# EE431 Homework2

Date: 16.11.2022

In this assignment, Gaussian filtering and Median filtering application was studied on an image. For this A gaussian filtering function in unsigned char **gaussf (**unsigned char img, int NC, int NR, int count) format and a median filtering function in unsigned char **medianf (**unsigned char img, int NC, int NR, int count) format is written.

**PART 1:** Gauss Function

Below is the gaussian function. The algorithm and parameters used with the function are explained in the position lines. Since we are aware that sides cannot be filtered, a padding variable is required to specify the side gap.

```
// ---------------------- GAUSSIAN FILTERING FUNCTION PART ----------------------
unsigned char **gaussf (unsigned char **img , int NC , int NR , int count)
{
    unsigned char **temp_1 , **temp_2; // to not destroy the original img, we need to create new variables to store and process
    int i , j , k , padding;
    temp_1 = alloc_img(NC , NR); // to allocate space for corrected pixel values (same size with img)
    temp_2 = alloc_img(NC , NR); // to allocate space for corrected pixel values (same size with img)
    padding = count / 2; // define a padding which prevents segmentation faults

    for(i = padding; i < NR - padding; i++)
    {
        for(j = padding; j < NC - padding; j++)
        {
            for(i = 0 ; i < NR ; i++)
            {
                for(j = 0 ; j < NC ; j++)
                {
                    temp_1[i][j] = img[i][j]; // to process the gaussian filter, and not destroy the original img, temp_1 used as a temporary variable
                }
            }

            for(k = 0 ; k < count ; k++)
            {
                for(i = 0 ; i < NR ; i++)
                {
                    for(j = 1 ; j < NC - 1 ; j++)
                    {
                        temp_2[i][j] = (temp_1[i][j-1] + 2 * temp_1[i][j] + temp_1[i][j+1]) / 4; // horizontal mask was done and stored in temp_2 variable
                    }
                }
                for(i = 1 ; i < NR - 1 ; i++)
                {
                    for(j = 0 ; j < NC ; j++)
                    {
                        temp_1[i][j] = (temp_2[i-1][j] + 2 * temp_2[i][j] + temp_2[i+1][j]) / 4; // vertical mask was done and stored in temp_1 varible
                    }
                }
            }
        }
    }
    free_img(temp_2); // since temp_2 is not gonna used, it was freed
    return temp_1; // in the main part, temp_1 variable will be assigned for displaying the processed image, so it was returned
}
```

**PART 2:** Median Function

Below is the median function. The algorithm and parameters used with the function are explained in the position lines. We require a calculating memory for sorting, loops for iteratively processing pixels, and median filtering. A padding variable is required to specify the side gap, like we used in gauss filter function seen above. We put the values of the neighboring pixels and the filter-applied pixels into the array. The median is then determined by setting the filter size's maximum values to zero.

```
// ---------------------- MEDIAN FILTERING FUNCTION PART ----------------------
unsigned char **medianf(unsigned char **img, int NC, int NR, int count)
{
    unsigned char **temp_1; // to not destroy the original img, we need to create new variables to store and process
    int i, j, k, l, maxval, maxindex, padding;
    int arr[count * count]; // create an array for filter to hold data
    temp_1= alloc_img(NC,NR); // to allocate space for corrected pixel values (same size with img)
    padding = count / 2; // define a padding which prevents segmentation faults

    // for loops and image coordinates includes padding for dynamic size operations
    for(i = padding; i < NR - padding; i++)
    {
        for(j = padding; j < NC - padding; j++)
        {

            for(k = 0; k < count; k++)
            {
                for(l = 0; l < count; l++)
                {
                    arr[count * k + l] = img[i - padding + k][j - padding + l];     // fill the array from pixel and its neighbors
                }
            }
            for(k = 0; k < (count * count) / 2 + 1; k++)
            {
                // to the middle term, zero the max values
                maxval = -1;
                maxindex = 0;
                for(l = 0; l < count * count; l++)
                {
                    if(arr[l] > maxval)
                    {
                        maxval = arr[l];
                        maxindex = l;       // find max value
                    }
                }
                arr[maxindex] = 0;
            }
            temp_1[i][j] = maxval;      // resulting pixel value
        }
    }
    return(temp_1);
}
```

**Part3:** Main Function

Below program operates according to the file name and flag value parameters entered by the user in the main function. Flag value corresponds to the count parameter according to the code written below. When count=1, the gaussian function is called in the main function and Gaussian filter is applied on the picture. When count=2, the median function is called and the Median filter is applied on the image.

The printing process of the images was done in the main part by applying filters on the image as many as the number of the quadrant in the form of four quadrants, namely Q1, Q2, Q3, and Q4. Among the parameters seen in the code below, top_left represents Q1, top_right represents Q2, bottom_left represents Q3, and bottom_right represents Q4. Filter operations were applied to Q1 once, to Q2 2 times, to Q3 3 times, and to Q4 4 times. In order to put the 4 quadrants together, an image parameter named temp_1 was created to be 4 times (2*NR*2*NC) of their dimensions NR*NC.

```
int main(int argc , char **argv)
{
unsigned char **img , **temp_1 , **top_left , **top_right , **bottom_left , **bottom_right;
char *pgm_file;
int i , j , k , NC , NR , count;

if(argc!=3)
{
  printf("\nUsage: 627635_hw2 [Image file (*.pgm)] [count]\n");
  printf("\nE.g.   627635_hw2 panda.pgm 1\n");
  exit(-1);
}

pgm_file = argv[1]; // 2nd input for which image should be corrected
count = atoi(argv[2]); // 3rd count variable got from user as string, we convert it into integer value to use it.
                       //It will be used as a flag to choose if we want to perform Gauss operation or Median operation

img = pgm_file_to_img(pgm_file , &NC , &NR); // to save uncorrected pixel values
show_pgm_file(pgm_file); // shows uncorrected image

// img allocation part
temp_1 = alloc_img(2 * NC , 2 * NR);    // to allocate space for corrected pixel values (same size with img)
top_left = alloc_img(NC , NR);      // to allocate space for corrected pixel values (same size with img)
top_right = alloc_img(NC , NR);     // to allocate space for corrected pixel values (same size with img)
bottom_left = alloc_img(NC , NR);   // to allocate space for corrected pixel values (same size with img)
bottom_right = alloc_img(NC , NR);  // to allocate space for corrected pixel values (same size with img)

// ---------------------- GAUSSIAN FILTERING MAIN (CALLING THE FUNCTION) PART ----------------------
if(count == 1)
{
    top_left = gaussf(img , NC , NR , 1);          // Filtering part, for top-left, 1 gaussian filter was processed
    top_right = gaussf(img , NC , NR , 2);         // Filtering part, for top-right, 2 gaussian filter was processed
    bottom_left = gaussf(img , NC , NR , 3);       // Filtering part, for bottom-left, 3 gaussian filter was processed
    bottom_right = gaussf(img , NC , NR , 4);      // Filtering part, for bottom-right, 4 gaussian filter was processed
}

// ---------------------- MEDIAN FILTERING MAIN (CALLING THE FUNCTION) PART ----------------------
else if(count == 2)
{
    top_left = medianf(img , NC , NR , 1);      // Filtering part, for top-left, 1 median filter was processed
    top_right = medianf(img , NC , NR , 2);     // Filtering part, for top-right, 2 median filter was processed
    bottom_left = medianf(img , NC , NR , 3);   // Filtering part, for bottom-left, 3 median filter was processed
    bottom_right = medianf(img , NC , NR , 4);  // Filtering part, for bottom-right, 4 median filter was processed
}
```

In an image with 2*NR*2*NC size, indexes smaller than NR and NC corresponds to quadrant Q1, pixels smaller than NR and larger than NC to Q2, values larger than NR and less than NC to Q3, and values larger than NR and NC to Q4. All these quadrants are collected in the temp_1 array of 4 times the size NR*NC.

```
// reconstruct the image by adding up the sliced and filtered pieces according to conditions mentioned below
for(i = 0 ; i < 2 * NR ; i++)
{
    for(j = 0 ; j < 2 *NC ; j++)
    {
        if(i < NR && j < NC)
        {
            temp_1[i][j] = top_left[i][j];          // Q1 allocation from top-left array to the temp_1 array
        }
        else if(i < NR && j > NC)
        {
            temp_1[i][j] = top_right[i][j - NC];        // Q2 allocation from top_right array to the temp_1 array
        }
        else if(i > NR && j < NC)
        {
            temp_1[i][j] = bottom_left[i - NR][j];      // Q3 allocation from bottom_left array to the temp_1 array
        }
        else if(i > NR && j > NC)
        {
            temp_1[i][j] = bottom_right[i - NR][j - NC];    // Q4 allocation from bottom_right array to the temp_1 array
        }
    }
}
// temp_1 used to reconstruct the image because, we wanted a non-destructive approach, with this, we saved the original img
```

It is seen that according to the value entered below (1 or 2) gaussian filter or median filter is applied.

```
if(count == 1)
{
    img_to_pgm_file(temp_1 , "gauss.pgm" , NC * 2 , NR * 2);    // Converting image(matrix) to Grayscale(actual image)
    show_pgm_file("gauss.pgm");                 // Displays the filtered image
}

if(count == 2)
{
    img_to_pgm_file(temp_1 , "median.pgm" , NC * 2 , NR * 2);   // Converting image(matrix) to Grayscale(actual image)
    show_pgm_file("median.pgm");                // Displays the filtered image
}

// img free part
free_img(top_left);
free_img(top_right);
free_img(bottom_left);
free_img(bottom_right);
free_img(img);
free_img(temp_1);

return(1);
}
```

To run our code, we used the following commands to compile and execute;

**gcc hw2.c -o hw2 img_pro.c -lm**

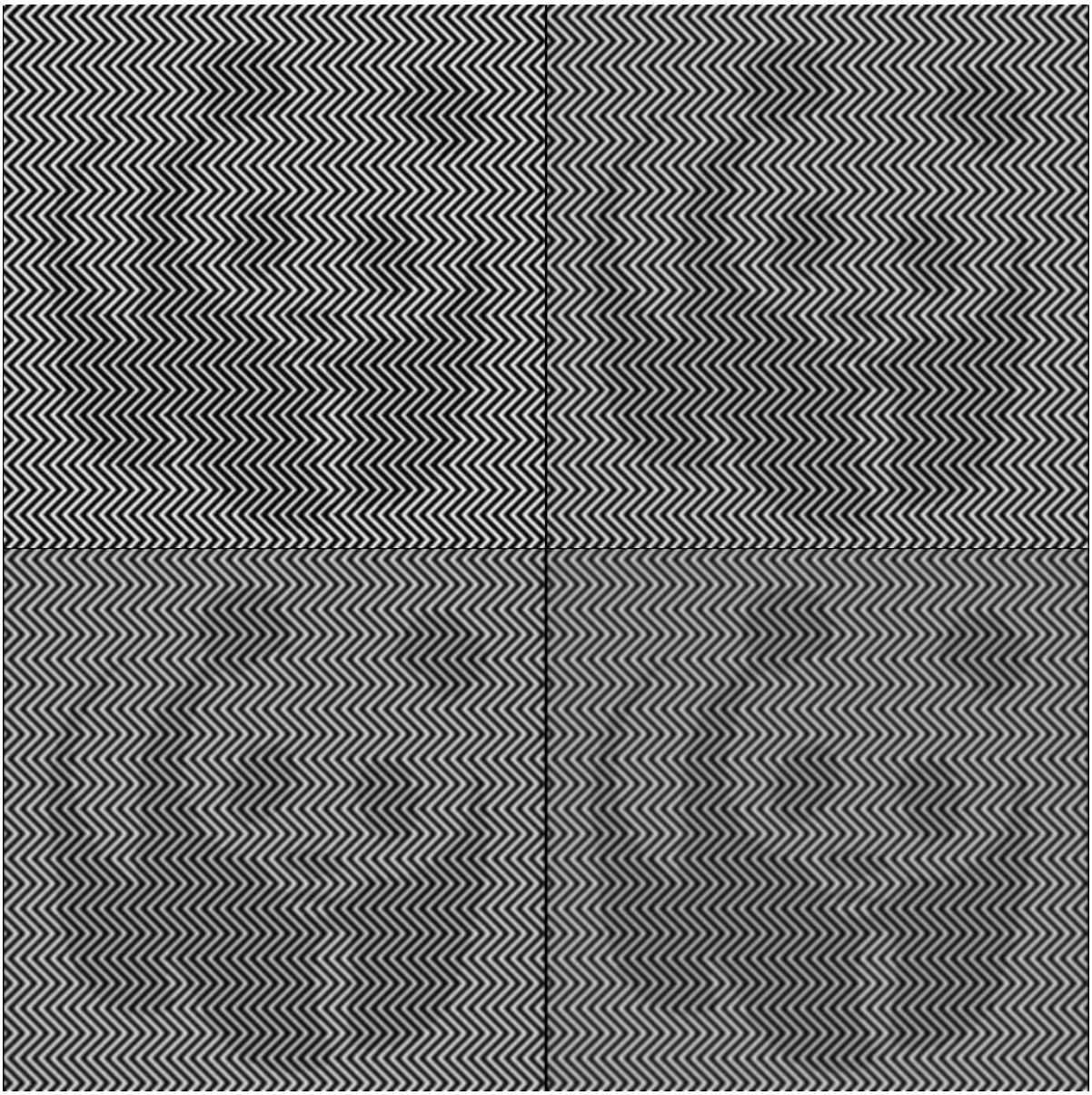
**To see implementation of gauss filter**

**./hw2 panda.pgm 1**

**To see implementation of median filter**

**./hw2 barbarasaltpepper.pgm 2**

**Gauss filtered images (Output of the program when count is equal to 1)**



**Median filtered images (Output of the program when count is equal to 2 )**

## Second Program

In the program below, first we misunderstood the assignment, so we divided the same picture into 4 parts and applied the desired number of filters to each part and created an original photograph in NC*NR size. In this program, the gaussian and median functions are the same, but the quadrant creation processes in the main is different.

```
int main(int argc , char **argv)
{
unsigned char **img , **temp_1 , **top_left , **top_right , **bottom_left , **bottom_right;
char *pgm_file;
int i , j , k , NC , NR , count;

if(argc!=3)
{
  printf("\nUsage: 627635_hw2 [Image file (*.pgm)] [count]\n");
  printf("\nE.g.   627635_hw2 panda.pgm 1\n");
  exit(-1);
}

pgm_file = argv[1]; // 2nd input for which image should be corrected
count = atoi(argv[2]); // 3rd count variable got from user as string, we convert it into integer value to use it.
                       // It will be used as a flag to choose if we want to perform Gauss operation or Median operation

img = pgm_file_to_img(pgm_file , &NC , &NR); // to save uncorrected pixel values
show_pgm_file(pgm_file); // shows uncorrected image

// img allocation part
temp_1 = alloc_img(NC , NR);        // to allocate space for corrected pixel values (same size with img)
top_left = alloc_img(NC , NR);      // to allocate space for corrected pixel values (same size with img)
top_right = alloc_img(NC , NR);     // to allocate space for corrected pixel values (same size with img)
bottom_left = alloc_img(NC , NR);   // to allocate space for corrected pixel values (same size with img)
bottom_right = alloc_img(NC , NR);  // to allocate space for corrected pixel values (same size with img)
```

Creating quadrants

```
// slicing image into 4 parts and assigning them into their labeled arrays
for(i = 0 ; i < NR ; i++)
{
    for(j = 0 ; j < NC ; j++)
    {
        if(i < NR / 2 && j < NC / 2)
        {
            top_left[i][j] = img[i][j]; // Q1 allocation from img array to the top-left array
        }
        else if(i < NR / 2 && j > NC / 2)
        {
            top_right[i][j] = img[i][j];    // Q2 allocation from img array to the top-right array
        }
        else if(i > NR / 2 && j < NC / 2)
        {
            bottom_left[i][j] = img[i][j];  // Q3 allocation from img array to the top-left array
        }
        else if(i > NR / 2 && j > NC / 2)
        {
            bottom_right[i][j] = img[i][j]; // Q4 allocation from img array to the bottom-right array
        }
    }
}
```

Calling filter functions based on count value. When count=1 gaussian filter is applied and when count=2, median filter is applied.

```
// ---------------------- GAUSSIAN FILTERING MAIN (CALLING THE FUNCTION) PART ----------------------
if(count == 1)
{
    top_left = gaussf(top_left , NC , NR , 1);      // Filtering part, for top-left, 1 gaussian filter was processed
    top_right = gaussf(top_right , NC , NR , 2);        // Filtering part, for top-right, 2 gaussian filter was processed
    bottom_left = gaussf(bottom_left , NC , NR , 3);    // Filtering part, for bottom-left, 3 gaussian filter was processed
    bottom_right = gaussf(bottom_right , NC , NR , 4);  // Filtering part, for bottom-right, 4 gaussian filter was processed
}

// ---------------------- MEDIAN FILTERING MAIN (CALLING THE FUNCTION) PART ----------------------
if(count == 2)
{
    top_left = medianf(top_left , NC , NR , 1);         // Filtering part, for top-left, 1 gaussian filter was processed
    top_right = medianf(top_right , NC , NR , 2);       // Filtering part, for top-right, 2 gaussian filter was processed
    bottom_left = medianf(bottom_left , NC , NR , 3);   // Filtering part, for bottom-left, 3 gaussian filter was processed
    bottom_right = medianf(bottom_right , NC , NR , 4); // Filtering part, for bottom-right, 4 gaussian filter was processed
}
```

We combined 4 quadrants to create an NC*NR image and assigned the temp_1 parameter.

```
// reconstruct the image by adding up the sliced and filtered pieces according to conditions mentioned below
for(i = 0 ; i < NR ; i++)
{
    for(j = 0 ; j < NC ; j++)
    {
        if(i < NR / 2 && j < NC / 2)
        {
            temp_1[i][j] = top_left[i][j];      // Q1 allocation from top-left array to the temp_1 array
        }
        else if(i < NR / 2 && j > NC / 2)
        {
            temp_1[i][j] = top_right[i][j];     // Q2 allocation from img array to the top-right array
        }
        else if(i > NR / 2 && j < NC / 2)
        {
            temp_1[i][j] = bottom_left[i][j];   // Q3 allocation from img array to the top-left array
        }
        else if(i > NR / 2 && j > NC / 2)
        {
            temp_1[i][j] = bottom_right[i][j];  // Q4 allocation from img array to the bottom-right array
        }
    }
}
// temp_1 used to reconstruct the image because, we wanted a non-destructive approach, with this, we saved the original img
```

Calling processes according to the desired function

```
// img to pgm conversion and showing part
if(count == 1)
{
    img_to_pgm_file(temp_1 , "gauss_slicing.pgm" , NC , NR);    // Converting image(matrix) to Grayscale(actual image)
    show_pgm_file("gauss_slicing.pgm");                 // Displays the filtered image
}

if(count == 2)
{
    img_to_pgm_file(temp_1 , "median_slicing.pgm" , NC , NR);   // Converting image(matrix) to Grayscale(actual image)
    show_pgm_file("median_slicing.pgm");                // Displays the filtered image
}

// img free part
free_img(top_left);
free_img(top_right);
free_img(bottom_left);
free_img(bottom_right);
free_img(img);
free_img(temp_1);

return(1);
}
```

To run our code, we used the following commands to compile and execute;

**gcc slicing_try.c -o slicing_try img_pro.c -lm**

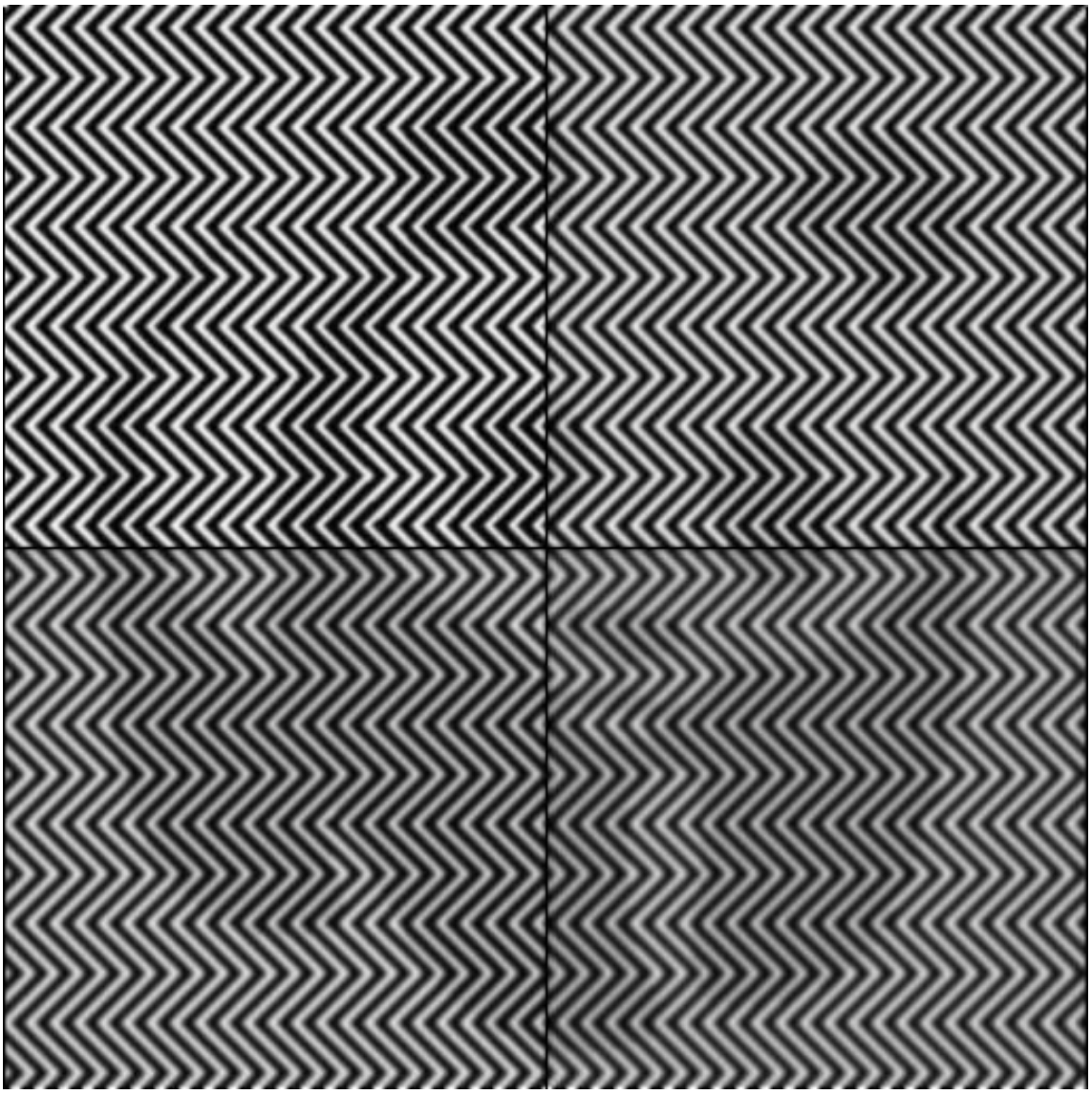
**To see implementation of gauss filter**

**. /slicing_try panda.pgm 1**

**To see implementation of median filter**

**. /slicing_try barbarasaltpepper.pgm 2**

**Gauss Slicing**



**Median Slicing**