absolute value of the result for both of the
convolution. If the sum is greater than the threshold value, the
pixel is white else black.
*/

/*  gx and gy are both masks for the edge detection
*/
    int gx[3][3] = { { -1,-2,-1 },{ 0,0,0 },{ 1,2,1 } };
    int gy[3][3] = { { -1, 0, 1 },{ -2,0,2 },{ -1,0,1 } };

    /*  a and b are the result of the convolution
    */

    int a;
    int b;

    // t is the threshold value. User input should allow it to be changed

    int t = 150;

    for (j = 1; j< height - 1; j++)
        for (k = 1; k < width - 1; k++)
        {
            a = image_in[j - 1][k - 1] * gx[0][0]
                + image_in[j + 0][k - 1] * gx[1][0]
                + image_in[j + 1][k - 1] * gx[2][0]
                + image_in[j - 1][k + 0] * gx[0][1]
                + image_in[j + 0][k + 0] * gx[1][1]
                + image_in[j + 1][k + 0] * gx[2][1]
                + image_in[j - 1][k + 1] * gx[0][2]
                + image_in[j + 0][k + 1] * gx[1][2]
                + image_in[j + 1][k + 1] * gx[2][2];

            b = image_in[j - 1][k - 1] * gy[0][0]
                + image_in[j + 0][k - 1] * gy[1][0]
                + image_in[j + 1][k - 1] * gy[2][0]
                + image_in[j - 1][k + 0] * gy[0][1]
                + image_in[j + 0][k + 0] * gy[1][1]
                + image_in[j + 1][k + 0] * gy[2][1]
                + image_in[j - 1][k + 1] * gy[0][2]
                + image_in[j + 0][k + 1] * gy[1][2]
                + image_in[j + 1][k + 1] * gy[2][2];

            if ((abs(a + b)) > t) {
                image_out[j][k] = 255;
            }
            else
            {
                image_out[j][k] = 0;
            }
        }
```

Sobel Diagonal Detection

```
/*  Sobels edge detection algorithm. Convolute the image with the
mask below. Take the absolute value of the result for both of the
convolution. If the sum is greater than the threshold value, the
pixel is white else black.
*/

/*  gx and gy are both masks for the edge detection
*/
    int gx[3][3] = { { 0, 1, 2 },{ -1,0,1 },{ -2,-1,0 } };
    int gy[3][3] = { { -2, -1, 0 },{ -1,0,1 },{ 0,1,2 } };

    /*  a and b are the result of the convolution
    */

    int a;
    int b;

    // t is the threshold value. User input should allow it to be changed

    int t = 100;

    for (j = 1; j< height - 1; j++)
        for (k = 1; k < width - 1; k++)
        {
            a = image_in[j - 1][k - 1] * gx[0][0]
                + image_in[j + 0][k - 1] * gx[1][0]
                + image_in[j + 1][k - 1] * gx[2][0]
                + image_in[j - 1][k + 0] * gx[0][1]
                + image_in[j + 0][k + 0] * gx[1][1]
                + image_in[j + 1][k + 0] * gx[2][1]
                + image_in[j - 1][k + 1] * gx[0][2]
                + image_in[j + 0][k + 1] * gx[1][2]
                + image_in[j + 1][k + 1] * gx[2][2];

            b = image_in[j - 1][k - 1] * gy[0][0]
                + image_in[j + 0][k - 1] * gy[1][0]
                + image_in[j + 1][k - 1] * gy[2][0]
                + image_in[j - 1][k + 0] * gy[0][1]
                + image_in[j + 0][k + 0] * gy[1][1]
                + image_in[j + 1][k + 0] * gy[2][1]
                + image_in[j - 1][k + 1] * gy[0][2]
                + image_in[j + 0][k + 1] * gy[1][2]
                + image_in[j + 1][k + 1] * gy[2][2];

            if ((abs(a + b)) > t) {
                image_out[j][k] = 255;
            }
            else
            {
                image_out[j][k] = 0;
            }
        }
```

Threshold Value: 100 (Horizontal/Vertical)

Threshold Value: 200 (Horizontal/Vertical)

Threshold Value: 100 (Diagonal)

Threshold Value: 200 (Diagonal)

This algorithm is a type of edge detection algorithm known as the Sobel edge detection algorithm. This algorithm applies convolution to two 3x3 masks at the right through the entire image initialized as gx and gy in the code above. The top two are the Horizontal/Vertical Edge detection and the bottom two are the Diagonal Edge Detection. The edge detection algorithm works by approximating the gradient magnitude of the image values in the neighboring area. In other words, the gradient magnitude of the image values is the amount of change in the amplitude values. The result of the convolution is equal to a and b respectively for each pixel. The absolute value of the sum of a and b is compared towards the threshold value. The threshold value is the only value that the user can change in this code. Since the absolute value is a nonnegative value, the threshold range is any nonnegative real number. The lower the threshold value, the more pixels become white. The higher the threshold value, the more pixels become black. There are four outputs in the page before. The top two outputs are with the horizontal/vertical edge detection masks and the bottom two outputs are with the diagonal edge detection masks. Although the masks are different, the result are very similar and the edge detection works fairly well at both amplitudes.

−1	−2	−1	−1	0	1
0	0	0	−2	0	2
1	2	1	−1	0	1

Sobel

0	1	2	−2	−1	0
−1	0	1	−1	0	1
−2	−1	0	0	1	2

Sobel

Prewitt Horizontal and Diagonal Edge Detection

Horizontal Edge Detection

```
/*  Prewitt's edge detection algorithm. Convolute the image with the
mask below. Take the absolute value of the result for both of the
convolution. If the sum is greater than the threshold value, the
pixel is white else black.
*/

/*  gx and gy are both masks for the edge detection
*/
    int gx[3][3] = { { -1,-1,-1 },{ 0,0,0 },{ 1,1,1 } };
    int gy[3][3] = { { -1, 0, 1 },{ -1,0,1 },{ -1,0,1 } };

    /*  a and b are the result of the convolution
    */

    int a;
    int b;

    // t is the threshold value. User input should allow it to be changed

    int t = 70;

    for (j = 1; j< height - 1; j++)
        for (k = 1; k < width - 1; k++)
        {
            a = image_in[j - 1][k - 1] * gx[0][0]
                + image_in[j + 0][k - 1] * gx[1][0]
                + image_in[j + 1][k - 1] * gx[2][0]
                + image_in[j - 1][k + 0] * gx[0][1]
                + image_in[j + 0][k + 0] * gx[1][1]
                + image_in[j + 1][k + 0] * gx[2][1]
                + image_in[j - 1][k + 1] * gx[0][2]
                + image_in[j + 0][k + 1] * gx[1][2]
                + image_in[j + 1][k + 1] * gx[2][2];

            b = image_in[j - 1][k - 1] * gy[0][0]
                + image_in[j + 0][k - 1] * gy[1][0]
                + image_in[j + 1][k - 1] * gy[2][0]
                + image_in[j - 1][k + 0] * gy[0][1]
                + image_in[j + 0][k + 0] * gy[1][1]
                + image_in[j + 1][k + 0] * gy[2][1]
                + image_in[j - 1][k + 1] * gy[0][2]
                + image_in[j + 0][k + 1] * gy[1][2]
                + image_in[j + 1][k + 1] * gy[2][2];

            if ((abs(a) + abs(b)) > t) {
                image_out[j][k] = 255;
            }
            else
            {
                image_out[j][k] = 0;
            }
        }
```

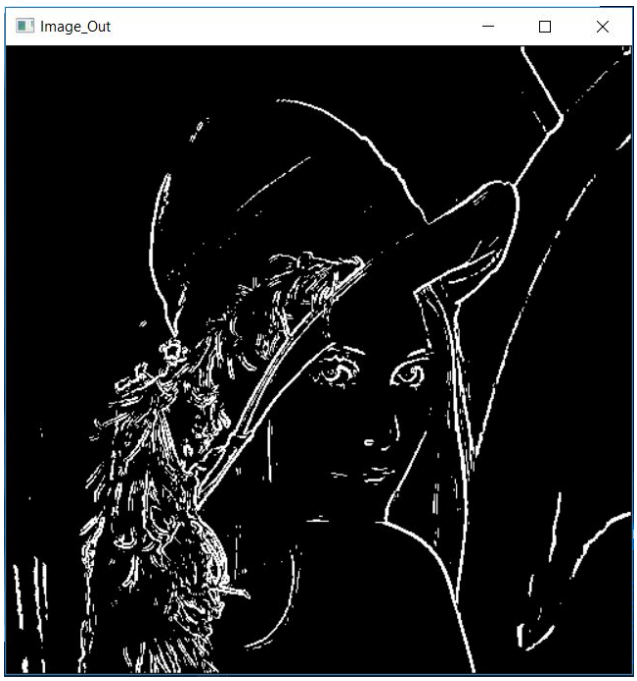
Diagonal Edge Detection

```
/*  Prewitt's edge detection algorithm. Convolute the image with the
mask below. Take the absolute value of the result for both of the
convolution. If the sum is greater than the threshold value, the
pixel is white else black.
*/

/*  gx and gy are both masks for the edge detection
*/
    int gx[3][3] = { { 0,1,1 },{ -1,0,1 },{ -1,-1,0 } };
    int gy[3][3] = { { -1, -1, 0 },{ -1,0,1 },{ 0,1,1 } };

    /*  a and b are the result of the convolution
    */

    int a;
    int b;

    // t is the threshold value. User input should allow it to be changed

    int t = 150;

    for (j = 1; j< height - 1; j++)
        for (k = 1; k < width - 1; k++)
        {
            a = image_in[j - 1][k - 1] * gx[0][0]
                + image_in[j + 0][k - 1] * gx[1][0]
                + image_in[j + 1][k - 1] * gx[2][0]
                + image_in[j - 1][k + 0] * gx[0][1]
                + image_in[j + 0][k + 0] * gx[1][1]
                + image_in[j + 1][k + 0] * gx[2][1]
                + image_in[j - 1][k + 1] * gx[0][2]
                + image_in[j + 0][k + 1] * gx[1][2]
                + image_in[j + 1][k + 1] * gx[2][2];

            b = image_in[j - 1][k - 1] * gy[0][0]
                + image_in[j + 0][k - 1] * gy[1][0]
                + image_in[j + 1][k - 1] * gy[2][0]
                + image_in[j - 1][k + 0] * gy[0][1]
                + image_in[j + 0][k + 0] * gy[1][1]
                + image_in[j + 1][k + 0] * gy[2][1]
                + image_in[j - 1][k + 1] * gy[0][2]
                + image_in[j + 0][k + 1] * gy[1][2]
                + image_in[j + 1][k + 1] * gy[2][2];

            if ((abs(a + b)) > t) {
                image_out[j][k] = 255;
            }
            else
            {
                image_out[j][k] = 0;
            }
        }
```

Threshold Value: 70 (Horizontal/Vertical)

Threshold Value: 150 (Horizontal/Vertical)

Threshold Value: 70 (Diagonal)



Threshold Value: 150 (Diagonal)

This algorithm is a type of edge detection algorithm known as the Prewitt edge detection algorithm. This algorithm applies convolution to two 3x3 masks at the right through the entire image initialized as gx and gy in the code above. The top two are the Horizontal/Vertical Edge detection and the bottom two are the Diagonal Edge Detection. The edge detection algorithm works by approximating the gradient magnitude of the image values in the neighboring area. In other words, the gradient magnitude of the image values is the amount of change in the amplitude values. The result of the convolution is equal to a and b respectively for each pixel. The absolute value of the sum of a and b is compared towards the threshold value. The threshold value is the only value that the user can change in this code. Since the absolute value is a nonnegative value, the threshold range is any nonnegative real number. The lower the threshold value, the more pixels become white. The higher the threshold value, the more pixels become black. There are four outputs in the page before. The top two outputs are with the horizontal/vertical edge detection masks and the bottom two outputs are with the diagonal edge detection masks. Although the masks are different, the result are very similar and the edge detection works fairly well at both amplitudes.

−1	−1	−1	−1	0	1
0	0	0	−1	0	1
1	1	1	−1	0	1

Prewitt

0	1	1	−1	−1	0
−1	0	1	−1	0	1
−1	−1	0	0	1	1

Prewitt

Image segmentation

The below functions apply image segmentation, partitioning the image into multiple segments and then applying different filters to each segment.

Global thresholding

This function first finds a global threshold value. It does this by first taking an initial threshold of half the max pixel value (128) and then finding a new threshold value. This is done by counting the number of pixels below and above the initial threshold and then finding the average. This process repeats until the difference between the initial and new threshold is less than a given parameter (5 in this example). It then displays any pixels below the threshold as black and any above as white. In this example, it takes two iterations to find the global threshold.

```
//Global Thresholder

    int global_t = 256 / 2;  //global threshold (middle intensity)
    int initial_t = 0;  //initial threshold
    int lower_sum = 0;  //sum of intensities under initial_t
    int upper_sum = 0;  //sum of intensities over initial_t
    int lower_amount = 0;  //amount of pixels w/ intensities under initial_t
    int upper_amount = 0;  //amount of pixels w/ intensities over initial_t
    int lower_average = 0;  //average intensity under initial_t
    int upper_average = 0;  //average intensity over initial_t

    do
    {
        initial_t = global_t;  //makes the initial_t the value of global_t (initially 128)

        for (j = 0; j < height; j++)
            for (k = 0; k < width; k++)  //scans thru image
            {
                if (image_in[j][k] < initial_t)  //intensities under initial_t
                {
                    lower_sum = lower_sum + image_in[j][k];
                    lower_amount++;
                }
                else if (image_in[j][k] >= initial_t)  //intensities over initial_t
                {
                    upper_sum = upper_sum + image_in[j][k];
                    upper_amount++;
                }
            }
```

```
        lower_average = (lower_sum / lower_amount);
        upper_average = (upper_sum / upper_amount);

        global_t = (lower_average + upper_average) / 2;  //new global intensity

        printf("global_t is now %d and initial_t is %d\n", global_t, initial_t);

    } while (abs(initial_t - global_t) >= 5);  //repeats until difference is less than 5

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            if (image_in[j][k] < global_t)  //intensities under global_t
            {
                image_out[j][k] = 0;  //black
            }
            else if (image_in[j][k] >= global_t)  //intensities over global_t
            {
                image_out[j][k] = 255;  //white
            }
        }
```

```
global_t is now 122 and initial_t is 128
global_t is now 121 and initial_t is 122
```

Adaptive thresholding

Code

```
/*  tf is the calculated threshold value after the pass (t i+1)
ti is the initial threshold value before the pass( t i)
sum3 and count3 help compute the average pixel value of the distribution less than the threshold
sum4 and count4 help compute the average pixel value of the distribution more than the threshold
*/
int num = 32;
if (height % num != 0 || width % num != 0)
{
    printf("Error");
}
else
{
    for (int i = 0; i < num; i++) {
        for (int h = 0; h < num; h++) {
            for (j = i * height / num; j < (i + 1)*height / num; j++) {
                for (k = h * width / num; k < (h + 1) * width / num; k++) {
                    float tf = 200.0;
                    float ti = 0.0;
                    int sum3 = 0;
                    int count3 = 0;
                    int sum4 = 0;
                    int count4 = 0;
                    /*  I used a do while loop to determine the average
                    pixel value of both distributions below and above the
                    threshold.
                    */
```

```
do
{
    ti = tf;    //  Assign the final value from the previous iteration to the value for the current iteration

                /*  This double for loop parses through the image and appends sum and count
                values for the right distribution.
                */
    for (j = i * height / num; j < (i + 1)*height / num; j++) {
        for (k = h * width / num; k < (h + 1) * width / num; k++) {
            if (image_in[j][k] < ti) {
                sum3 += image_in[j][k];
                count3++;
            }
            if (image_in[j][k] >= ti) {
                sum4 += image_in[j][k];
                count4++;
            }
        }
    }

    if (count4 == 0) {
        tf = sum3 / count3;
        printf("%f and %f\n", tf, ti);
    } else if (count3 == 0) {
        tf = sum4 / count4;
        printf("%f and %f\n", tf, ti);
    }
    else {
        tf = (sum3 / count3 + sum4 / count4) / 2;
        printf("%f and %f\n", tf, ti);
    }
    /* Calculates new threshold value. The print statement is not needed but is just a
    check for the output for each iteration of the loop.
    */


    /*  Resets all sum and count values for each partition
    */

    sum3 = 0;
    count3 = 0;
    sum4 = 0;
    count4 = 0;
} while (abs(tf - ti) >= 1.0);
```

```
printf("Final Threshold value: %f\n", tf);

/*  Iterates through output image with final threshold value and
produces the two level binary image.
*/

for (j = i * height / num; j < (i + 1)*height / num; j++)
    for (k = h * width / num; k < (h + 1) * width / num; k++)
    {
        if (image_in[j][k] < tf) {
            image_out[j][k] = 0;
        }
        if (image_in[j][k] >= tf) {
            image_out[j][k] = 255;
        }
    }
}
}
}
}
}
```

Output with Different Values for Num

Num Value = 1

Num Value = 2

Num Value = 4

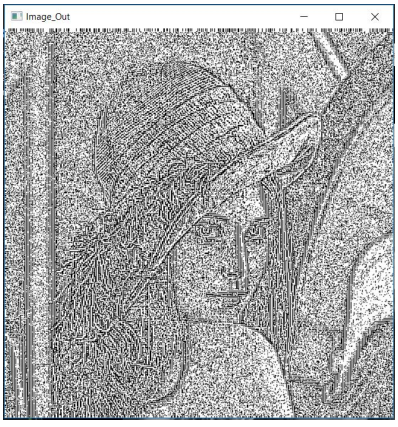
Num Value = 8

Num Value = 16

Num Value = 32

Num Value = 64

Num Value = 128



Num Value = 256

This algorithm is the adaptive thresholding algorithm. In this algorithm, the user inputs the number of regions to divide the one side of the image into. The total number of regions is equal to the number squared. For the output images above, the values chosen were divisible by 512 which gave equally spaced region. For values not divisible by 512, the last column and row would be a different sized region than everything else to include the remainder. The value of the number has to be less than the length of the row and column of the image. For each region, the algorithm is identical to the global thresholding algorithm. In the previous page, nine outputs are displayed from num value = 1 which is identical to the global thresholding output to num value = 256 in multiples of 2. As the value of num increases, the output image is less sensitive to local illumination.

Geometric Transformations

The following functions apply various geometric transformations to the input image that basically displace the pixels without actually changing their intensities.

Translation

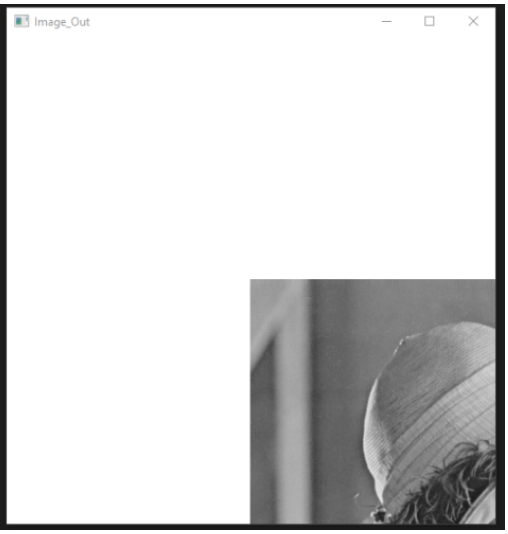
These functions translate the image, essentially shifting it either up, down, left, or right depending on the given parameters. Depending on the function used the rest of the image will either be black or white. In the examples below, the parameters used are half the max width and height of the image. 255 pixels. This makes the image shift half way in each direction.

Right/Down (Black Background)

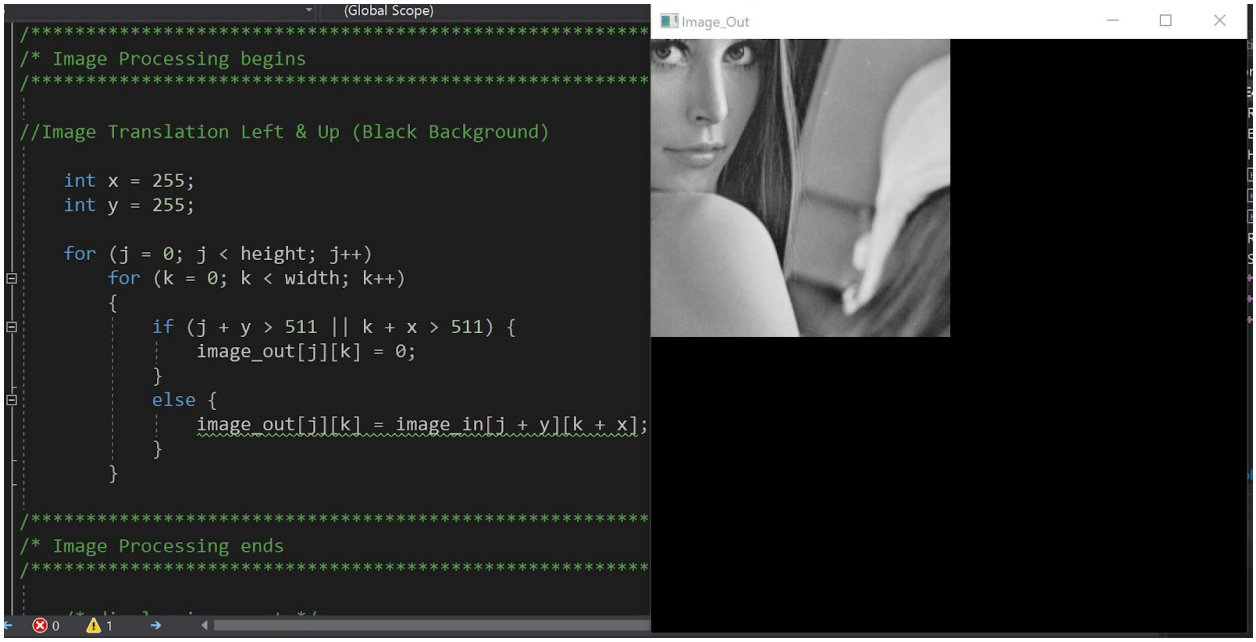
```
//Image Translation (255 pixels Right & Down (Black))

    int x = 255;
    int y = 255;

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            if (j - y < 0 || k - x < 0) {
                image_out[j][k] = 0;  //black pixels
            }
            else {
                //shifted pixels
                image_out[j][k] = image_in[j - y][k - x];
            }
        }
```

Right/Down (White Background)

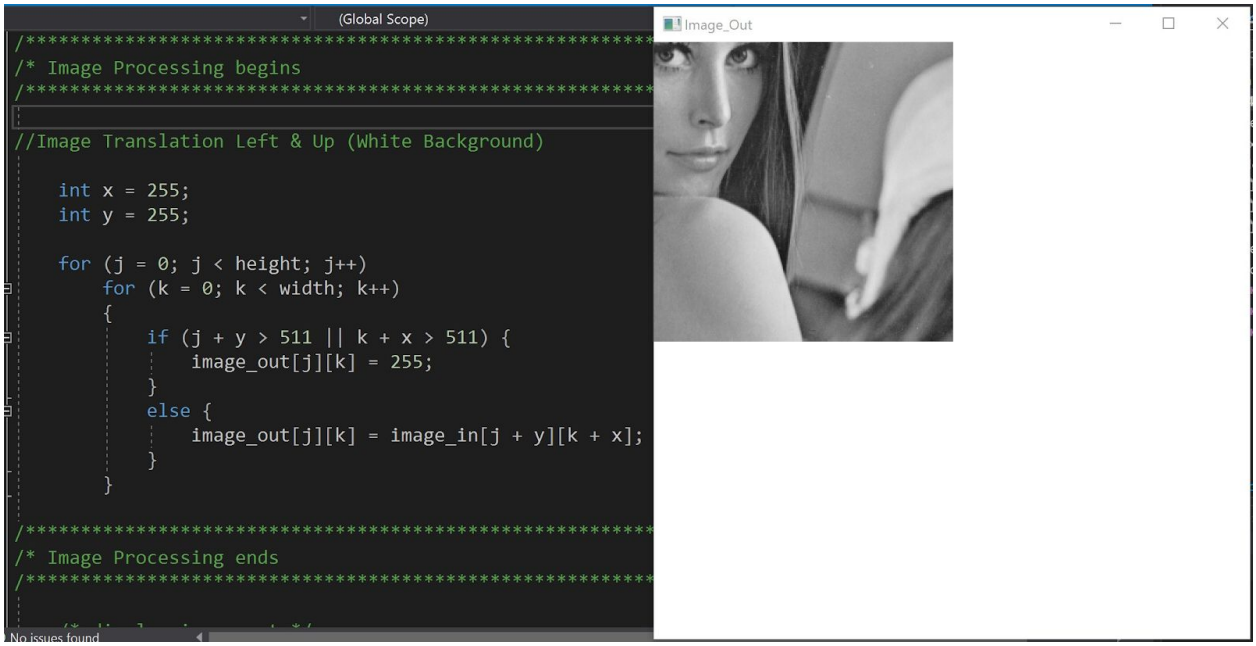
```
//Image Translation (255 pixels Right & Down (Black))

    int x = 255;
    int y = 255;

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            if (j - y < 0 || k - x < 0) {
                image_out[j][k] = 255;  //white pixels
            }
            else {
                //shifted pixels
                image_out[j][k] = image_in[j - y][k - x];
            }
        }
```



Left/Up (Black background)



Left/Up (White background)



Rotation

The three functions below rotate the image. The image can either be rotated 90 degrees clockwise or counter-clockwise, with multiple rotations resulting in rotations of 180, 270, 360, and so on (increasing by 90 degrees each time). The third function is a 180 degree rotation which basically combines two 90 degree rotations. There is no 270 degrees rotation because this can be simply accomplished by a 90 degree rotation in the opposite direction.

90 degree CW

```
/*************************************************************
/* Image Processing begins
/*************************************************************


//Image Rotation (90 degree CW)

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            // Rotated Image 90 degrees CW
            image_out[j][k] = image_in[height - 1 - k][j];
        }


/*************************************************************
/* Image Processing ends
/*************************************************************
```

90 degree CCW

```
/*************************************************
/* Image Processing begins
/*************************************************


//Image Rotation (90 degree CCW)

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            // Rotated Image 90 degrees CCW
            image_out[j][k] = image_in[k][width - 1 -
        }


/*************************************************
/* Image Processing ends
/*************************************************
```

180 degree

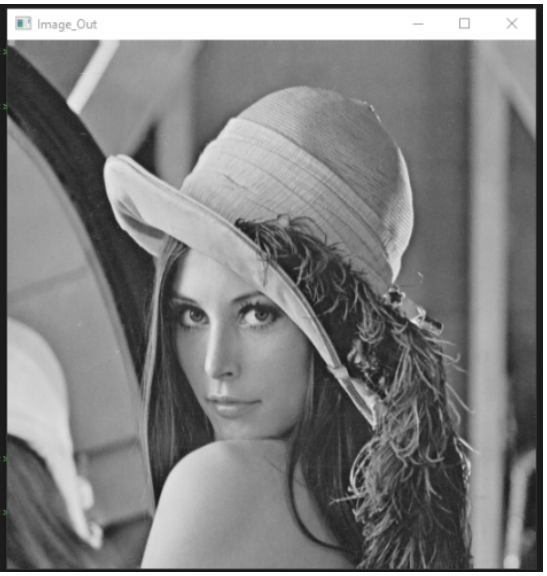
```
/*****************************************************************/
/* Image Processing begins                                       */
/*****************************************************************/


//Image Rotation (180 degree)

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            // Rotated Image 180 degrees
            image_out[j][k] = image_in[height - 1 - j][height - 1 - k];
        }


/*****************************************************************/
/* Image Processing ends                                         */
/*****************************************************************/
```

Reflection

The three functions below reflect the image about the x-axis, the y-axis, or both. This is accomplished by either keeping the height of each pixel constant but mirroring the width, keeping the width of each pixel constant but mirroring the height, or by mirroring both the height and the width.

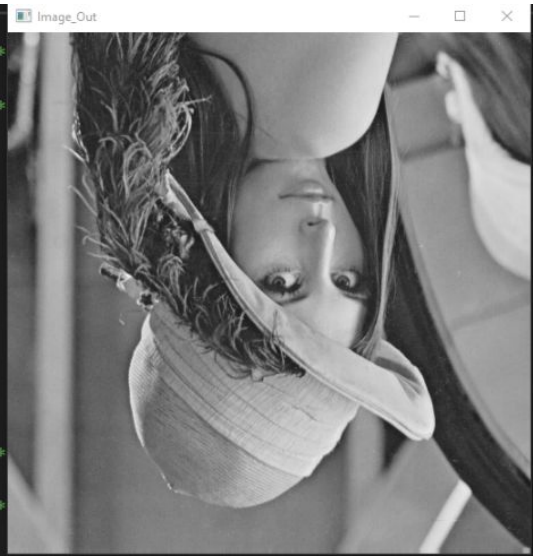
Horizontally

```
/*************************************************************
/* Image Processing begins
/*************************************************************


//Image Reflection (Horizontally)

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            // Mirror pixel horizontally
            image_out[j][k] = image_in[j][width - 1 - k];
        }


/*************************************************************
/* Image Processing ends
/*************************************************************

    /* display image out */
```



Vertically

```
/*************************************************************
/* Image Processing begins
/*************************************************************


//Image Reflection (Vertically)

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            // Mirror pixel vertically
            image_out[j][k] = image_in[height - 1 - j][k];
        }


/*************************************************************
/* Image Processing ends
/*************************************************************
```



Diagonal

```
/*****************************************************************/
/* Image Processing begins                                       */
/*****************************************************************/


//Image Reflection (Horizontally & Vertically)

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            // Mirror pixel
            //Horizontal combined with vertical reflection
            image_out[j][k] = image_in[height - 1 - j][width - 1 - k];
        }


/*****************************************************************/
/* Image Processing ends                                         */
/*****************************************************************/
```

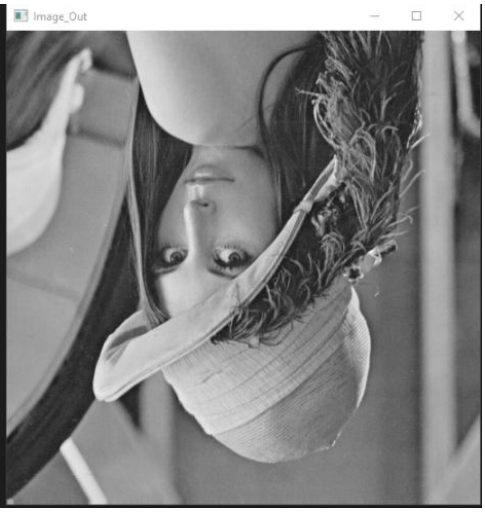
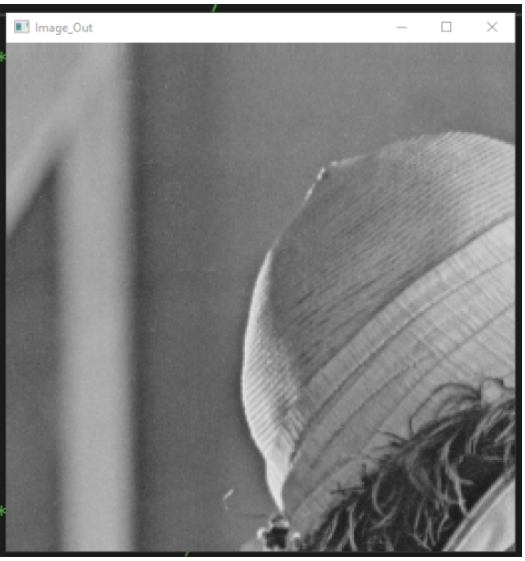


Image Zooming

The four functions below allow for zooming in. This is done by inserting a parameter for the x and y (horizontal and vertical) amount desired to be zoomed, and dividing each pixel by these values. Since only integer values (and no decimal values) are being used, this results in each pixel essentially appearing twice in each direction (in this example, since x and y are both 2), which produces the zooming effect. Increasing the values of x and/or y will increase the amount of zoom.

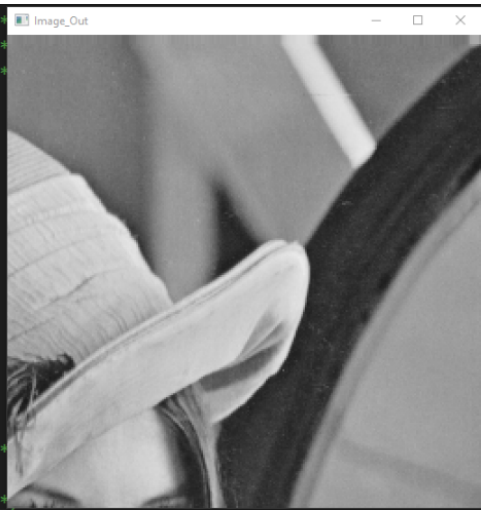
Top Left

```
/* Image Processing begins
/*****************************************************



//Image Zooming (Top Left Corner)

    int x = 2;
    int y = 2;

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            image_out[j][k] = image_in[j / x][k / y];
        }


/*****************************************************
/* Image Processing ends
```



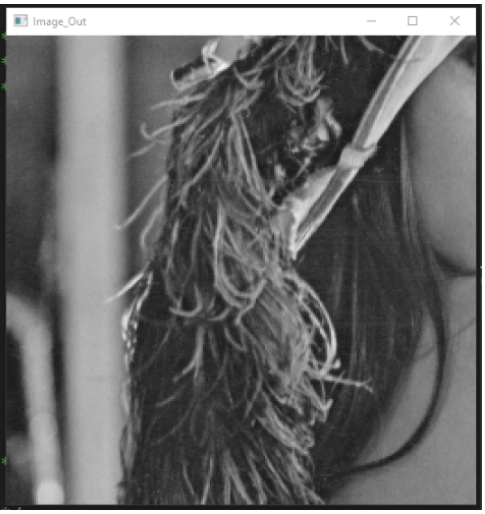
Top Right

```
/*****************************************************************
/* Image Processing begins                                       *
/*****************************************************************


//Image Zooming (BTop Right Corner)

    int x = 2;
    int y = 2;

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            image_out[j][k] = image_in[j / x][(k - 1 + height) / y];
        }


/*****************************************************************
/* Image Processing ends                                        */
/*****************************************************************
```



Bottom Left

```
/*****************************************************************
/* Image Processing begins                                       *
/*****************************************************************


//Image Zooming (Bottom Left Corner)

    int x = 2;
    int y = 2;

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            image_out[j][k] = image_in[(j - 1 + height) / x][k / y];
        }


/*****************************************************************
/* Image Processing ends                                        */
/*****************************************************************
```



Bottom Right

```
/*****************************************************************/
/* Image Processing begins                                       */
/*****************************************************************/


//Image Zooming (Bottom Right Corner)

    int x = 2;
    int y = 2;

    for (j = 0; j < height; j++)
        for (k = 0; k < width; k++)
        {
            image_out[j][k] = image_in[(j - 1 + height) / x][(k - 1 + height) / y];
        }


/*****************************************************************/
/* Image Processing ends                                         */
/*****************************************************************/

    /* display image out */
```

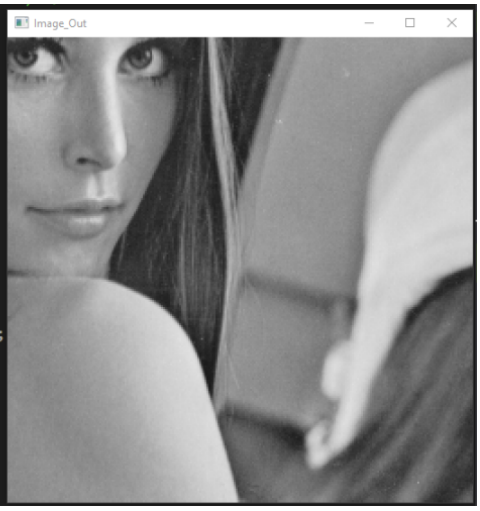


Image Scaling

The following functions apply scaling operations to the input image, which change the size of the image. This is done by down-sampling the image - using one value to represent a block of pixels.

Down-Sampling (Center Pixel)

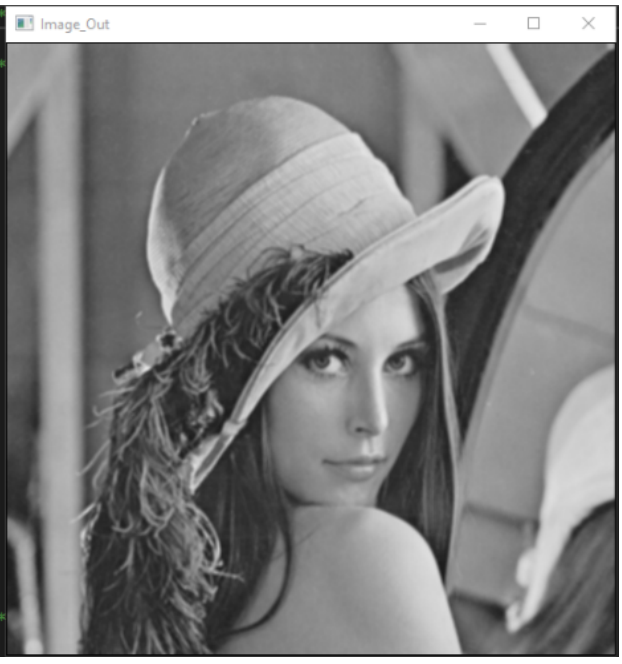
The following function down-samples the image with 3 x 3 blocks. It takes the value of the center block and applies it to the rest of the 8 pixels in the block. This produces a more pixelated image as seen below.

```
/*************************************************************
/* Image Processing begins
/*************************************************************


//Image Down-Sampling (Center Pixel)

    for (j = 1; j < height - 1; j += 3)
        for (k = 1; k < width - 1; k += 3)
        {
            image_out[j - 1][k - 1] = (int)(image_in[j][k]);
            image_out[j - 1][k] = (int)(image_in[j][k]);
            image_out[j - 1][k + 1] = (int)(image_in[j][k]);
            image_out[j][k - 1] = (int)(image_in[j][k]);
            image_out[j][k] = (int)(image_in[j][k]);
            image_out[j][k + 1] = (int)(image_in[j][k]);
            image_out[j + 1][k - 1] = (int)(image_in[j][k]);
            image_out[j + 1][k] = (int)(image_in[j][k]);
            image_out[j + 1][k + 1] = (int)(image_in[j][k]);

        }


/*************************************************************
/* Image Processing ends
```

Down-Sampling (Average)

This function is another method of down-sampling the image. In this case, the average value of the 9 pixels in a 3 x 3 block is taken and applied to the 9 pixels in the block.

```
/*************************************************************
/* Image Processing begins
/*************************************************************


//Image Down-Sampling (Average)

    for (j = 1; j < height - 1; j++) {
        for (k = 1; k < width - 1; k++)

            //Convolution
            (image_out[j][k]) = (int)(image_in[j - 1][k - 1]
                + image_in[j - 1][k]
                + image_in[j - 1][k + 1]
                + image_in[j][k - 1]
                + image_in[j][k]
                + image_in[j][k + 1]
                + image_in[j + 1][k - 1]
                + image_in[j + 1][k]
                + image_in[j + 1][k + 1]) / 9;

    }


/*************************************************************
/* Image Processing ends
```

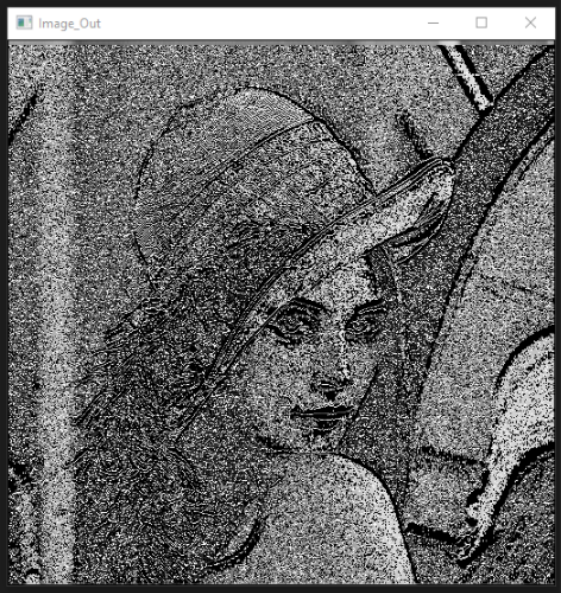


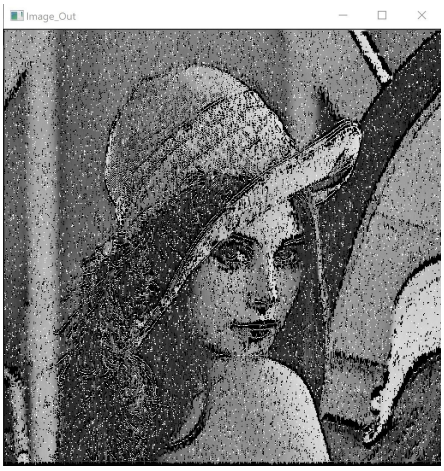
Adding Noise

Salt and Pepper Noise

This function adds severe salt and pepper noise to the input image. It does this using a similar function to the median filter, however instead of eliminating noise, it makes many of the pixels either white (255) or black (0). It was made to see how well the medium filter works, which eliminates noise.

```
// Salt pepper noise

   int array[3] = { };

   int a;
   int b;
   int c;

   for (j = 1; j < height - 1; j++)
       for (k = 1; k < width - 1; k++)
       {
           int a = image_in[j - 1][k]; //a = first pixel
           int b = image_in[  j  ][k]; //b = second pixel
           int c = image_in[j + 1][k]; //c = third pixel

           //if b is median, black pixel
           if ((a < b && b < c) || (c < b & b < a))
               image_out[j][k] = 0;

           //if a is median, white pixel
           else if ((b < a && a < c) || (c < a && a < b))
               image_out[j][k] = 255;

           //if c is median, displays c
           else
               image_out[j][k] = c;

       }
```





Here is the image after applying the medium filter several times to remove some noise.

Conclusion

As a result of this project, the team was able to develop a deeper and richer understanding on image processing, specifically how code can be used to manipulate the pixels of an image, to either move them or change their intensities. The team was able to accomplish this by creating an image enhancement toolkit, which is essence consisted of a multitude of different functions that can alter an image in different ways, from changing the pixel intensities, to brightening and darkening the image, to even performing geometric transformations.

The way to team decided to split the project up was by who would do each of the functions. Asif agreed to do the arithmetic operations, intensity transformations, feature detection, and image segmentation. Tim agreed to do the filters, geometric transformations, image zooming, and image scaling. Next to each function at the beginning of the report are the initials of the team member that worked on that function.