Class Lab 3

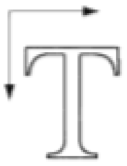
Use two images for each operation to do the following operations and write down their advantages and disadvantages and explain your results:

1. Image translation (lena, bridge):

Algorithm:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

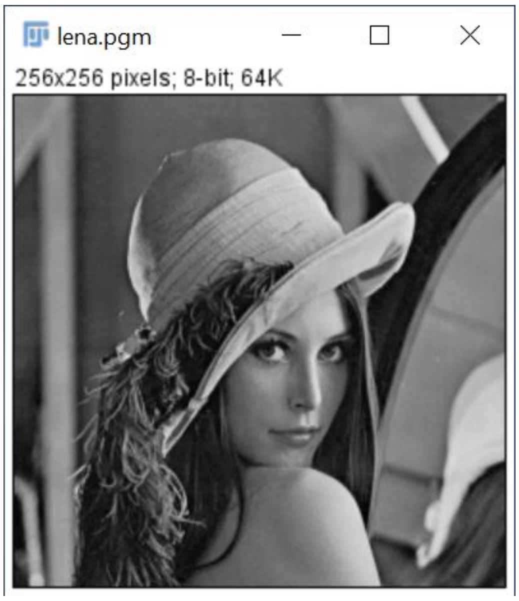
$$x = v + t_x$$
$$y = w + t_y$$



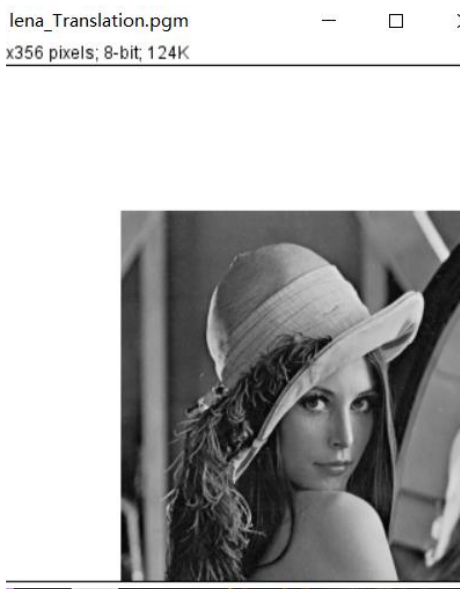
Results (including pictures):

Result of processing “Lena.pgm”:

Source Image:

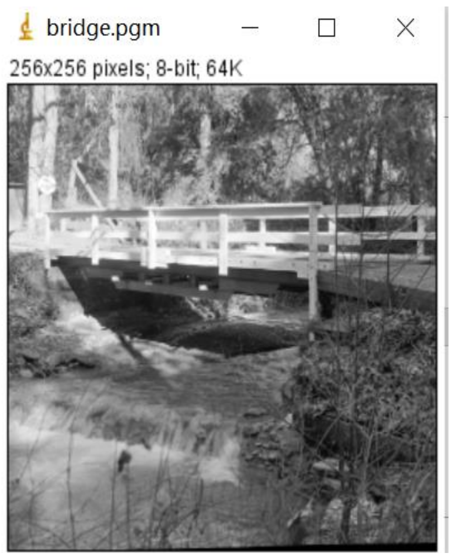


Result after translation:

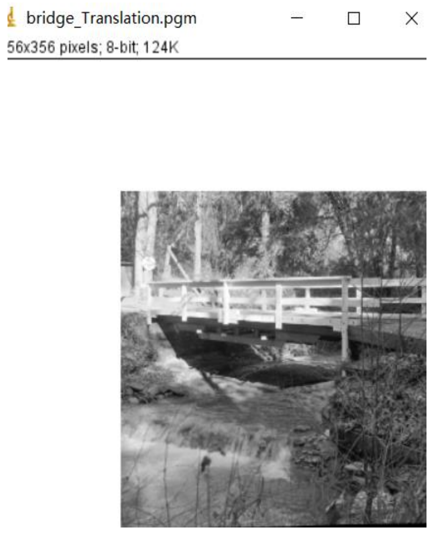


Result of processing “Bridge.pgm”:

Source Image:



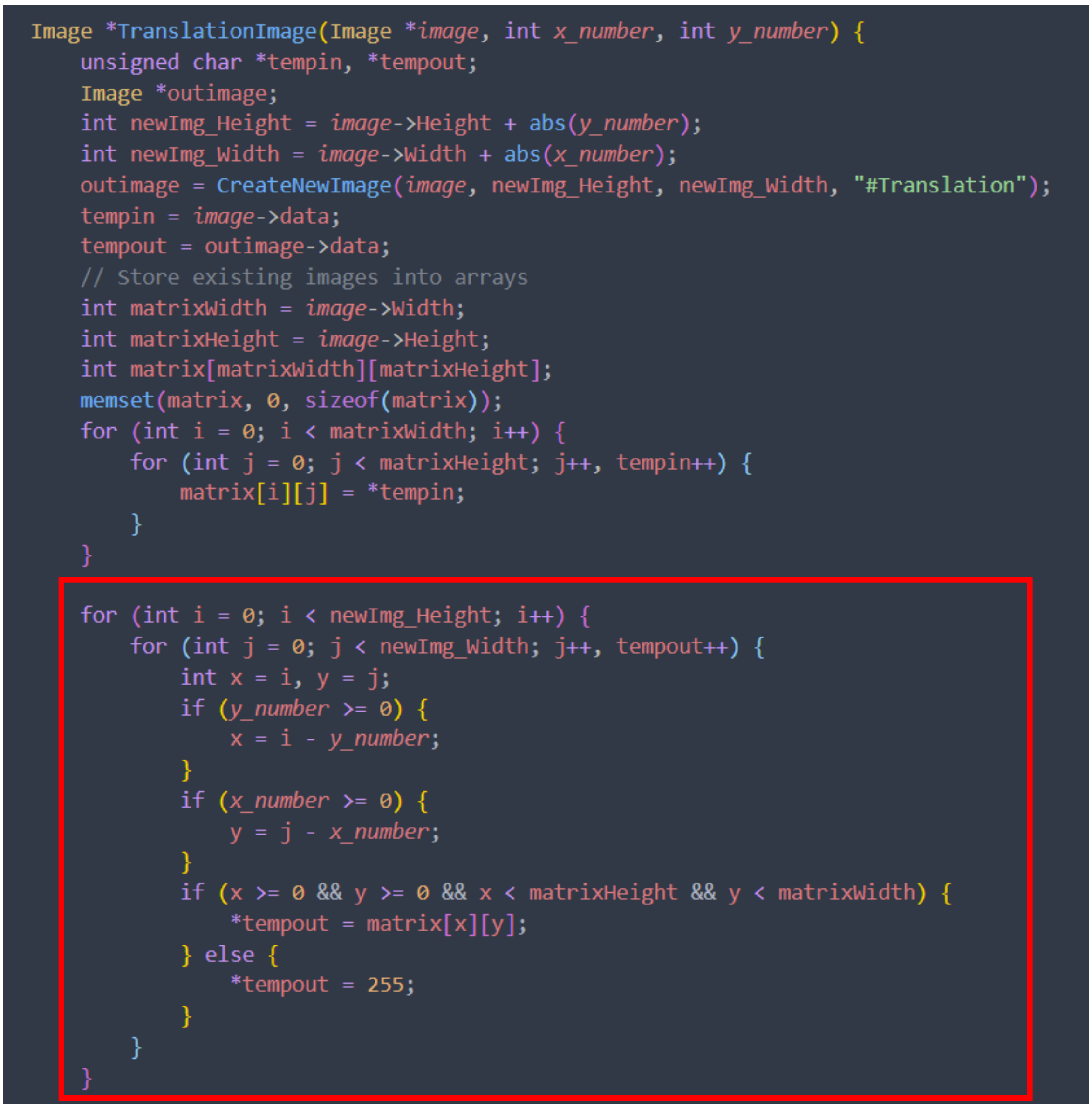
Result after translation



Discussion:

By using translation, the image has been shifted 100 pixels to the right and 100 pixels down.

Codes:



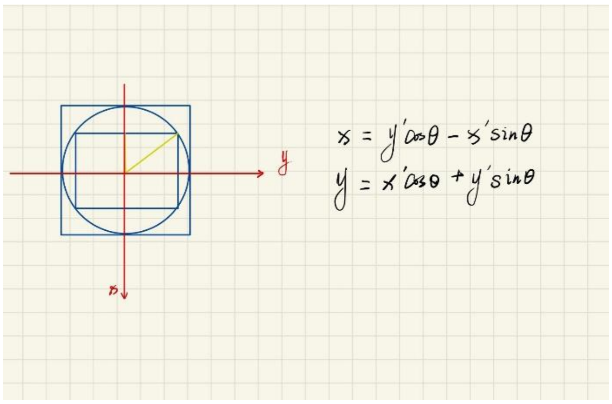
```
Image *TranslationImage(Image *image, int x_number, int y_number) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    int newImg_Height = image->Height + abs(y_number);
    int newImg_Width = image->Width + abs(x_number);
    outimage = CreateNewImage(image, newImg_Height, newImg_Width, "#Translation");
    tempin = image->data;
    tempout = outimage->data;
    // Store existing images into arrays
    int matrixWidth = image->Width;
    int matrixHeight = image->Height;
    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixHeight; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    for (int i = 0; i < newImg_Height; i++) {
        for (int j = 0; j < newImg_Width; j++, tempout++) {
            int x = i, y = j;
            if (y_number >= 0) {
                x = i - y_number;
            }
            if (x_number >= 0) {
                y = j - x_number;
            }
            if (x >= 0 && y >= 0 && x < matrixHeight && y < matrixWidth) {
                *tempout = matrix[x][y];
            } else {
                *tempout = 255;
            }
        }
    }
```

2. Image rotation (lena, bridge):

Algorithm:

$$\begin{matrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$\begin{cases} x' = xcos\theta - ysin\theta \\ y' = xsin\theta + ycos\theta \end{cases}$$

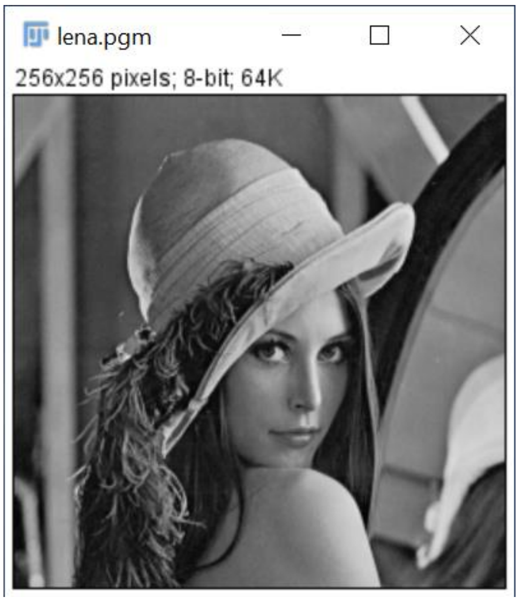


y
x
$x = y'cos\theta - x'sin\theta$
$y = x'cos\theta + y'sin\theta$

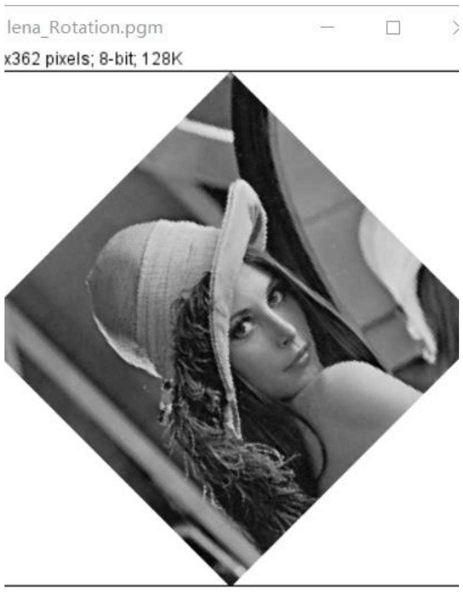
Results (including pictures):

Result of processing “Lena.pgm”:

Source Image:

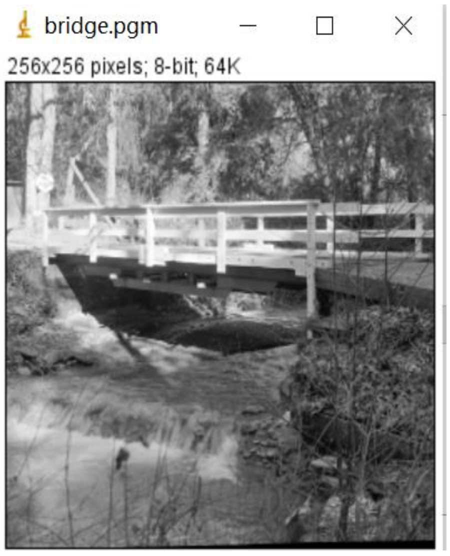


Result after rotation:

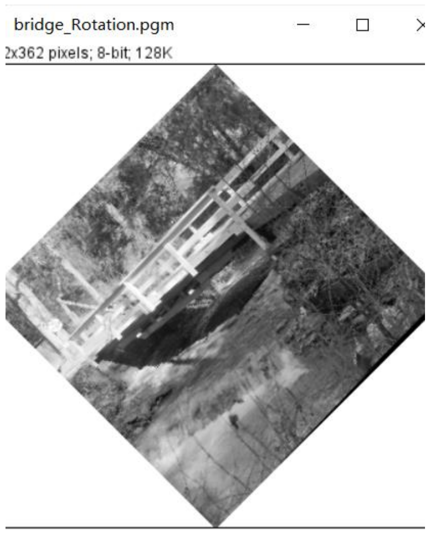


Result of processing “Bridge.pgm”:

Source Image:



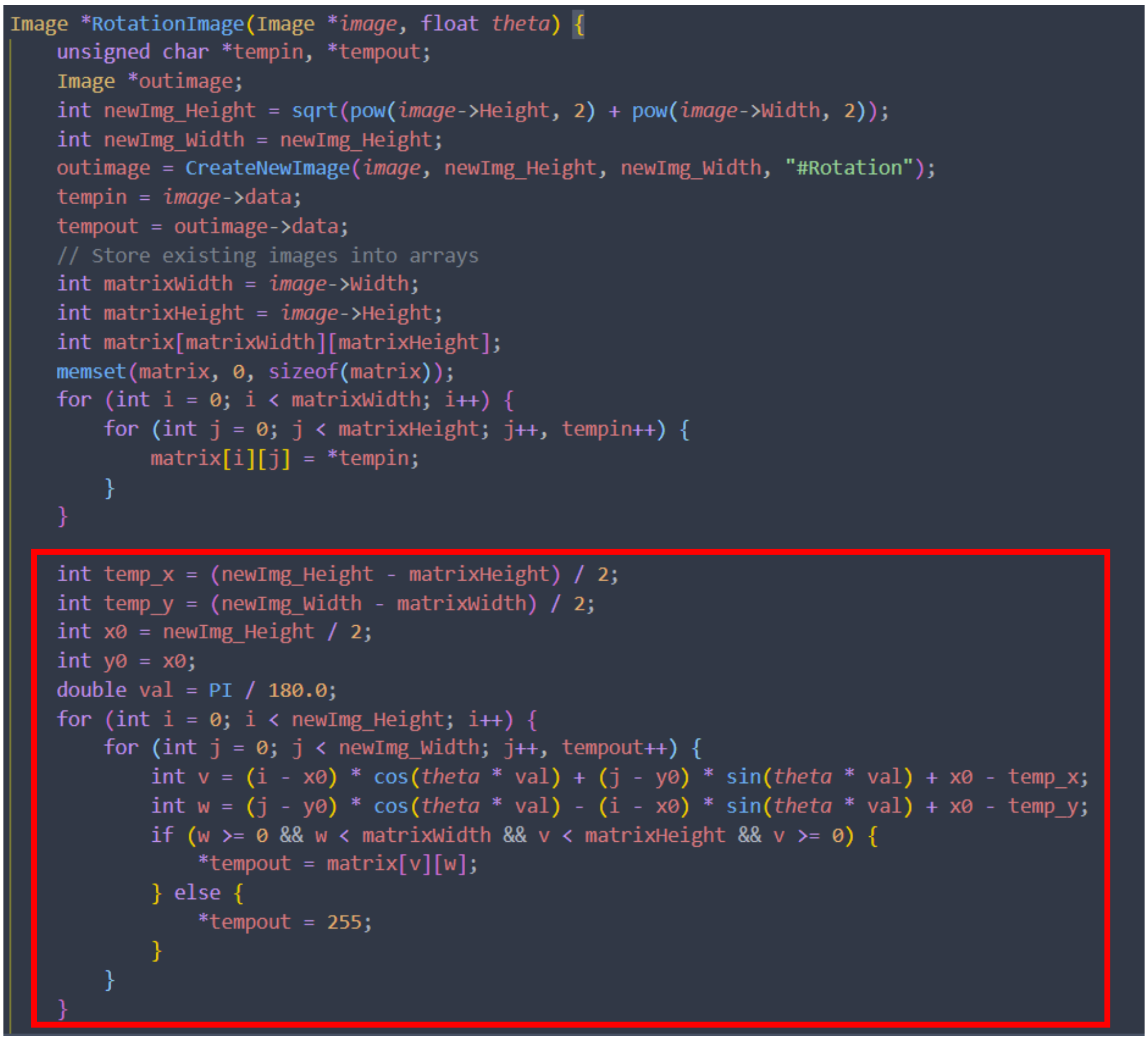
Result after rotation



Discussion:

By using rotation, the image has been rotated 45° .

Codes:

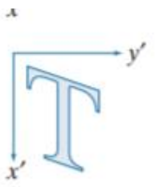


```
Image *RotationImage(Image *image, float theta) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    int newImg_Height = sqrt(pow(image->Height, 2) + pow(image->Width, 2));
    int newImg_Width = newImg_Height;
    outimage = CreateNewImage(image, newImg_Height, newImg_Width, "#Rotation");
    tempin = image->data;
    tempout = outimage->data;
    // Store existing images into arrays
    int matrixWidth = image->Width;
    int matrixHeight = image->Height;
    int matrix[matrixWidth][matrixHeight];
    memset(matrix, 0, sizeof(matrix));
    for (int i = 0; i < matrixWidth; i++) {
        for (int j = 0; j < matrixHeight; j++, tempin++) {
            matrix[i][j] = *tempin;
        }
    }

    int temp_x = (newImg_Height - matrixHeight) / 2;
    int temp_y = (newImg_Width - matrixWidth) / 2;
    int x0 = newImg_Height / 2;
    int y0 = x0;
    double val = PI / 180.0;
    for (int i = 0; i < newImg_Height; i++) {
        for (int j = 0; j < newImg_Width; j++, tempout++) {
            int v = (i - x0) * cos(theta * val) + (j - y0) * sin(theta * val) + x0 - temp_x;
            int w = (j - y0) * cos(theta * val) - (i - x0) * sin(theta * val) + x0 - temp_y;
            if (w >= 0 && w < matrixWidth && v < matrixHeight && v >= 0) {
                *tempout = matrix[v][w];
            } else {
                *tempout = 255;
            }
        }
    }
```

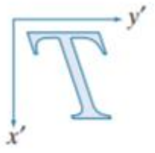
3. Image Shear (lena, bridge):

Algorithm:

Shear (vertical) $\begin{bmatrix} 1 & s_v & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $x' = x + s_v y$, $y' = y$



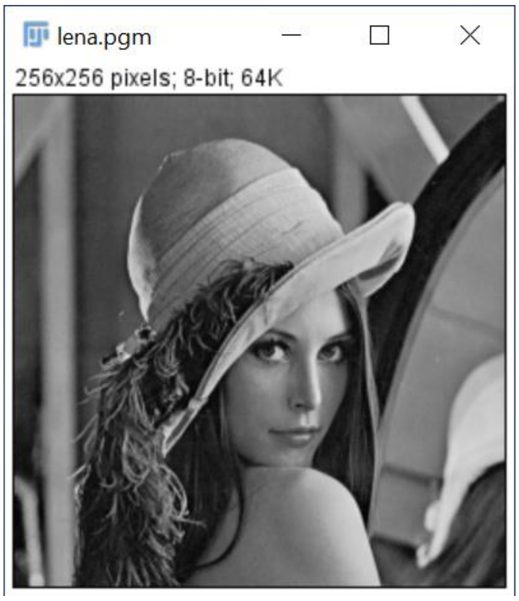
Shear (horizontal) $\begin{bmatrix} 1 & 0 & 0 \\ s_h & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ $x' = x$, $y' = s_h x + y$



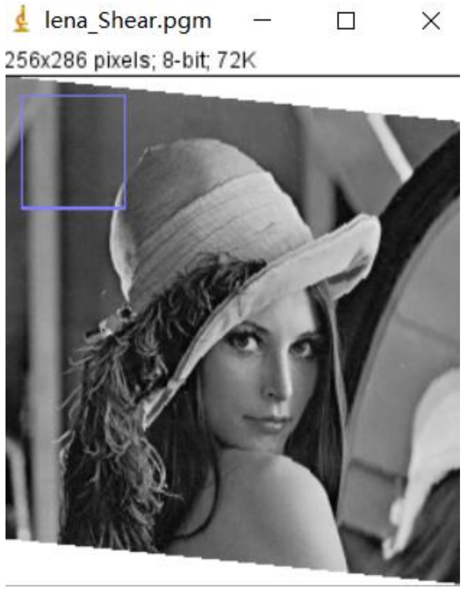
Results (including pictures):

Result of processing “Lena.pgm”:

Source Image:

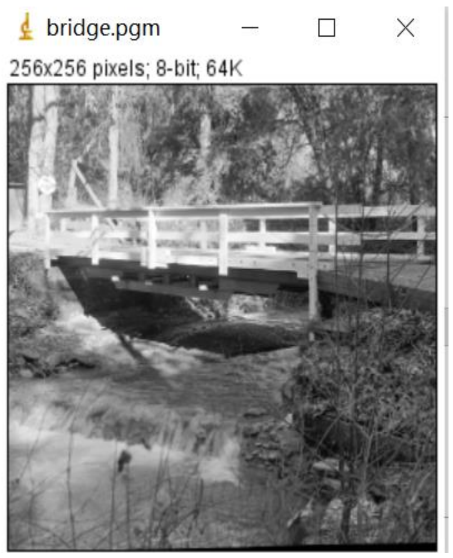


Result after translation:

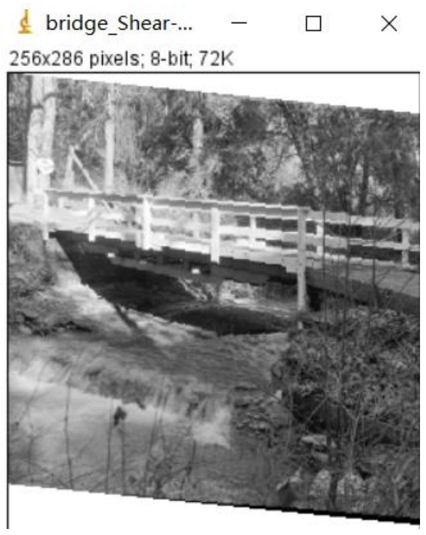


Result of processing “Bridge.pgm”:

Source Image:



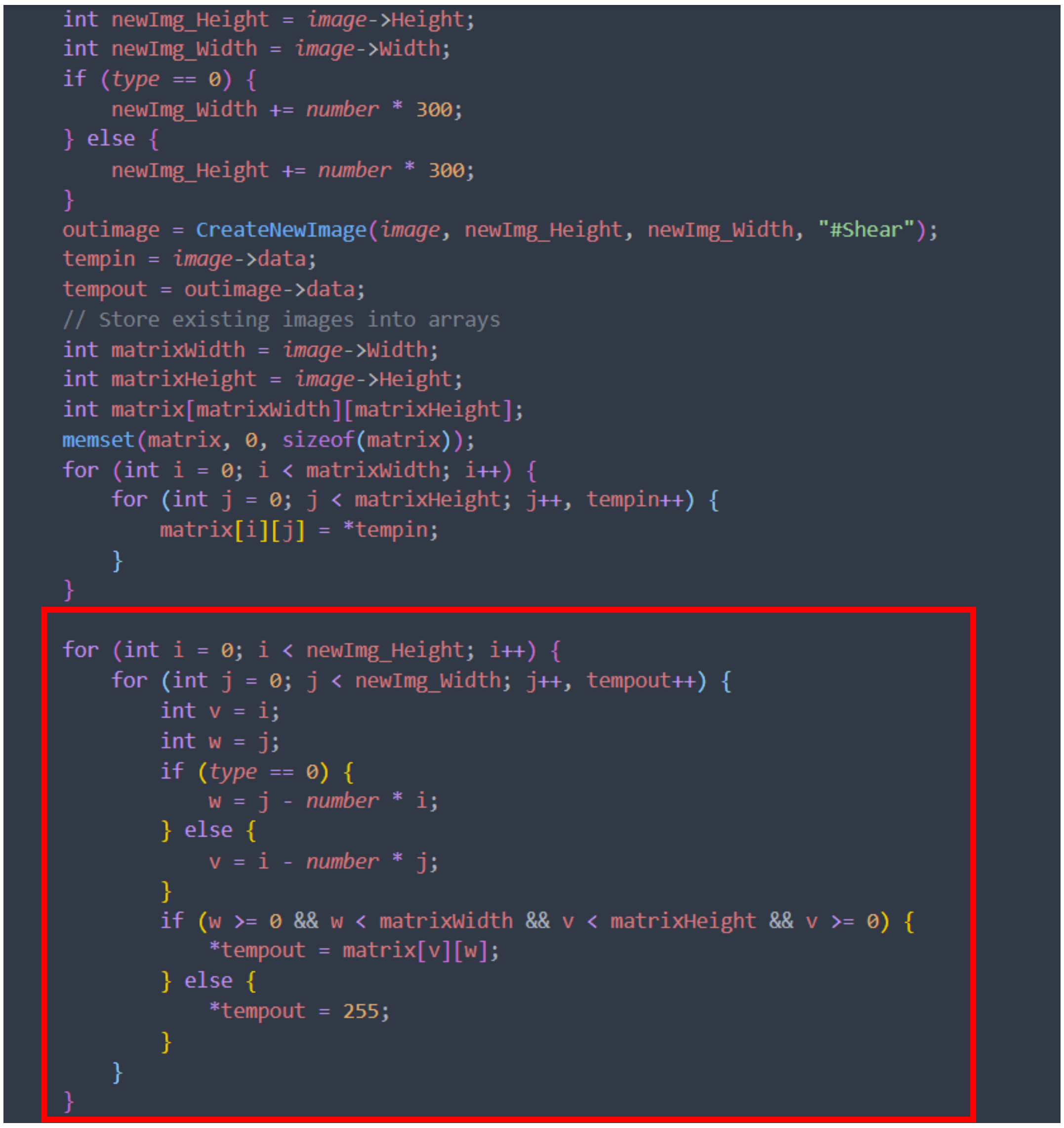
Result after translation



Discussion:

By using shear, the image has been sheared 0.1° .

Codes:

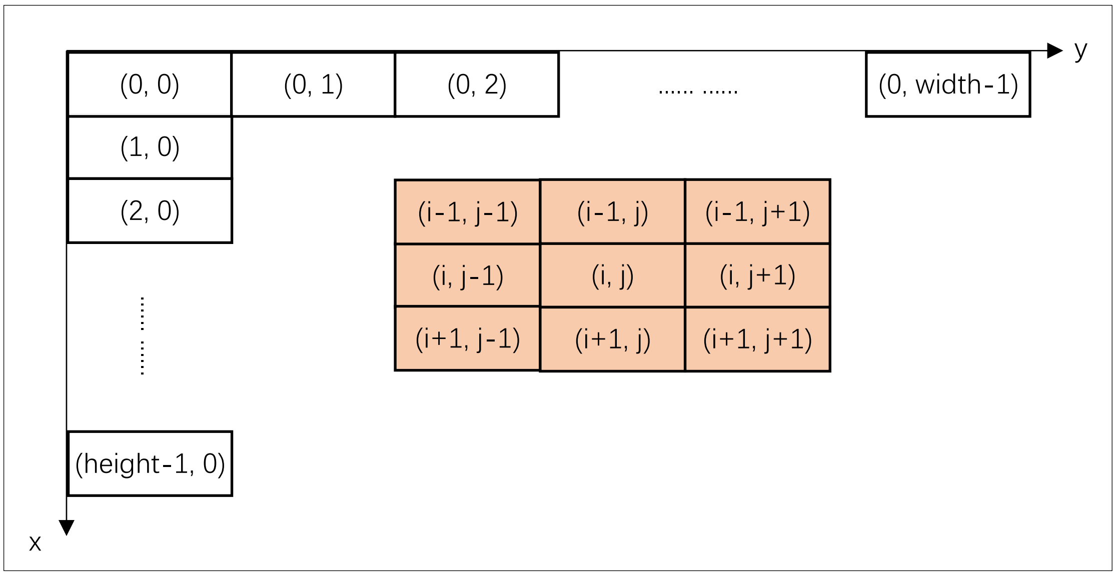


```
int newImg_Height = image->Height;
int newImg_Width = image->Width;
if (type == 0) {
    newImg_Width += number * 300;
} else {
    newImg_Height += number * 300;
}
outimage = CreateNewImage(image, newImg_Height, newImg_Width, "#Shear");
tempin = image->data;
tempout = outimage->data;
// Store existing images into arrays
int matrixWidth = image->Width;
int matrixHeight = image->Height;
int matrix[matrixWidth][matrixHeight];
memset(matrix, 0, sizeof(matrix));
for (int i = 0; i < matrixWidth; i++) {
    for (int j = 0; j < matrixHeight; j++, tempin++) {
        matrix[i][j] = *tempin;
    }
}

for (int i = 0; i < newImg_Height; i++) {
    for (int j = 0; j < newImg_Width; j++, tempout++) {
        int v = i;
        int w = j;
        if (type == 0) {
            w = j - number * i;
        } else {
            v = i - number * j;
        }
        if (w >= 0 && w < matrixWidth && v < matrixHeight && v >= 0) {
            *tempout = matrix[v][w];
        } else {
            *tempout = 255;
        }
    }
}
```

4. Filter:

- 3*3 Average Filter

Algorithm:

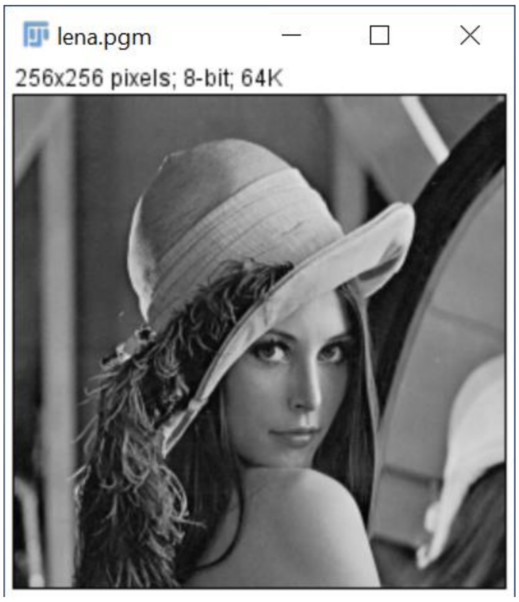


$$outImage(i,j) = \sum_{k=-1}^{1}\sum_{t=-1}^{1} inImage(i+k, j+t)\,/9 \qquad (1.1)$$
$$indexInData \;=\; i * width + j \qquad (1.2)$$
$$outData[i * width + j] = \sum_{k=-1}^{1}\sum_{t=-1}^{1} inData[(i+k) * width + (j+t)]\,/9 \qquad (1.3)$$

Results (including pictures):

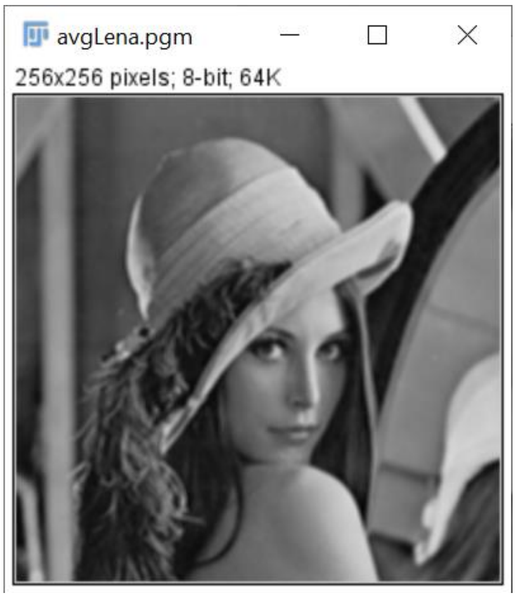
Result of processing "Lena.pgm":

Source Image:


lena.pgm

256x256 pixels; 8-bit; 64K

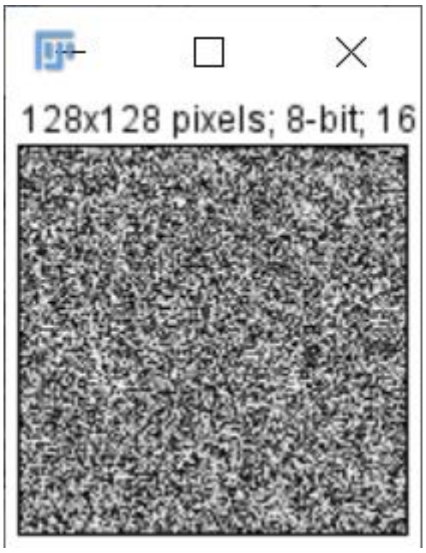
Result after average filter:

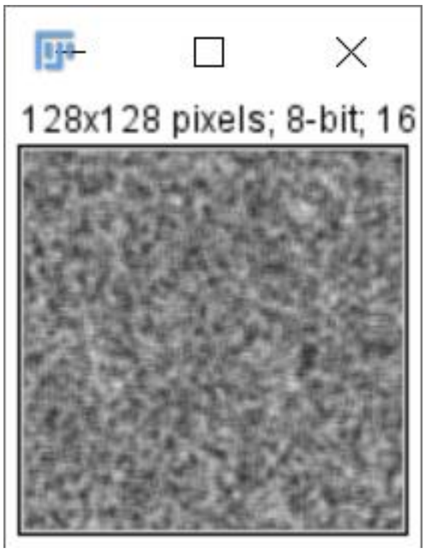

avgLena.pgm

256x256 pixels; 8-bit; 64K

Result of processing "Noise.pgm":

Source Image:





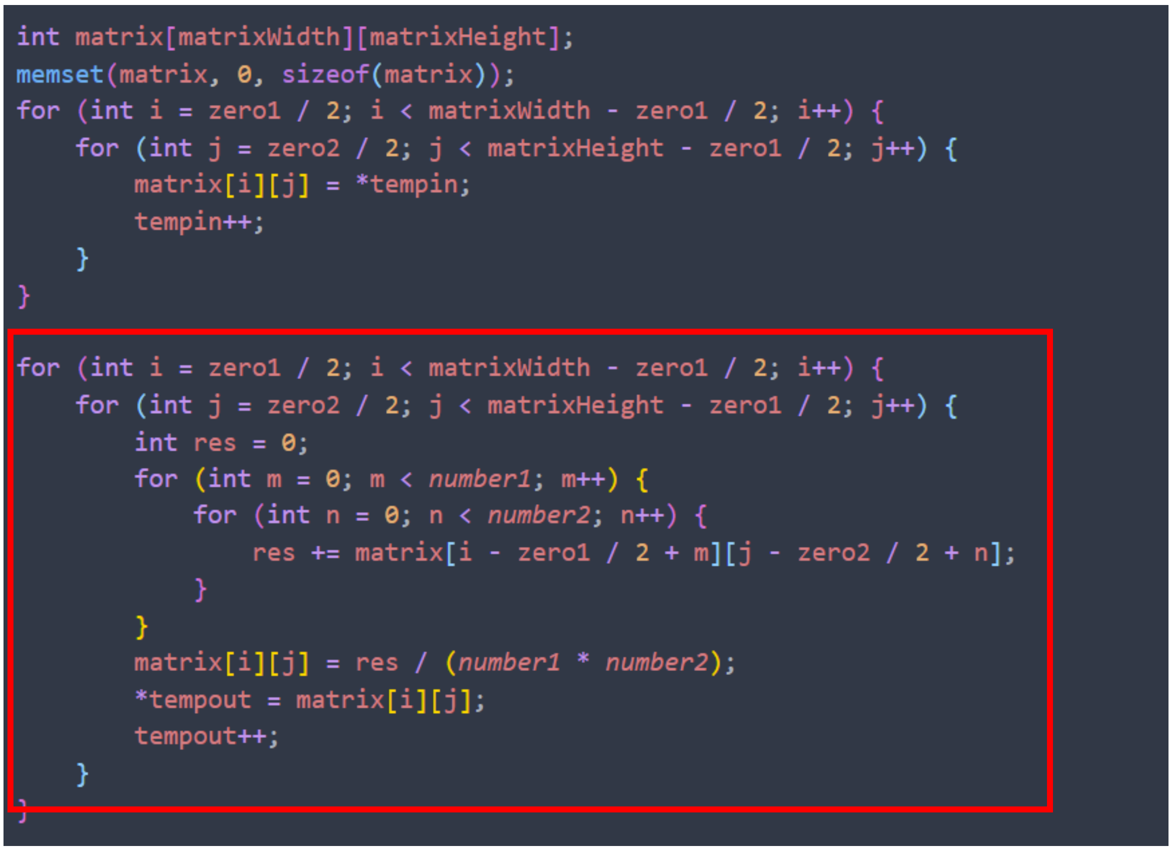
Result after average filter:

Discussion:

From equation (1.1), we can conclude that differences between neighbor pixels become smaller by averaging, which leads to the smoother of image.

The result images do turn out to be smoother. For example, the edges between objects are less obvious.

Codes:



```
int matrix[matrixWidth][matrixHeight];
memset(matrix, 0, sizeof(matrix));
for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
    for (int j = zero2 / 2; j < matrixHeight - zero1 / 2; j++) {
        matrix[i][j] = *tempin;
        tempin++;
    }
}

for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
    for (int j = zero2 / 2; j < matrixHeight - zero1 / 2; j++) {
        int res = 0;
        for (int m = 0; m < number1; m++) {
            for (int n = 0; n < number2; n++) {
                res += matrix[i - zero1 / 2 + m][j - zero2 / 2 + n];
            }
        }
        matrix[i][j] = res / (number1 * number2);
        *tempout = matrix[i][j];
        tempout++;
    }
}
```

- 3*3 Median Filter

Algorithm:

Input: a unsigned char array that stores the source image data

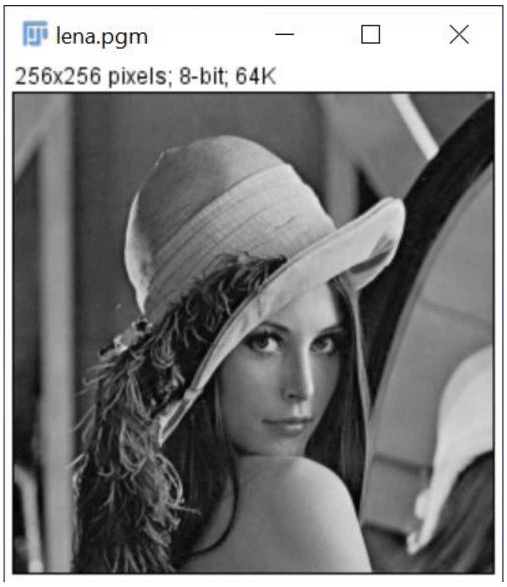
Output: a unsigned char array that stores the output image data

```
For inImage(i, j):
    Store its' 3 x 3 neighbor including itself in a unsigned char array **local**[9];
    Find out the median of **local**;
    Assign the median to outImage(i,j) => $outData[i * width + j] = findMedian(\boldsymbol{local}, 9)$
END
```

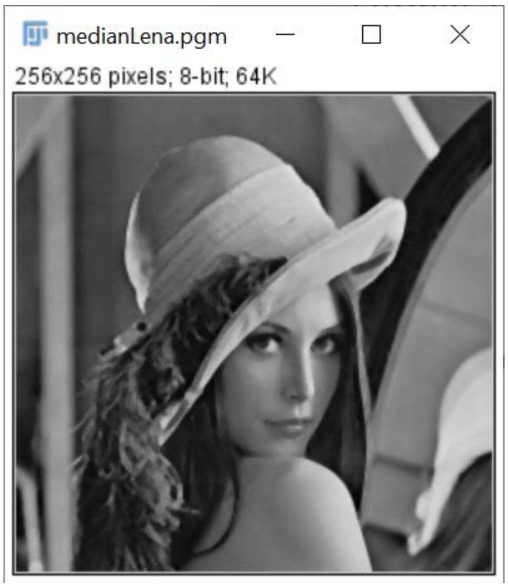
Results (including pictures):

Result of processing "Lena.pgm":

Source Image:

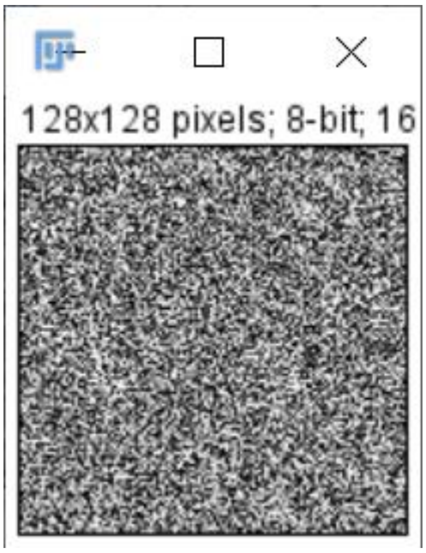

lena.pgm
256x256 pixels; 8-bit; 64K

Result after median filter:

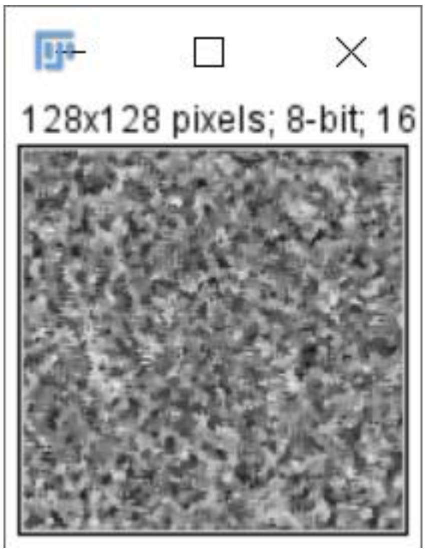

medianLena.pgm
256x256 pixels; 8-bit; 64K

Result of processing "Noise.pgm":

Source Image:


128x128 pixels; 8-bit; 16

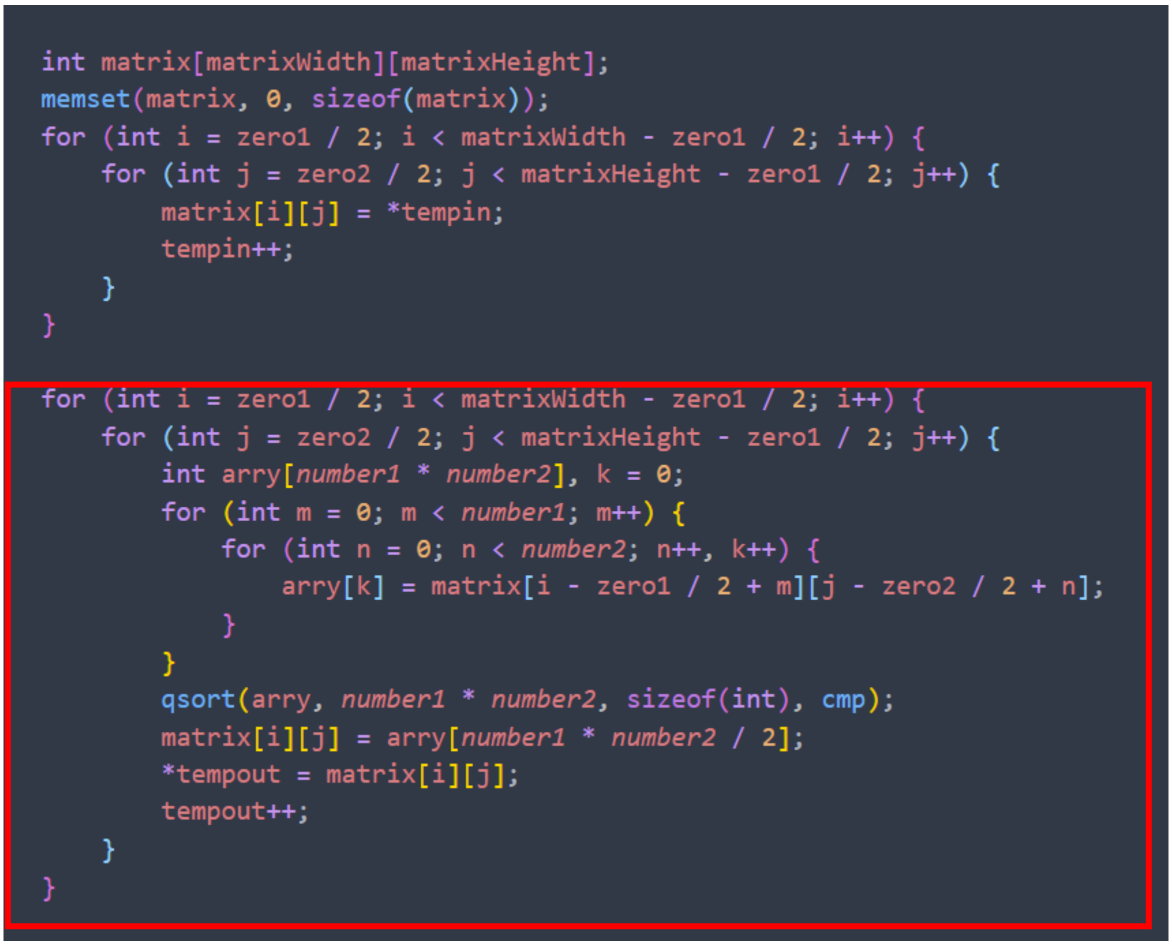
Result after median filter:


128x128 pixels; 8-bit; 16

Discussion:

The median filter also smooths the input image. But the result of median filter is sharper than that of average filter. Median filter simply substitutes the target pixel with the median of its 3 x 3 neighbor, that is, the operation does not operate all pixels in the neighbor. So the differences between pixels in the output image by median filter is larger than that by average filter.

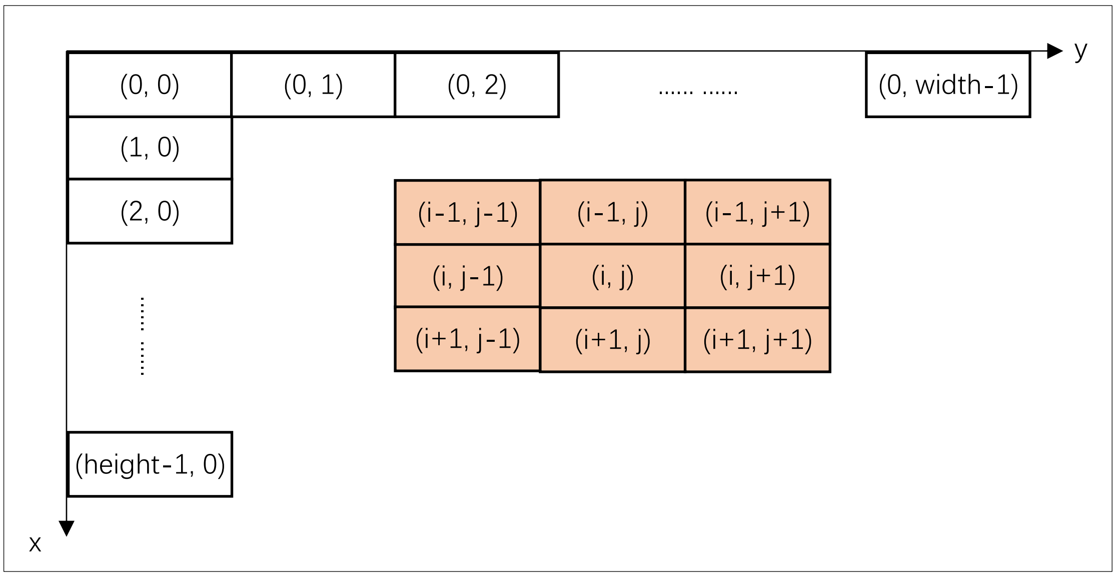
Codes:



```
int matrix[matrixWidth][matrixHeight];
memset(matrix, 0, sizeof(matrix));
for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
    for (int j = zero2 / 2; j < matrixHeight - zero1 / 2; j++) {
        matrix[i][j] = *tempin;
        tempin++;
    }
}

for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
    for (int j = zero2 / 2; j < matrixHeight - zero1 / 2; j++) {
        int arry[number1 * number2], k = 0;
        for (int m = 0; m < number1; m++) {
            for (int n = 0; n < number2; n++, k++) {
                arry[k] = matrix[i - zero1 / 2 + m][j - zero2 / 2 + n];
            }
        }
        qsort(arry, number1 * number2, sizeof(int), cmp);
        matrix[i][j] = arry[number1 * number2 / 2];
        *tempout = matrix[i][j];
        tempout++;
    }
}
```

- 5*5 Average Filter

Algorithm:



$$outImage(i,j) = \sum_{k=-1}^{1}\sum_{t=-1}^{1} inImage(i+k, j+t)\,/9 \quad (1.1)$$

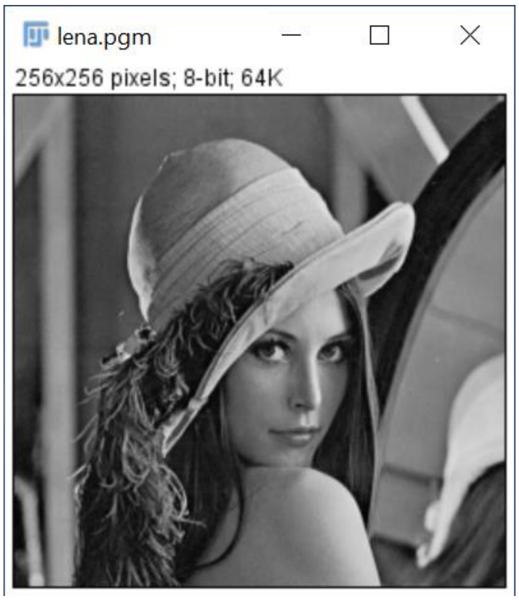
$$indexInData \;=\; i * width + j \quad (1.2)$$

$$outData[i * width + j] = \sum_{k=-1}^{1}\sum_{t=-1}^{1} inData[(i+k) * width + (j+t)]\,/9 \quad (1.3)$$

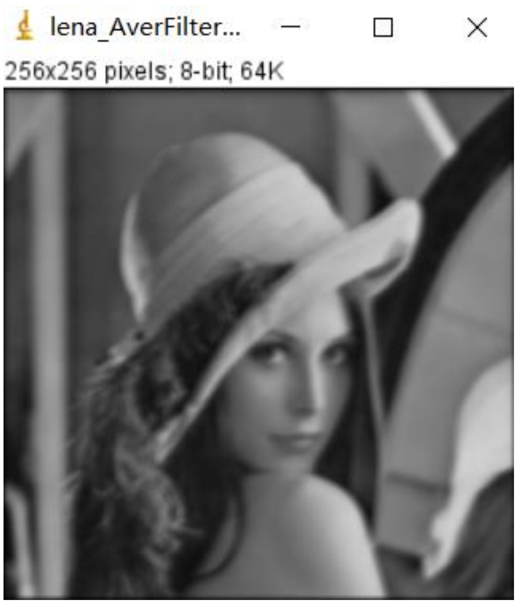
Results (including pictures):

Result of processing “Lena.pgm”:

Source Image:

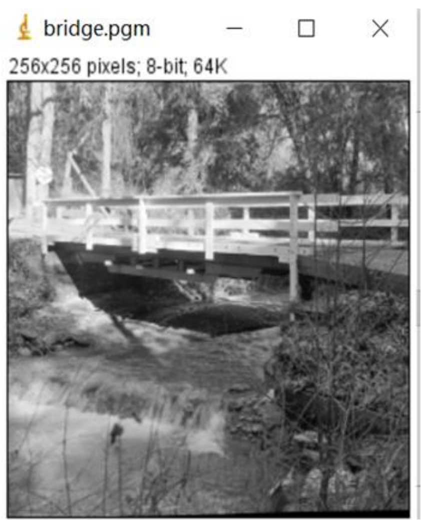

lena.pgm
256x256 pixels; 8-bit; 64K

Result after average filter:

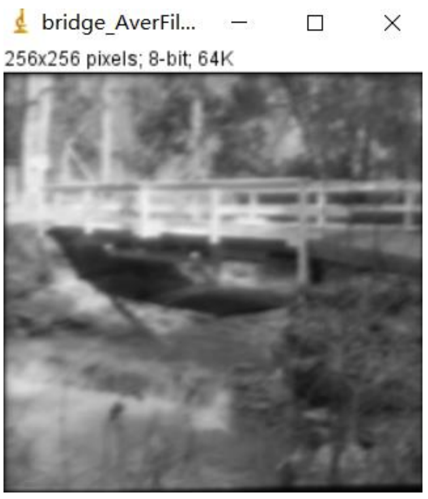

lena_AverFilter...
256x256 pixels; 8-bit; 64K

Result of processing “Noise.pgm”:

Source Image:



Result after average filter:

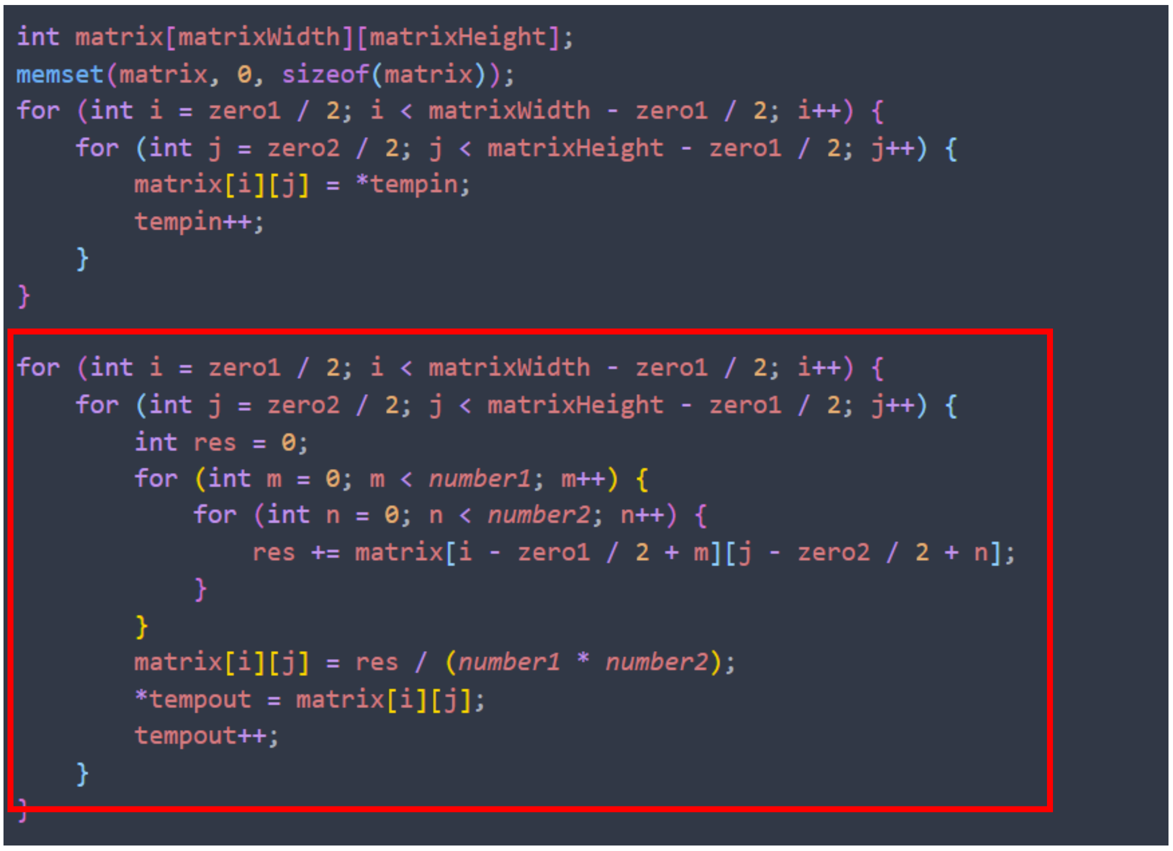


Discussion:

From equation (1.1), we can conclude that differences between neighbor pixels become smaller by averaging, which leads to the smoother of image.

The result images do turn out to be smoother. For example, the edges between objects are less obvious.

Codes:



```
int matrix[matrixWidth][matrixHeight];
memset(matrix, 0, sizeof(matrix));
for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
    for (int j = zero2 / 2; j < matrixHeight - zero1 / 2; j++) {
        matrix[i][j] = *tempin;
        tempin++;
    }
}

for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
    for (int j = zero2 / 2; j < matrixHeight - zero1 / 2; j++) {
        int res = 0;
        for (int m = 0; m < number1; m++) {
            for (int n = 0; n < number2; n++) {
                res += matrix[i - zero1 / 2 + m][j - zero2 / 2 + n];
            }
        }
        matrix[i][j] = res / (number1 * number2);
        *tempout = matrix[i][j];
        tempout++;
    }
}
```

- 5*5 Median Filter

Algorithm:

Input: a unsigned char array that stores the source image data

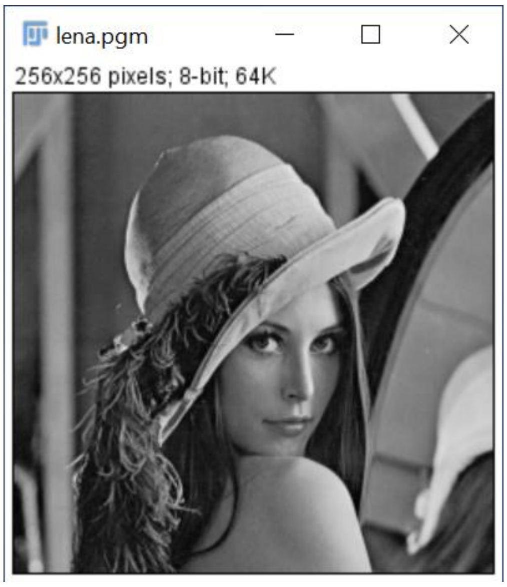
Output: a unsigned char array that stores the output image data

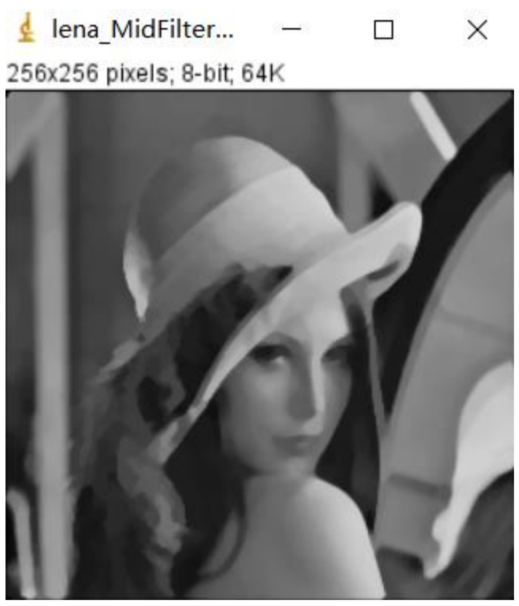
```
For inImage(i, j):
    Store its' 3 x 3 neighbor including itself in a unsigned char array **local**[9];
    Find out the median of **local**;
    Assign the median to outImage(i,j) => $outData[i * width + j] = findMedian(\boldsymbol{local}, 9)$
END
```

Results (including pictures):

Result of processing "Lena.pgm":

Source Image:

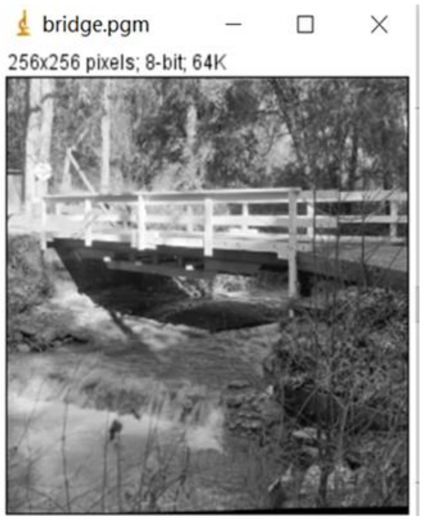


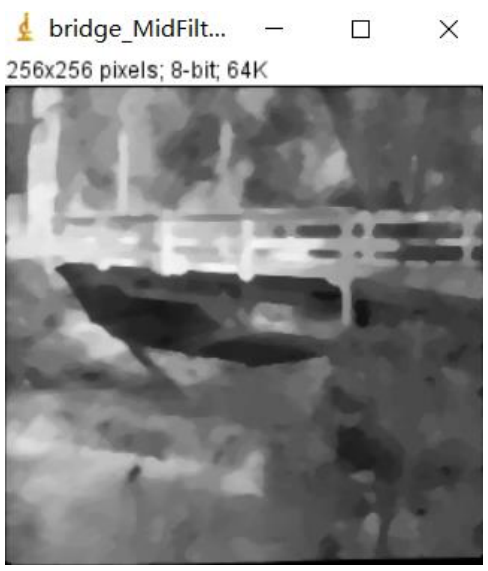


Result after median filter:

Result of processing "Noise.pgm":

Source Image:



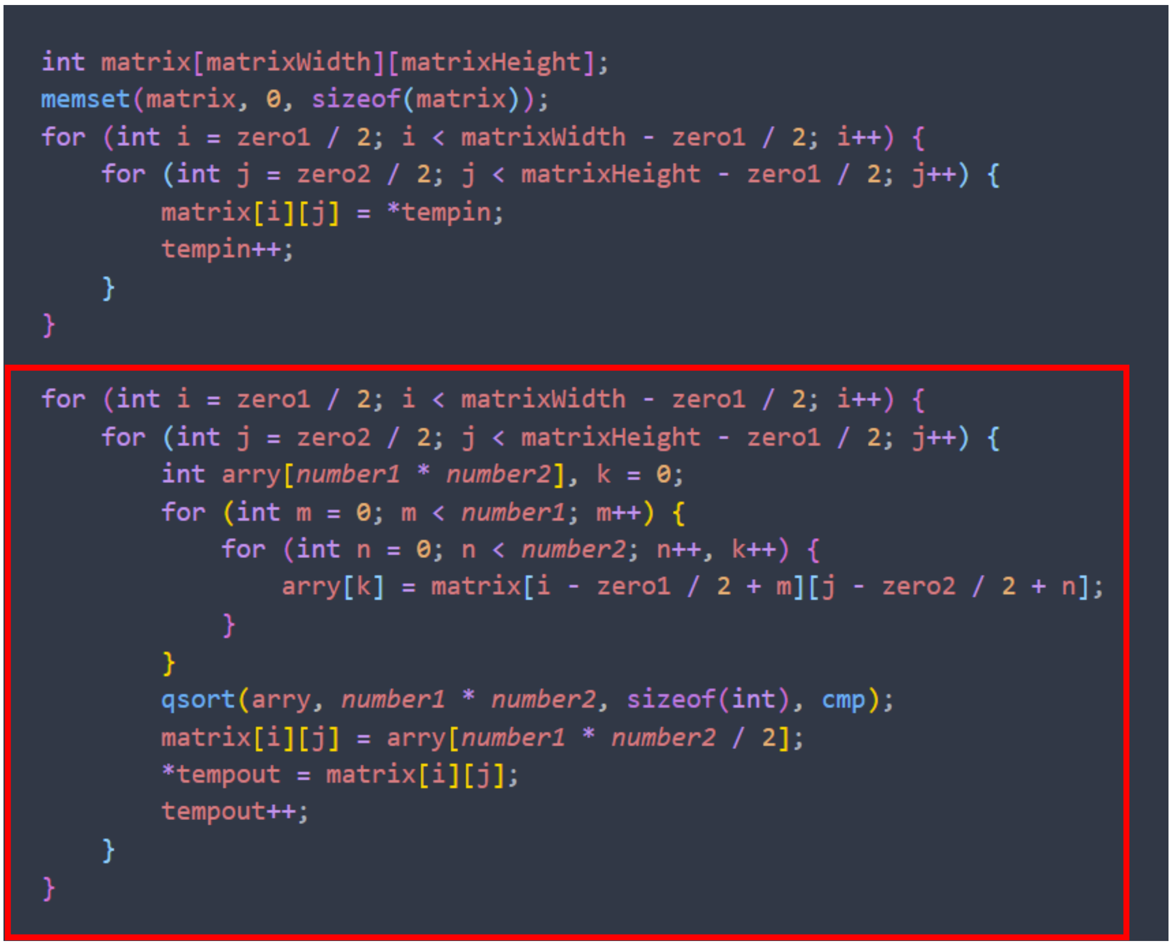


Result after median filter:

Discussion:

We can conclude that differences between neighbor pixels become more smaller by averaging, which leads to the smoother of image.

Codes:



```
int matrix[matrixWidth][matrixHeight];
memset(matrix, 0, sizeof(matrix));
for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
    for (int j = zero2 / 2; j < matrixHeight - zero1 / 2; j++) {
        matrix[i][j] = *tempin;
        tempin++;
    }
}

for (int i = zero1 / 2; i < matrixWidth - zero1 / 2; i++) {
    for (int j = zero2 / 2; j < matrixHeight - zero1 / 2; j++) {
        int arry[number1 * number2], k = 0;
        for (int m = 0; m < number1; m++) {
            for (int n = 0; n < number2; n++, k++) {
                arry[k] = matrix[i - zero1 / 2 + m][j - zero2 / 2 + n];
            }
        }
        qsort(arry, number1 * number2, sizeof(int), cmp);
        matrix[i][j] = arry[number1 * number2 / 2];
        *tempout = matrix[i][j];
        tempout++;
    }
}
```