Image Enhancement Toolkit

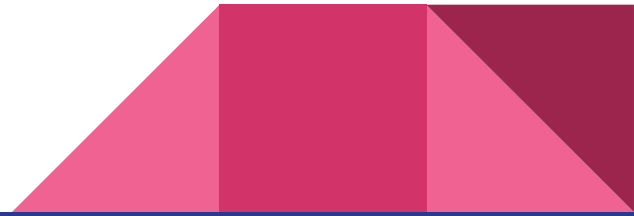
CPE462 - Tim Demetriades & Asif Uddin

Enhancements

- The following slides show all the enhancements created by the team
- Included is a brief description followed by the code and output image
- The input image for all of the examples is lenna.512 found below

Arithmetic Operations

These 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 creates the negative of the input image, which essentially is inverting the pixel values.

```
/*****************************************************
/* 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.

```
//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;
        }
```

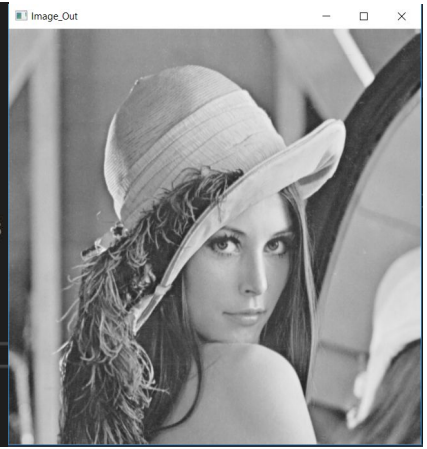
Intensity transformation

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

Logarithm

This function takes the logarithm with base 2 of the input image, which makes the new image. The logarithm transformation always lightens the image.

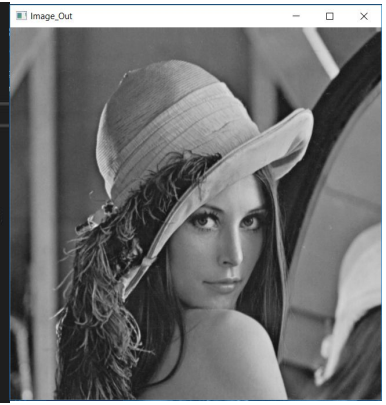
```
/*****************************************************************/
/* Image Processing begins                                       */
/*****************************************************************/

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



Inverse Logarithm

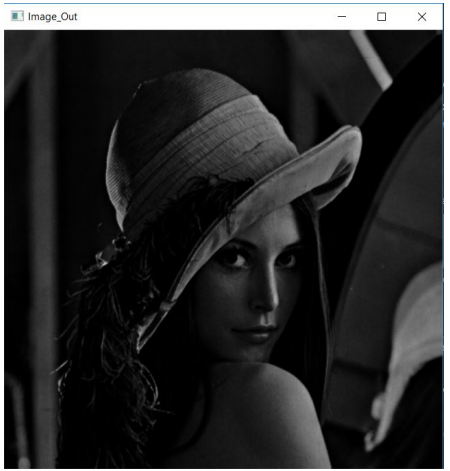
This algorithm is an intensity inverse logarithm transformation.The inverse logarithm transformation always darkens the image.

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

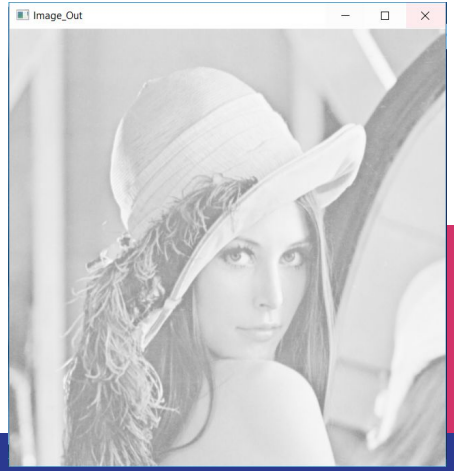


Power and Root Transform

This algorithm is an intensity power and root transformation. The root transformation lightens the image while the power transformation darkens the image.




```
// 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;
        }
```

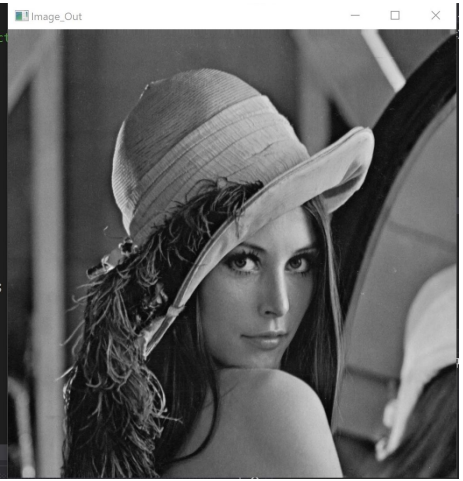


Gamma correction

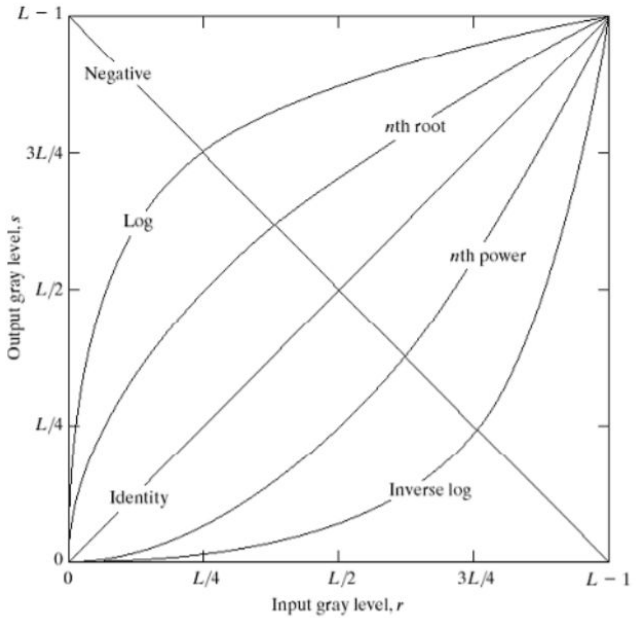
Gamma correction is used for accurately displaying images on a computer screen and controls the overall brightness of the image. It is similar to the power/root transforms. It functions by pre-distorting the image to make the displayed image look like the original. Here is an example with a gamma value of 1.5

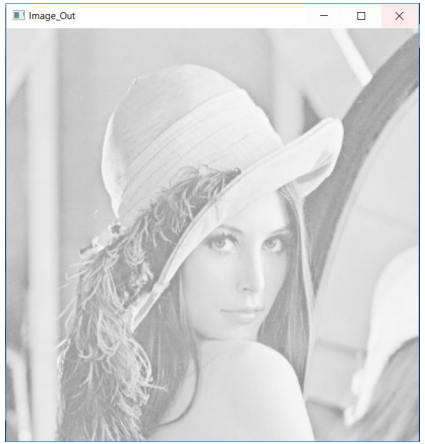
```
    CImgDisplay disp_in(image_disp,"Image_In",0);
    /* CImgDisplay display_name(image_displayed, "window title", normalization_fac

/*****************************************************************/
/* 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

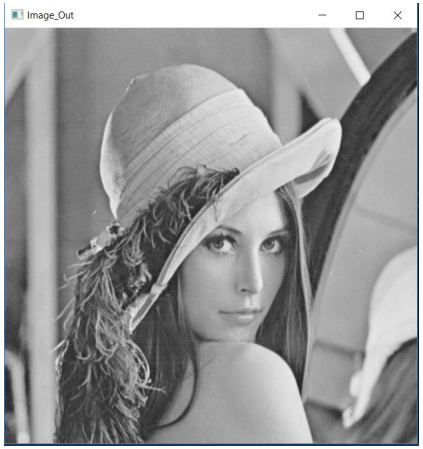


Comparison of Intensity Transformations

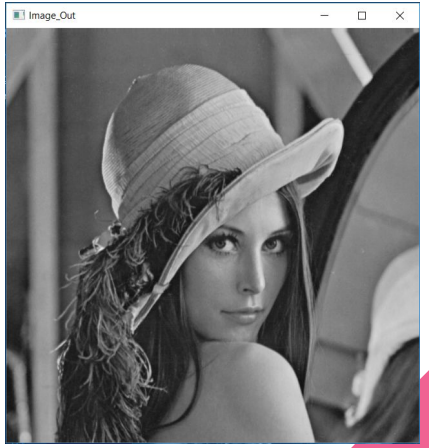




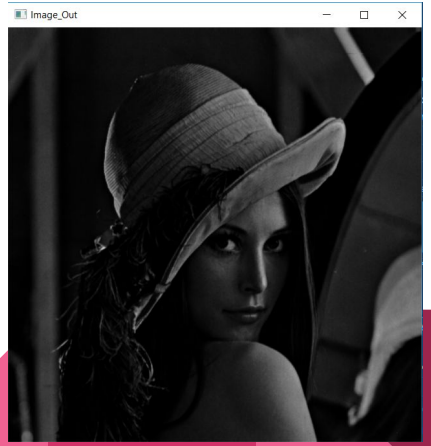
Root transformation



Logarithm

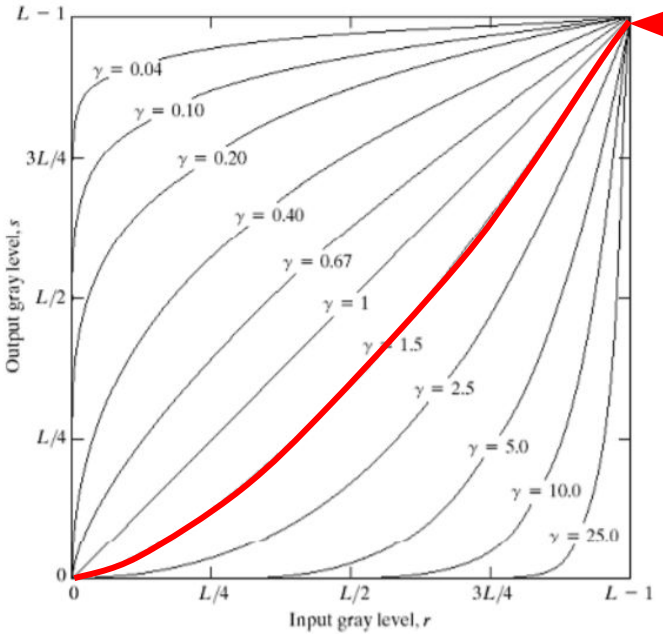


Inverse Logarithm

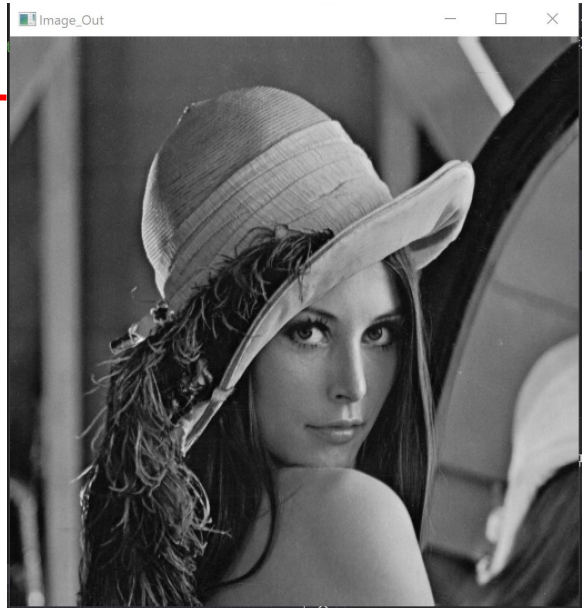


Power Transformation

Plot versus Image



$s = c \cdot r^{\gamma}$ ($c = 1$ in the plot)

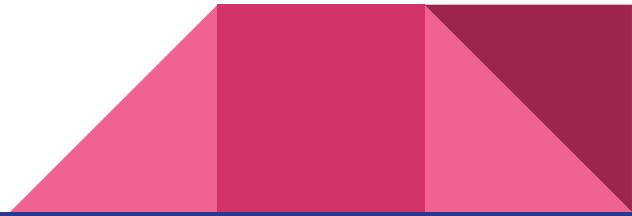


Gamma of 1.5

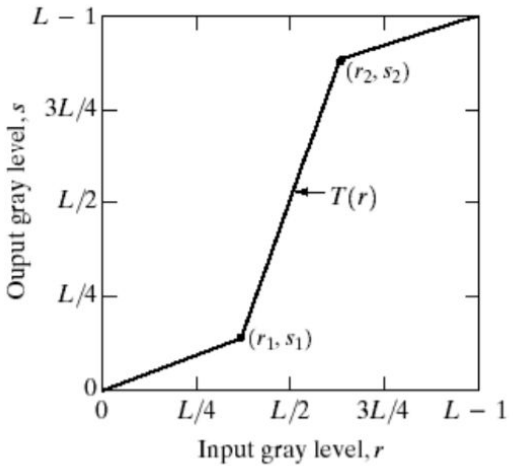
Constant of 1

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.

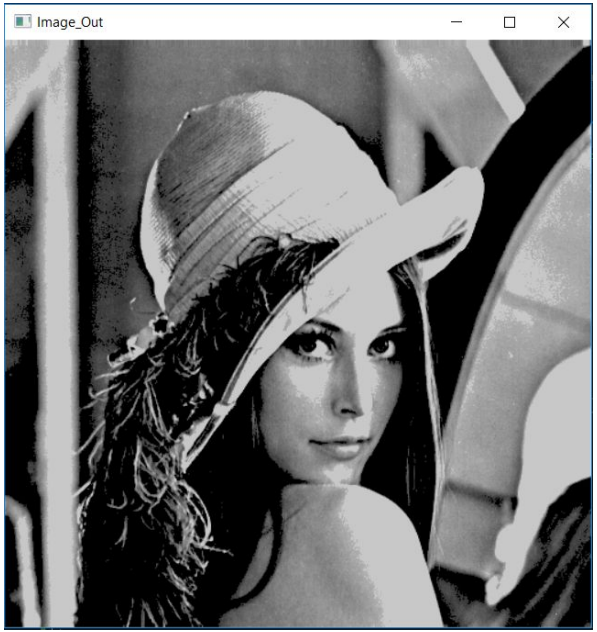


Contrast Stretching Function



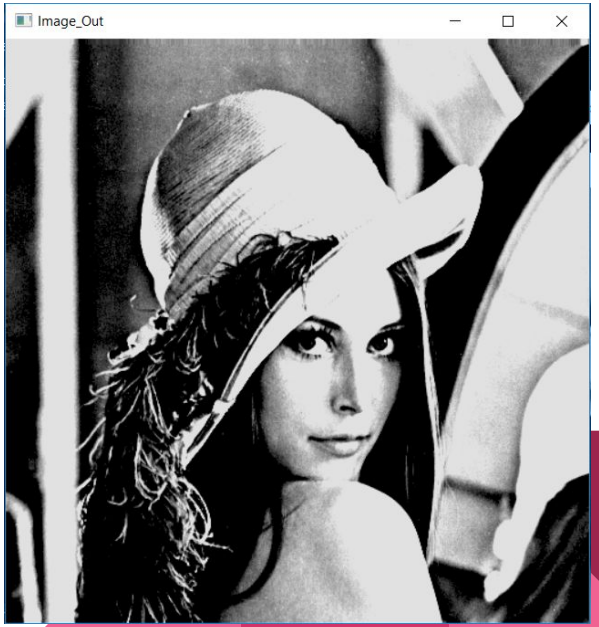
r1 = 96, r2 = 160

s1 = 50, s2 = 200



r1 = 100, r2 = 150

s1 = 25, s2 = 225



Histogram equalization

This function applies a histogram equalization, which increases the contrast or difference between intensities of the image.

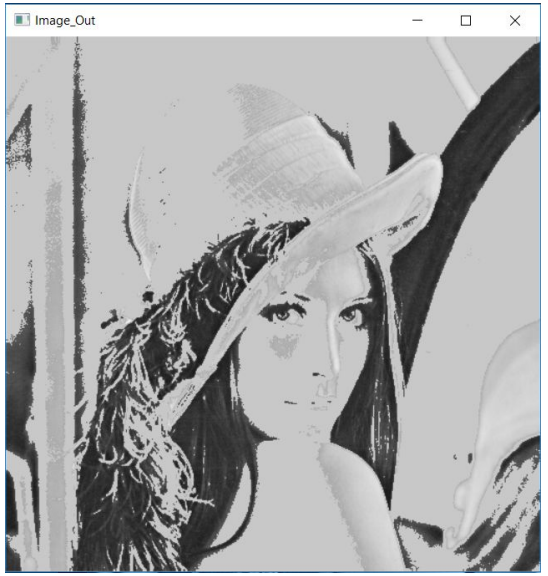
```
//Histogram Equilization (fixes contrast)

    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]];

    }
```

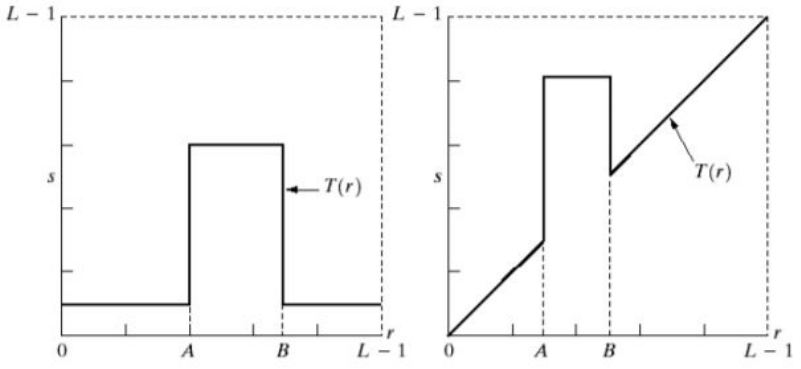
Gray level slicing

Image on Left is Gray Level Slicing without Background

Image on Right is Gray Level Slicing With Background

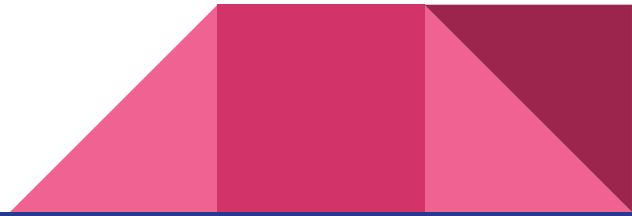
Regions A and B are between input values 90 and 170.





Filters

The below functions apply filters to the input image, altering the overall look of the image. It can be used to modify or enhance and image.



Smoothing / Low pass filters

This function applies a low pass filter to the input image, effectively smoothing.

```
W6                                                    (Global Scope)
/* 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
```

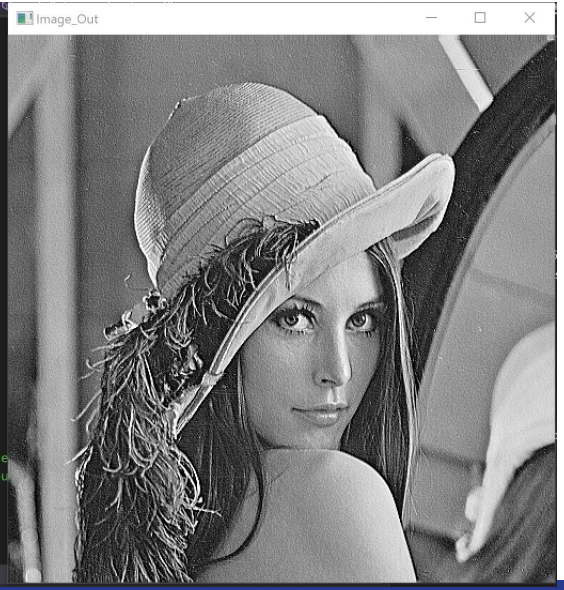
$$\frac{1}{9} \times \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

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.

```
/* 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



0	−1	0
−1	5	−1
0	−1	0

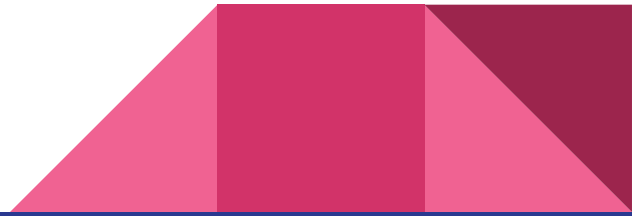
Median filters

The function of this filter is to smooth the input image and eliminate noise.

```
// 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;

       }
```

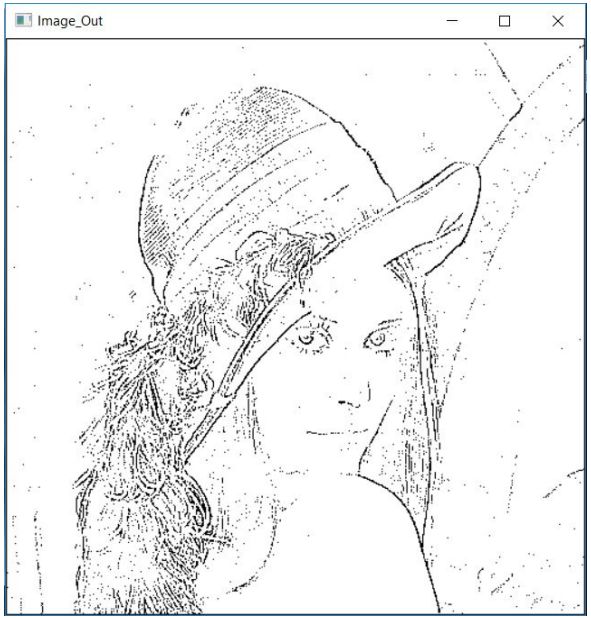
List of Feature detection

- Point
- Line
  - Horizontal
  - Vertical
  - Positive 45
  - Negative 45
- Edge
  - Robert
  - Sobel
    - Horizontal
    - Vertical
  - Prewitt
    - Horizontal
    - Vertical



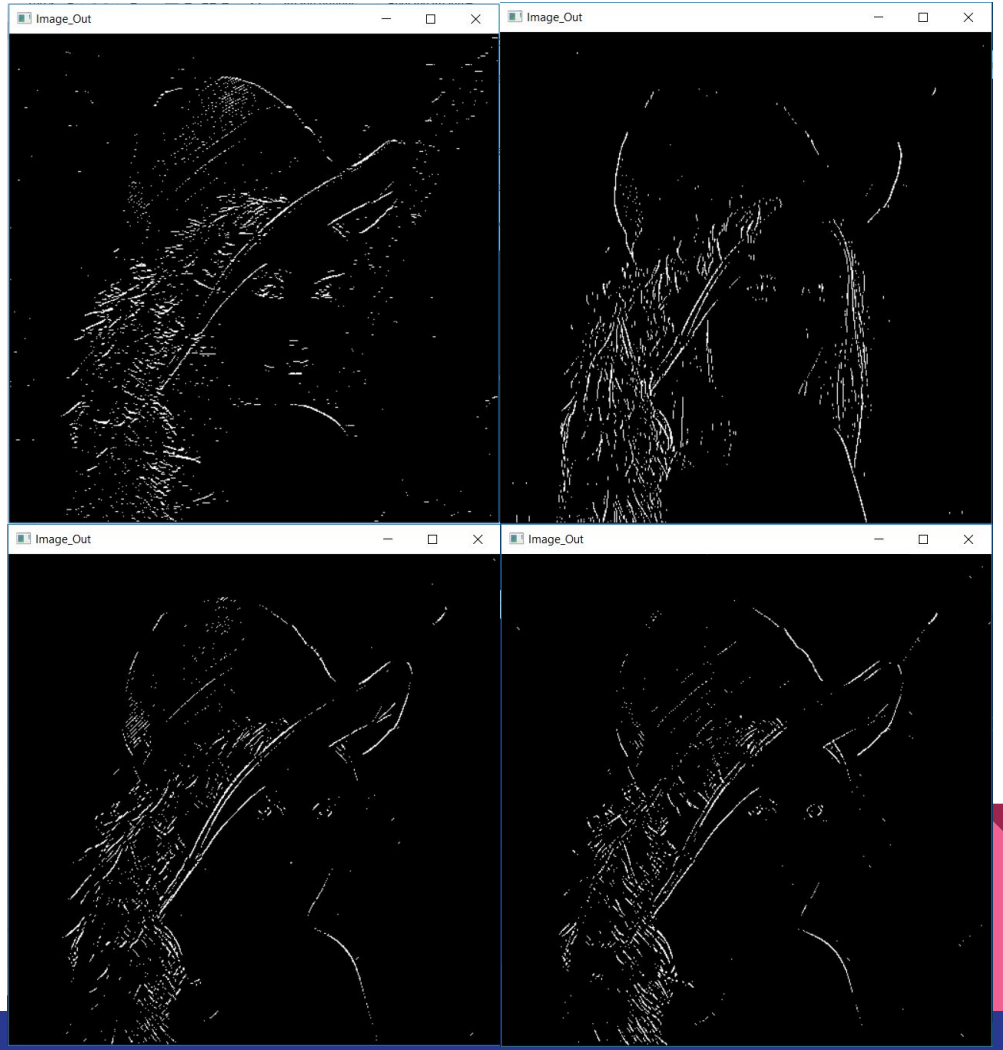
Point detection

Identifies points of local maximum and local minimum in the image



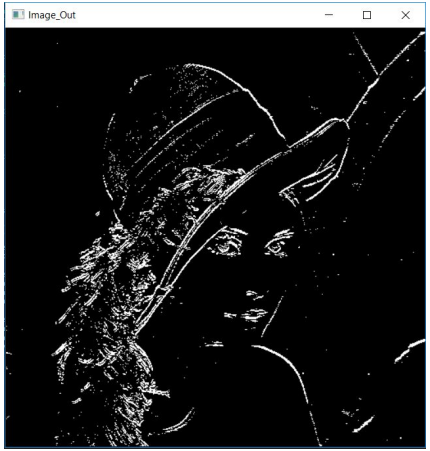
−1	−1	−1
−1	8	−1
−1	−1	−1

Line detection

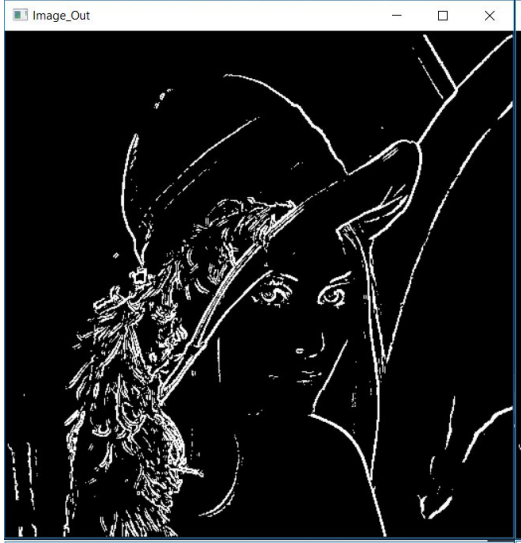


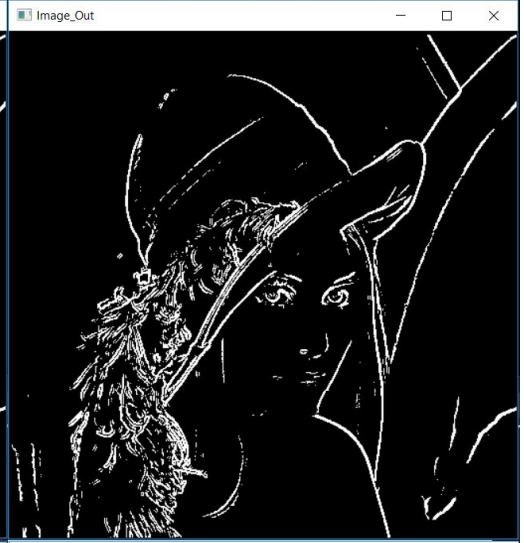
Edge detection

Robert

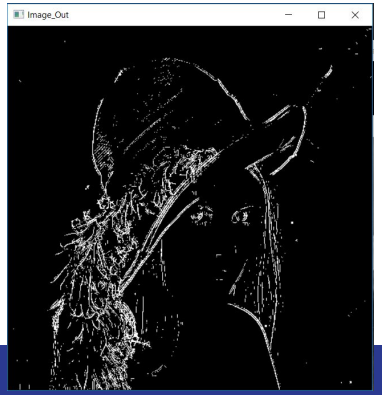


Sobel

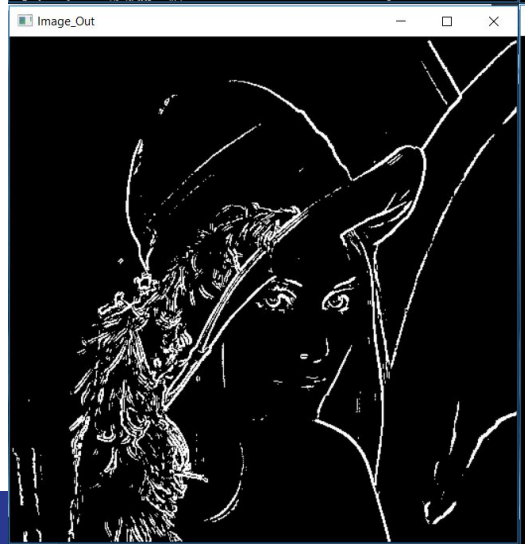




Combined Line Detection



Prewitt



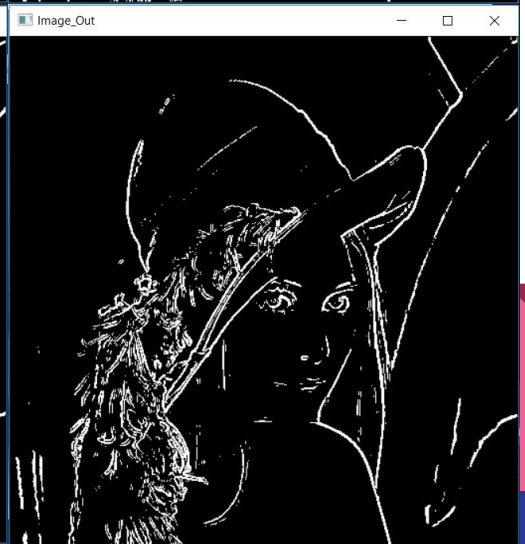
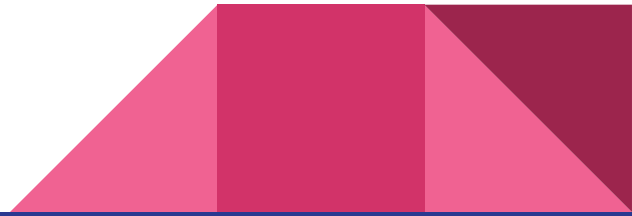


Image segmentation

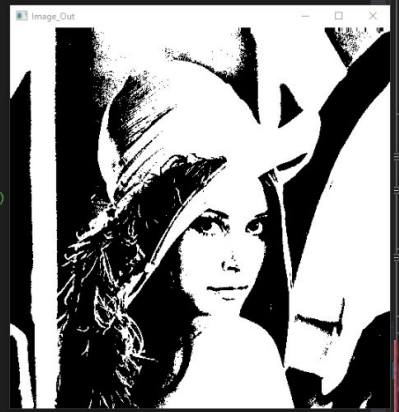
The below functions apply image segmentation, partitioning the image into multiple segments and then applying different filters to each segment.



Global thresholding

This function first finds a global threshold value, and then outputs a black and white image based on the threshold found.

```
//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

Num Value = 2

Num Value = 4

Num Value = 64

Num Value = 256

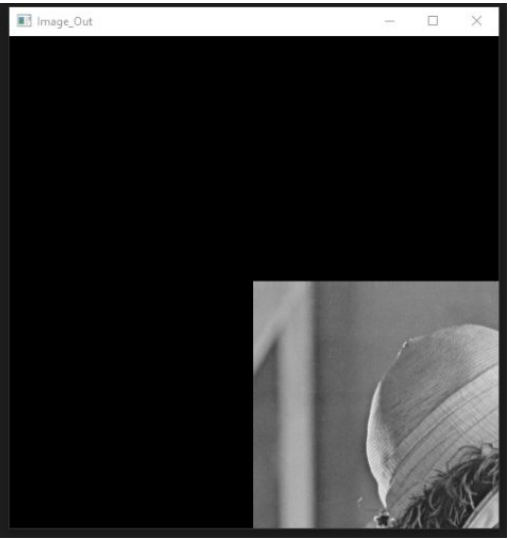
Geometric Transformations

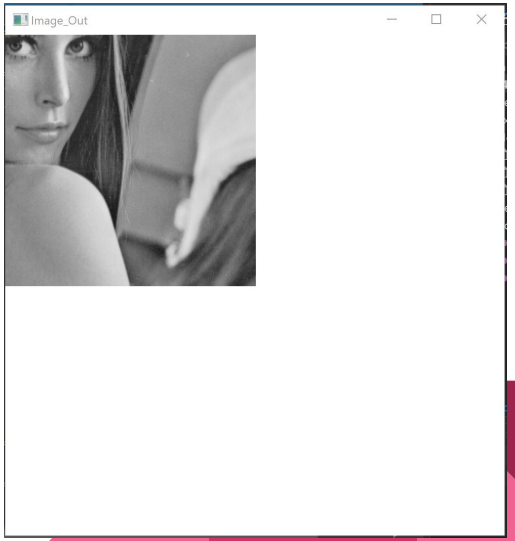
The following functions apply various geometric transformations to the input image that basically move around the pixels without actually changing their intensities.

Translation

These functions translate the image, essentially shifting it either up, down, left, or right depending on the given parameters.

```
//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];
            }
        }
```





Rotation

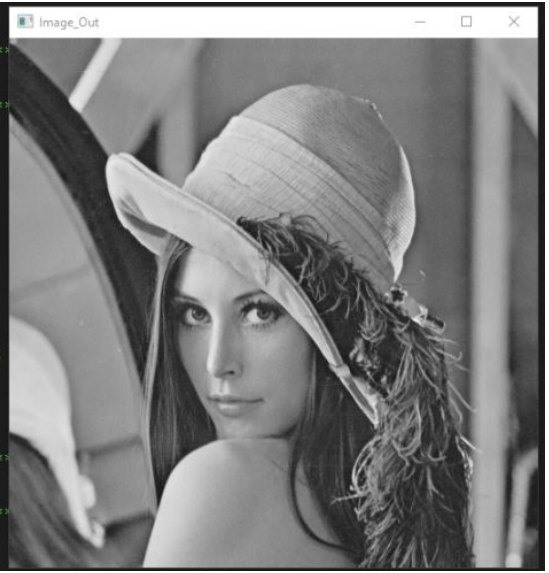
The three functions for this rotate the image. The image can either be rotated 90 degrees clockwise, 90 degrees counterclockwise, or 180 degrees.

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

Reflection

These three functions will reflect the image about the x-axis, the y-axis, or both.

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



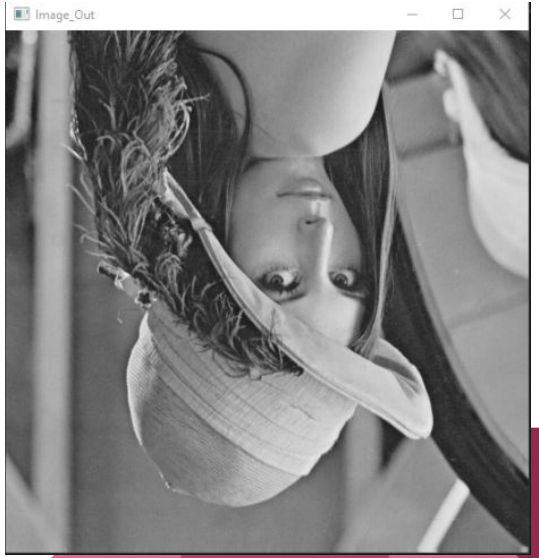
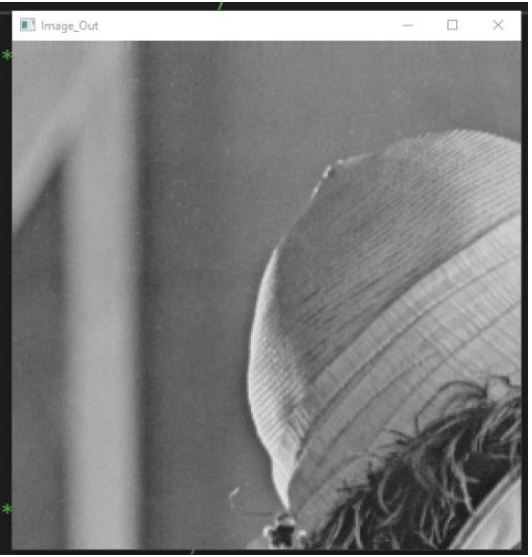
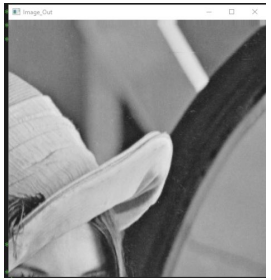


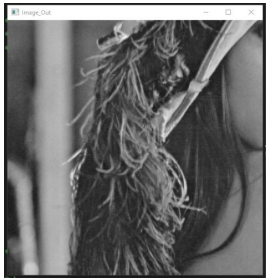
Image Zooming

There are four functions that allow for zooming into the image, each function zooming into one of the four corners of the image.

```
/* 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
```







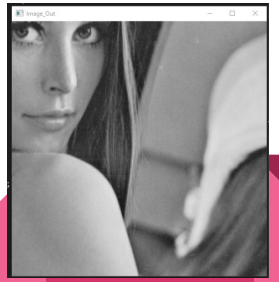
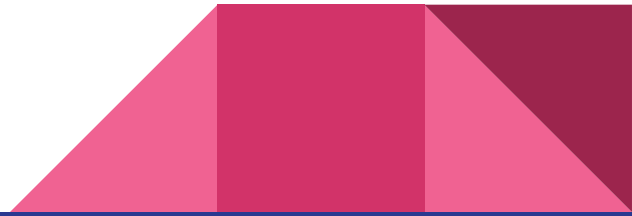


Image Scaling

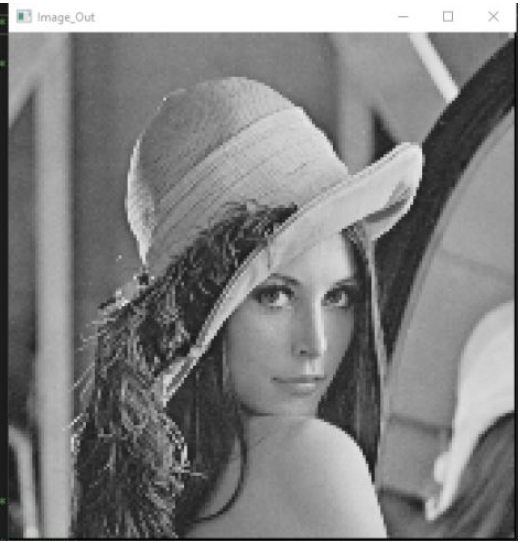
The following functions apply scaling operations to the input image, which change the size of the image. This is done by down-sampling the image - using one value to represent a block of pixels.



Down-Sampling (Center Pixel)

This 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.

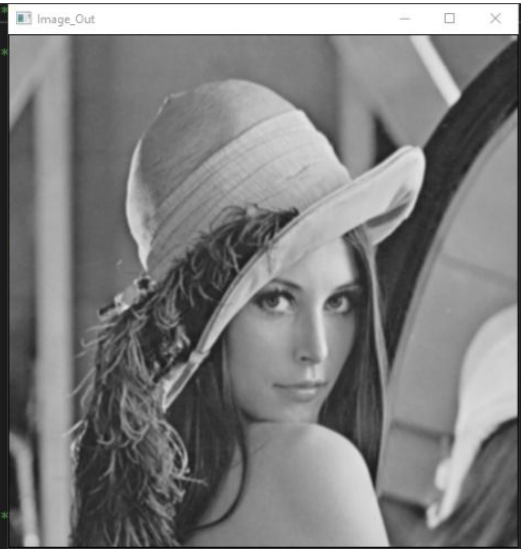
```
/*************************************************************
/* Image Processing begins
/*************************************************************


//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

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).

```
// 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;

       }
```