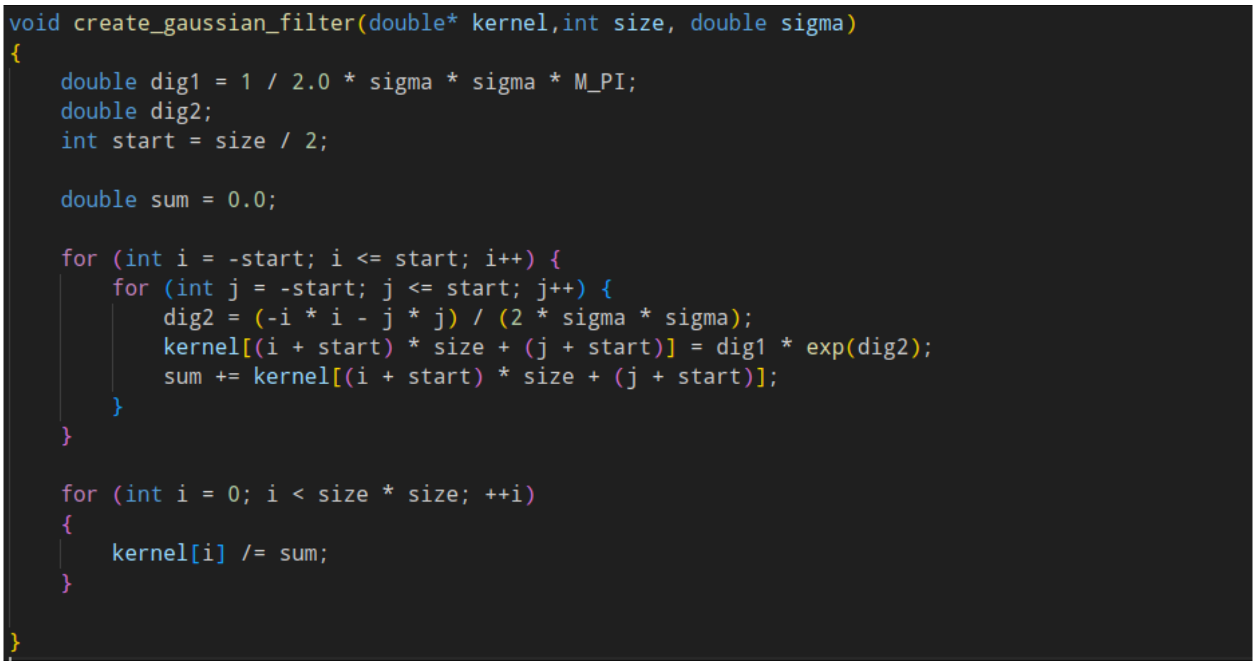
Multimedia Report:

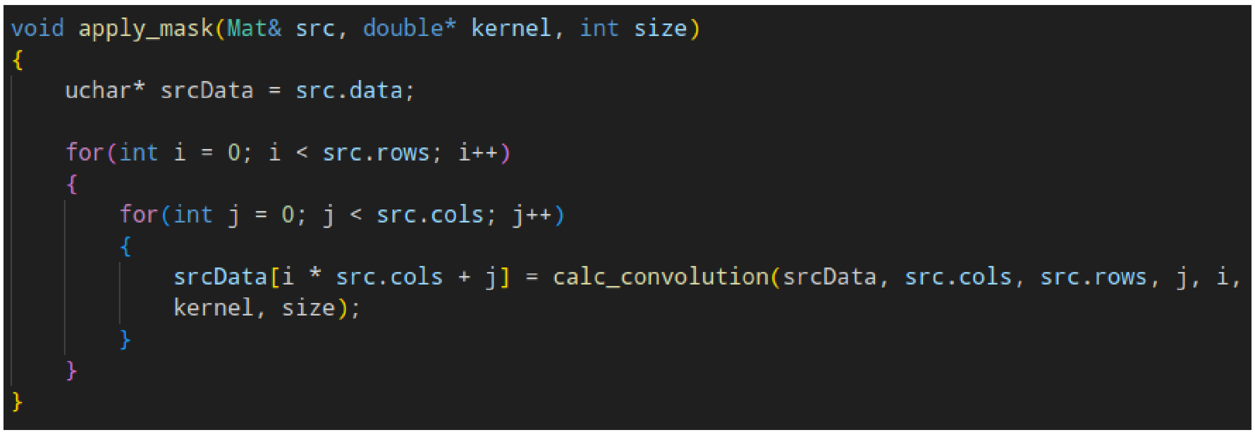
Task 1:

Code provided is implementing the Gaussian blur on input Image. In order to image to become blurred the kernel provided by functions given below has to be convolved with image:



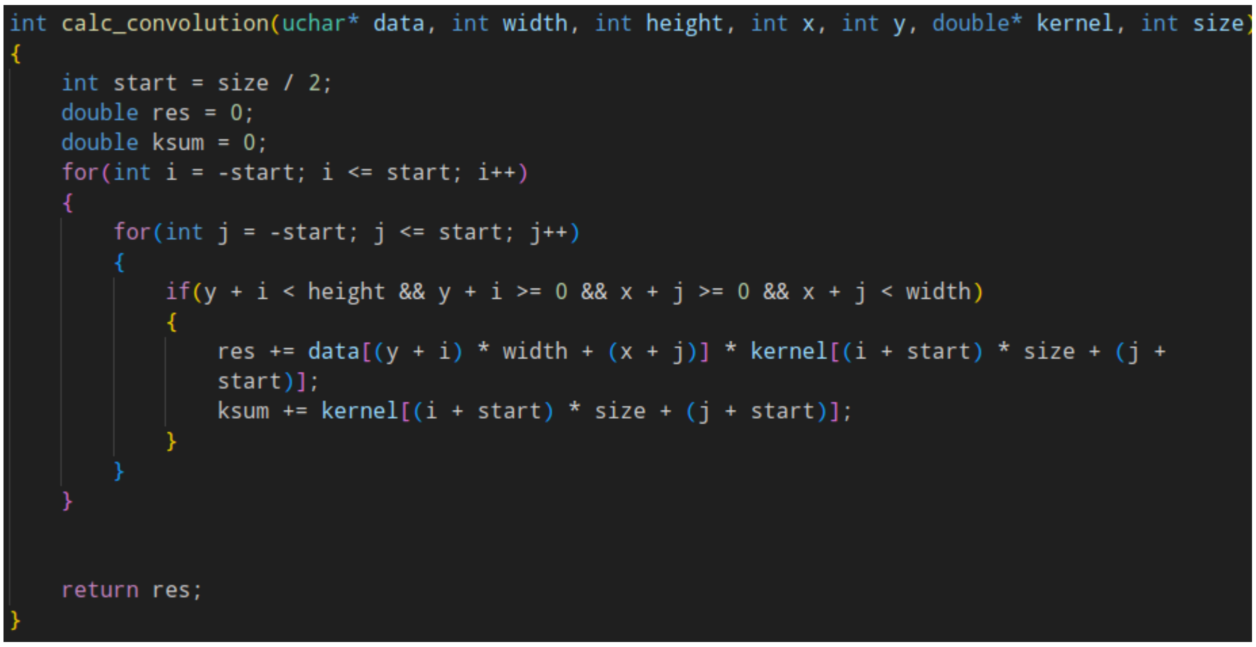
```
void create_gaussian_filter(double* kernel,int size, double sigma)
{
    double dig1 = 1 / 2.0 * sigma * sigma * M_PI;
    double dig2;
    int start = size / 2;

    double sum = 0.0;

    for (int i = -start; i <= start; i++) {
        for (int j = -start; j <= start; j++) {
            dig2 = (-i * i - j * j) / (2 * sigma * sigma);
            kernel[(i + start) * size + (j + start)] = dig1 * exp(dig2);
            sum += kernel[(i + start) * size + (j + start)];
        }
    }

    for (int i = 0; i < size * size; ++i)
    {
        kernel[i] /= sum;
    }

}
```

The create filter function generates a Gaussian filter kernel based on the given parameters. It calculates the kernel values based on the Gaussian distribution and normalizes them.



```
void apply_mask(Mat& src, double* kernel, int size)
{
    uchar* srcData = src.data;

    for(int i = 0; i < src.rows; i++)
    {
        for(int j = 0; j < src.cols; j++)
        {
            srcData[i * src.cols + j] = calc_convolution(srcData, src.cols, src.rows, j, i,
            kernel, size);
        }
    }
}
```

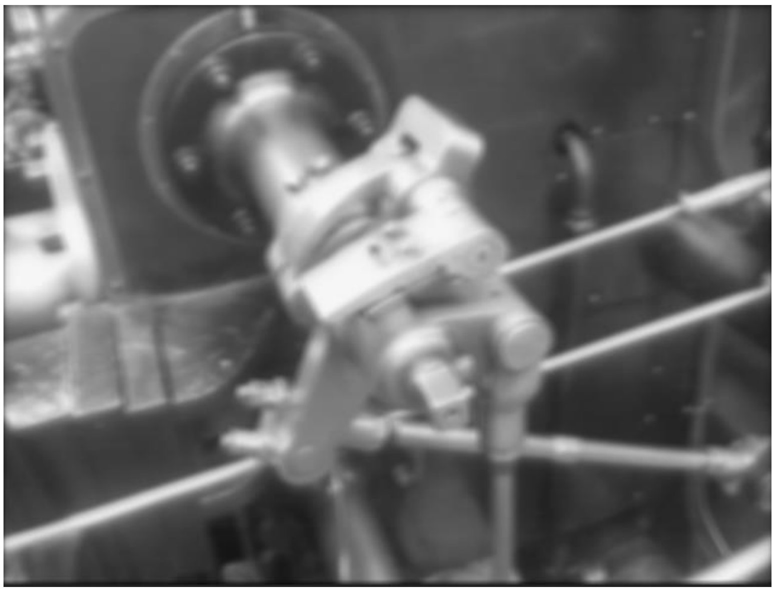
The apply_mask function applies a given kernel to an image using the convolution operation.

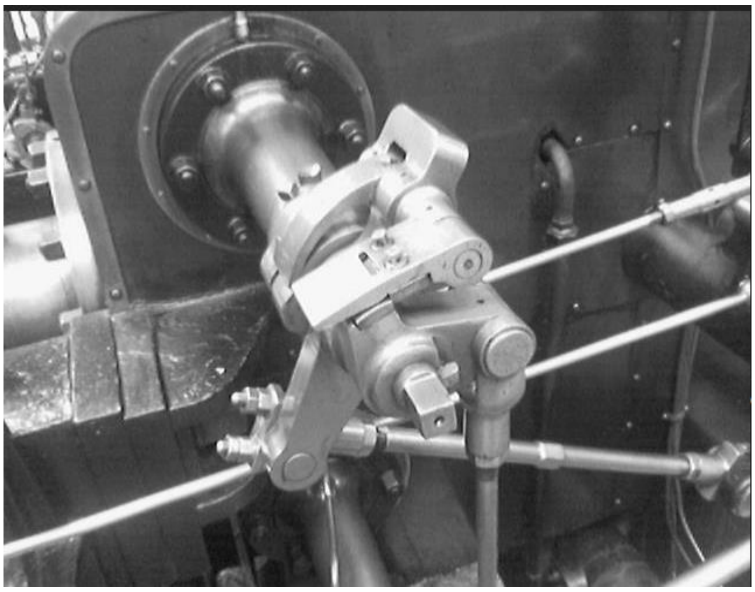


```
int calc_convolution(uchar* data, int width, int height, int x, int y, double* kernel, int size)
{
    int start = size / 2;
    double res = 0;
    double ksum = 0;
    for(int i = -start; i <= start; i++)
    {
        for(int j = -start; j <= start; j++)
        {
            if(y + i < height && y + i >= 0 && x + j >= 0 && x + j < width)
            {
                res += data[(y + i) * width + (x + j)] * kernel[(i + start) * size + (j +
                start)];
                ksum += kernel[(i + start) * size + (j + start)];
            }
        }
    }


    return res;
}
```

The calc_convolution function calculates the convolution of an image using a specified kernel.

Result of the code can be observed below:

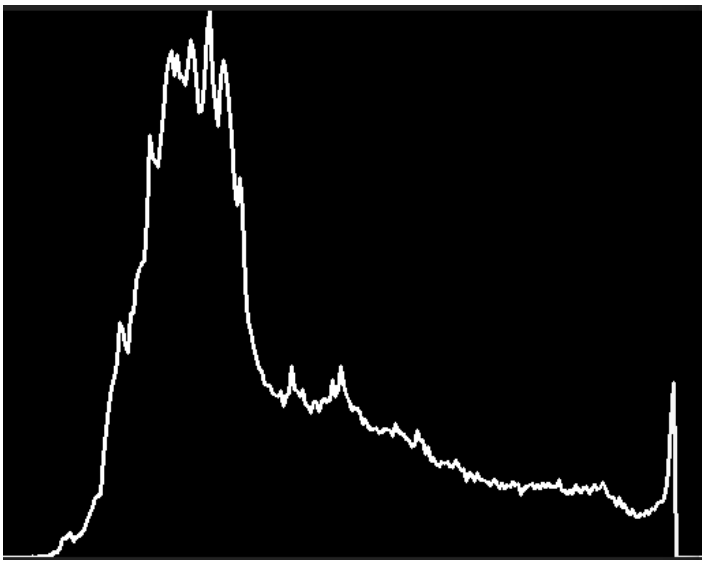




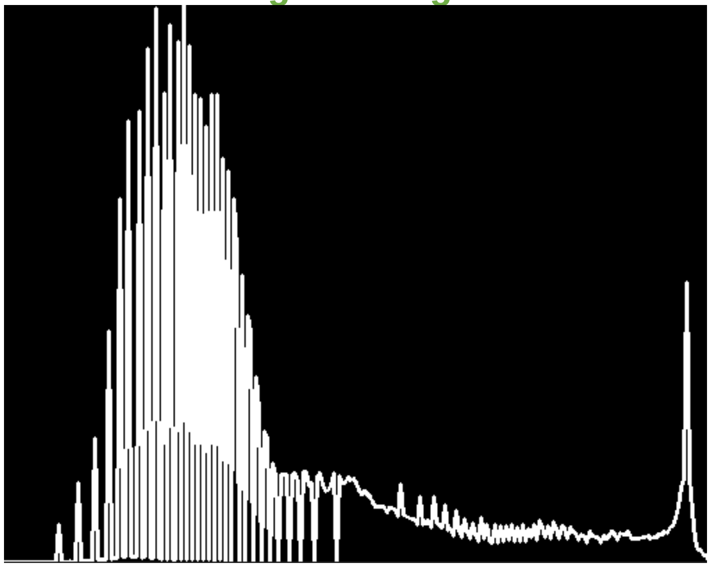
Task 2:

Change in histogram of both images seen in previous task are provided below:

Blurred Image



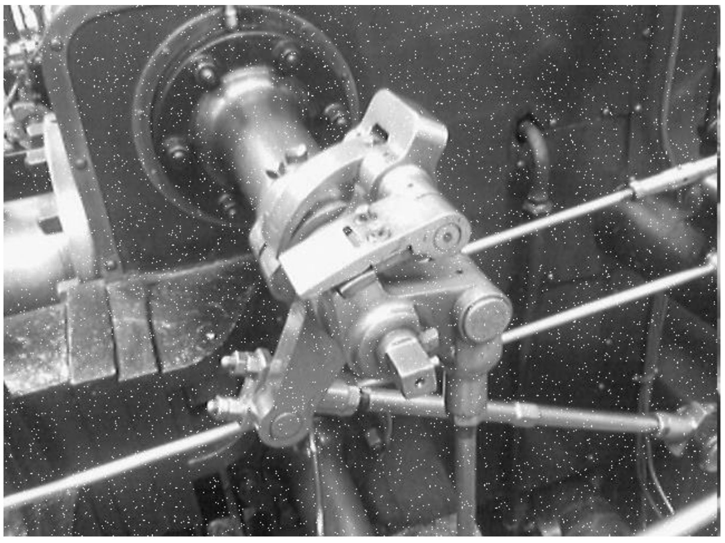
Original Image



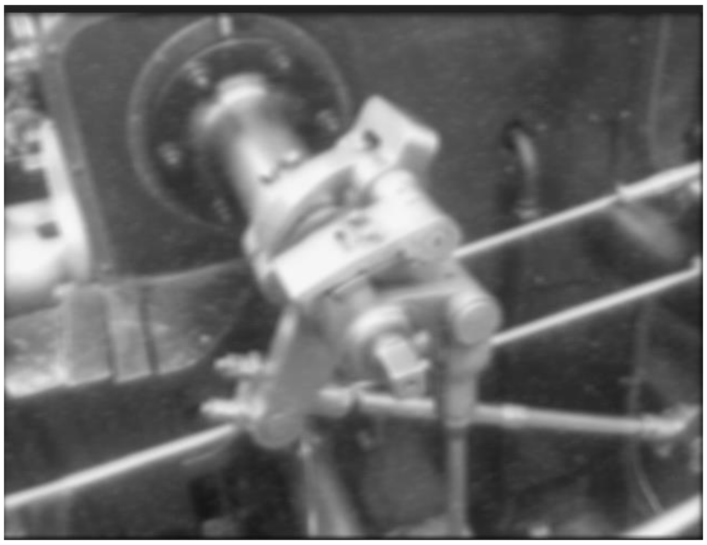
Task 3:

Original, noised and blurred images are shown below:

Noisy Image

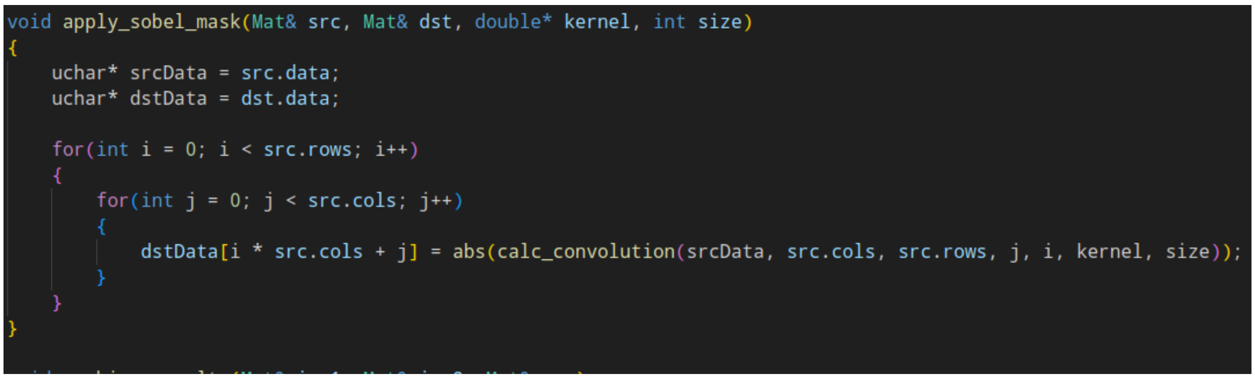


Blurred Image



Task 4:

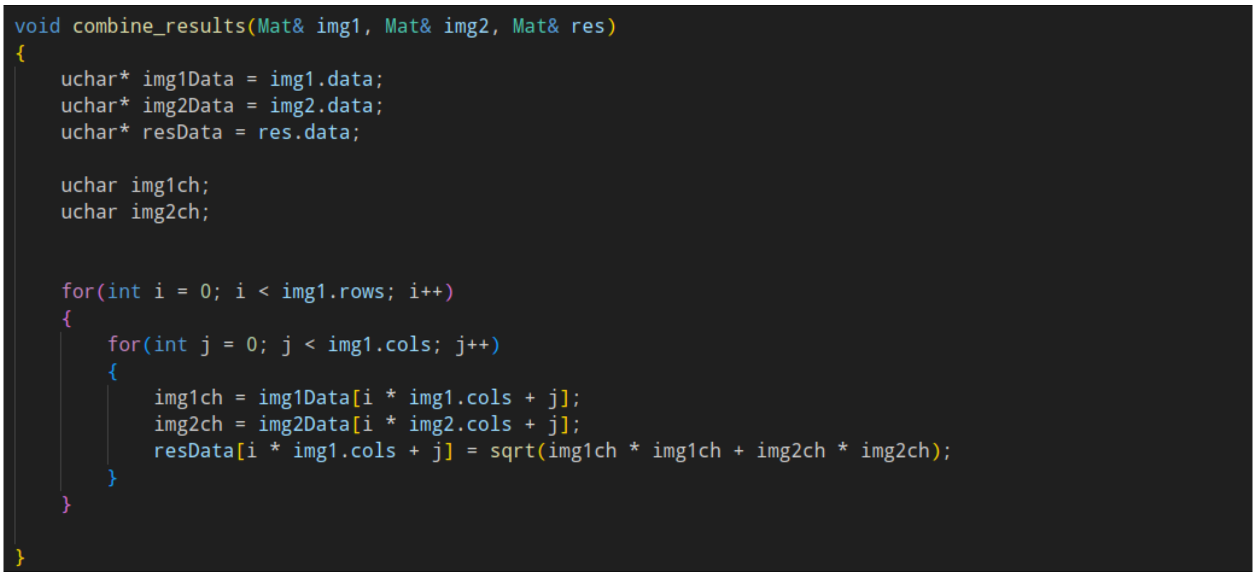
In order to apply Sobel filter some slight changes to convolution code should be applied:



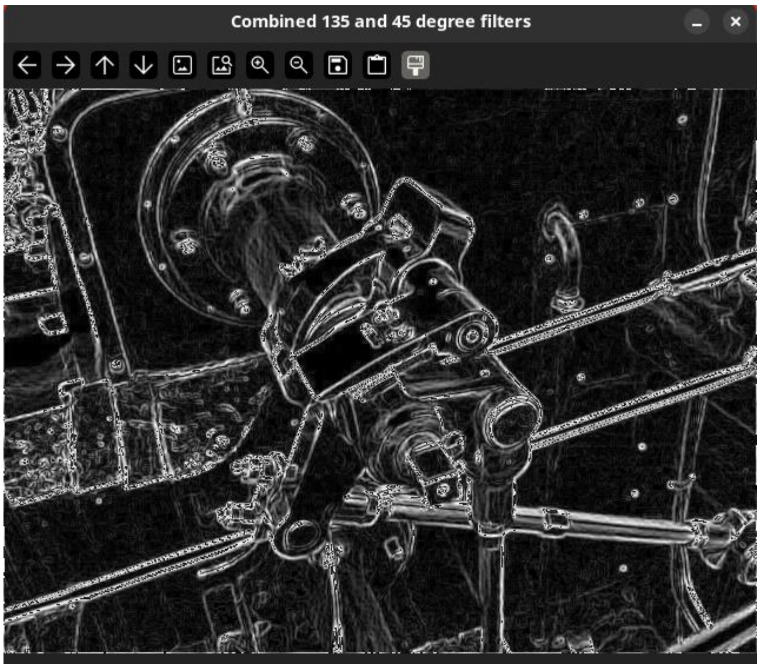
```
void apply_sobel_mask(Mat& src, Mat& dst, double* kernel, int size)
{
    uchar* srcData = src.data;
    uchar* dstData = dst.data;

    for(int i = 0; i < src.rows; i++)
    {
        for(int j = 0; j < src.cols; j++)
        {
            dstData[i * src.cols + j] = abs(calc_convolution(srcData, src.cols, src.rows, j, i, kernel, size));
        }
    }
}
```

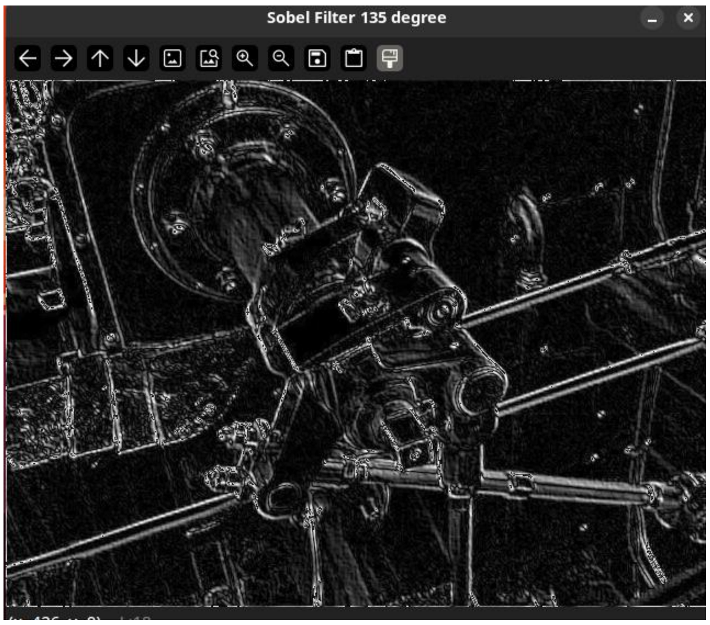
We took absolute value of convolution result

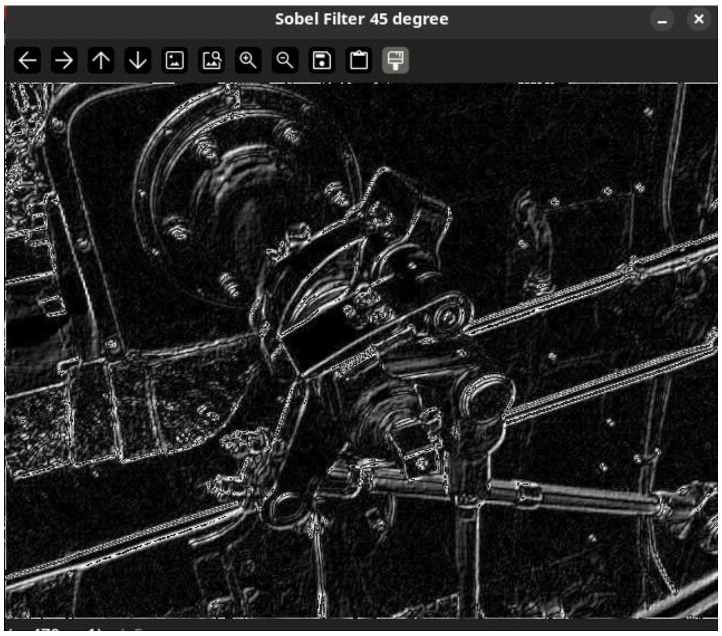
Code and Result of applying 135 and 45 degree filters are provided below:



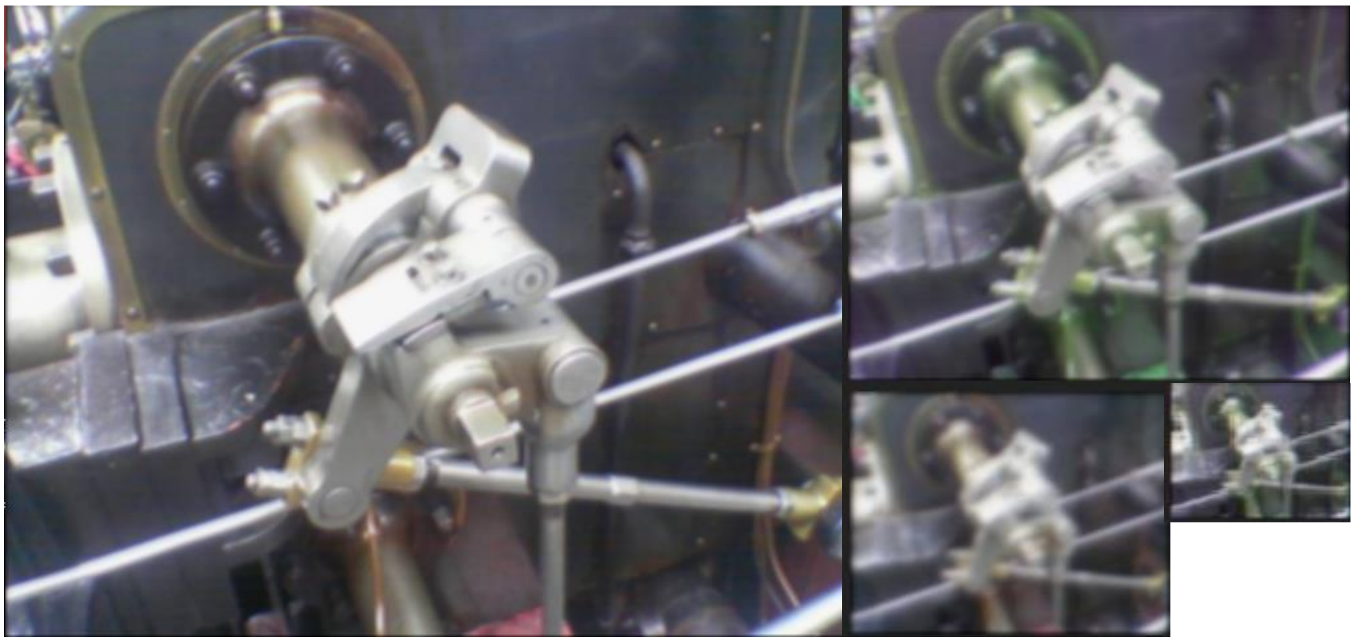
```
void combine_results(Mat& img1, Mat& img2, Mat& res)
{
    uchar* img1Data = img1.data;
    uchar* img2Data = img2.data;
    uchar* resData = res.data;

    uchar img1ch;
    uchar img2ch;


    for(int i = 0; i < img1.rows; i++)
    {
        for(int j = 0; j < img1.cols; j++)
        {
            img1ch = img1Data[i * img1.cols + j];
            img2ch = img2Data[i * img2.cols + j];
            resData[i * img1.cols + j] = sqrt(img1ch * img1ch + img2ch * img2ch);
        }
    }

}
```





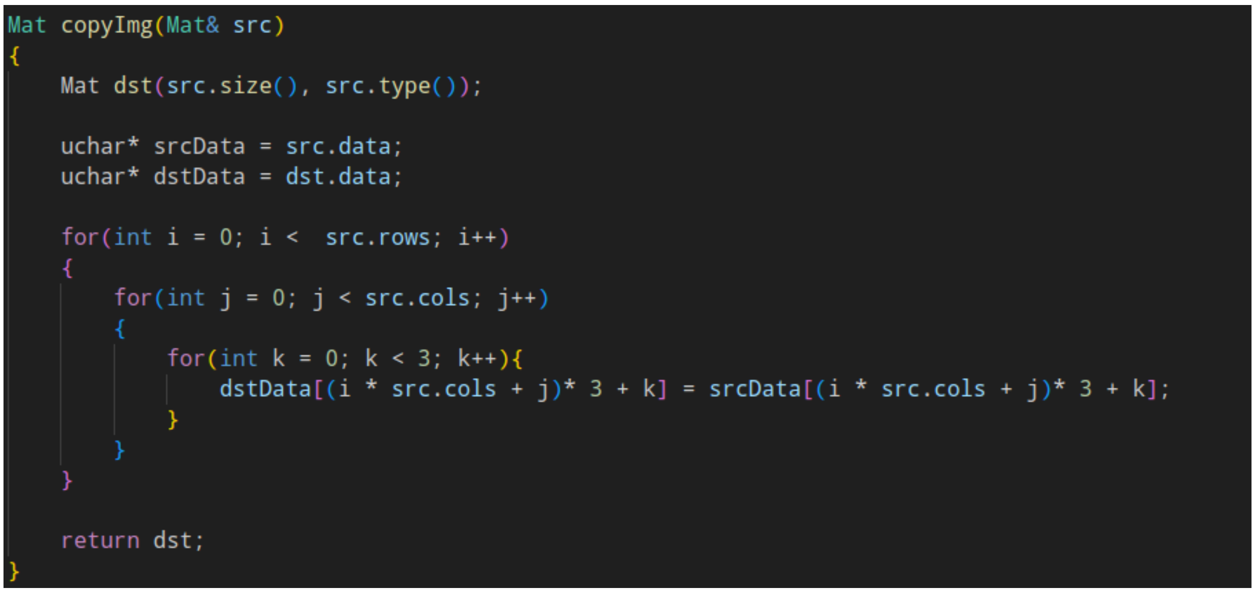


Task 5:



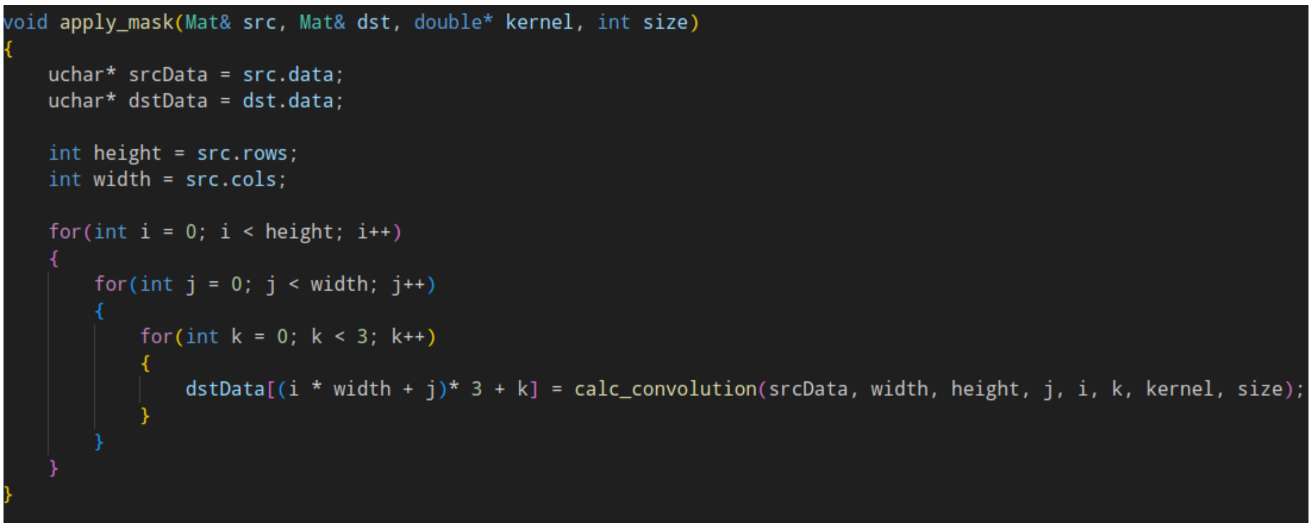
Gaussian Pyramid

The result of task 5 is provided above. In order to achieve this result the convolution and sampling and copying code should be adapted to considering RGB channels. The code for all necessary functions are provided below:



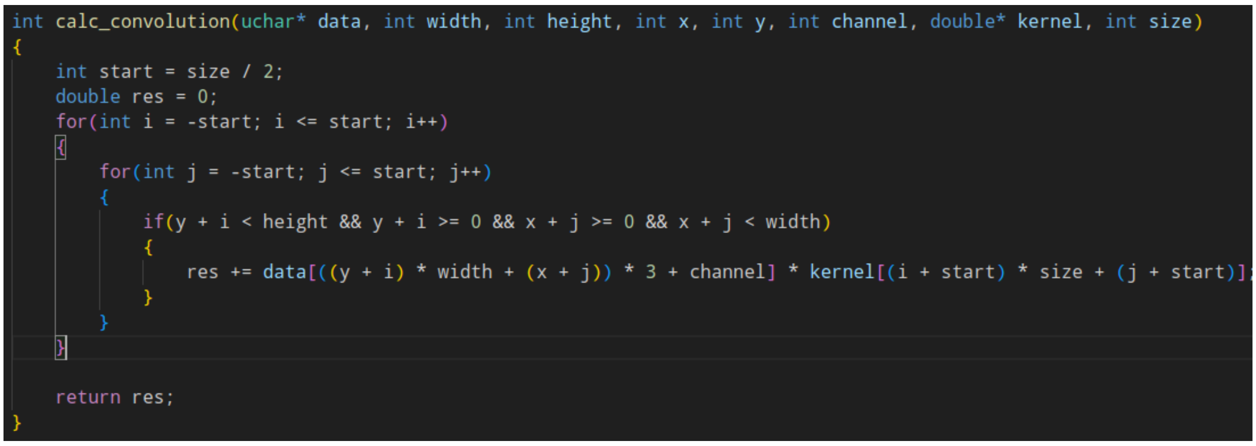
```
Mat copyImg(Mat& src)
{
    Mat dst(src.size(), src.type());

    uchar* srcData = src.data;
    uchar* dstData = dst.data;

    for(int i = 0; i <  src.rows; i++)
    {
        for(int j = 0; j < src.cols; j++)
        {
            for(int k = 0; k < 3; k++){
                dstData[(i * src.cols + j)* 3 + k] = srcData[(i * src.cols + j)* 3 + k];
            }
        }
    }

    return dst;
}
```

Mat copyImg(Mat& src): This function takes an input Mat (src) and creates a copy (dst) of the input image with the same size and data type. The nested loops (i, j, and k) iterate over the rows, columns, and color channels of the image.



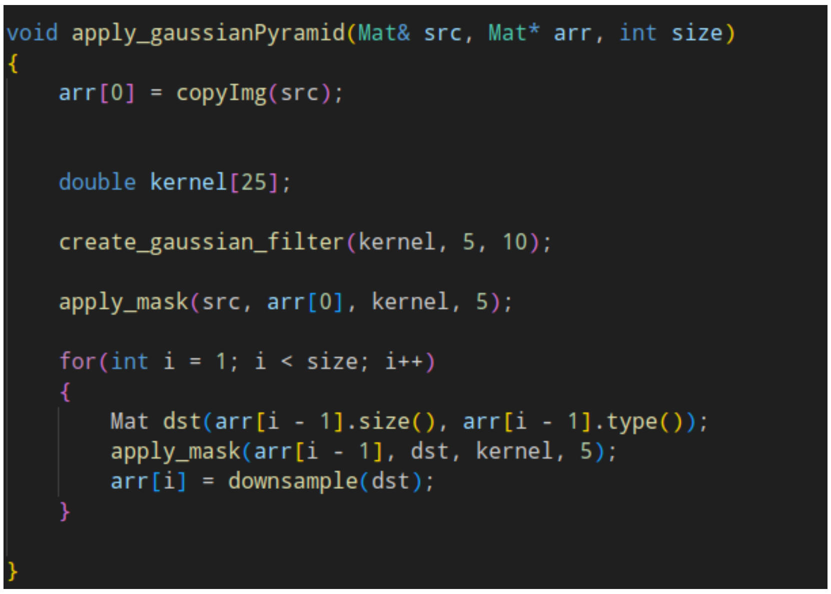
```
void apply_mask(Mat& src, Mat& dst, double* kernel, int size)
{
    uchar* srcData = src.data;
    uchar* dstData = dst.data;

    int height = src.rows;
    int width = src.cols;

    for(int i = 0; i < height; i++)
    {
        for(int j = 0; j < width; j++)
        {
            for(int k = 0; k < 3; k++)
            {
                dstData[(i * width + j)* 3 + k] = calc_convolution(srcData, width, height, j, i, k, kernel, size);
            }
        }
    }
}
```

The function's purpose is to apply the convolution filter kernel to each color channel of the input image (src) and store the resulting filtered image in the output image (dst). This function is suitable for color images with three channels (RGB).



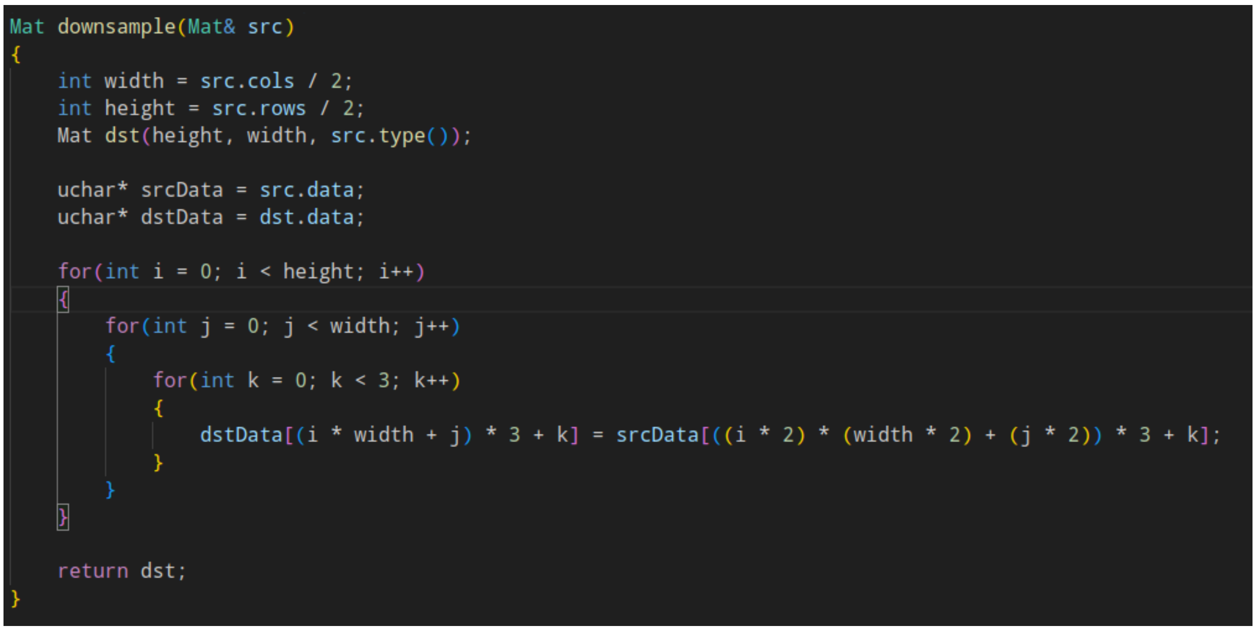
```
int calc_convolution(uchar* data, int width, int height, int x, int y, int channel, double* kernel, int size)
{
    int start = size / 2;
    double res = 0;
    for(int i = -start; i <= start; i++)
    {
        for(int j = -start; j <= start; j++)
        {
            if(y + i < height && y + i >= 0 && x + j >= 0 && x + j < width)
            {
                res += data[((y + i) * width + (x + j)) * 3 + channel] * kernel[(i + start) * size + (j + start)];
            }
        }
    }

    return res;
}
```

This function performs a convolution operation for a specific color channel of an image using a given kernel. The convolution result is a weighted sum of pixel values in the neighborhood of the specified pixel location.



```
void apply_gaussianPyramid(Mat& src, Mat* arr, int size)
{
    arr[0] = copyImg(src);


    double kernel[25];

    create_gaussian_filter(kernel, 5, 10);

    apply_mask(src, arr[0], kernel, 5);

    for(int i = 1; i < size; i++)
    {
        Mat dst(arr[i - 1].size(), arr[i - 1].type());
        apply_mask(arr[i - 1], dst, kernel, 5);
        arr[i] = downsample(dst);
    }

}
```

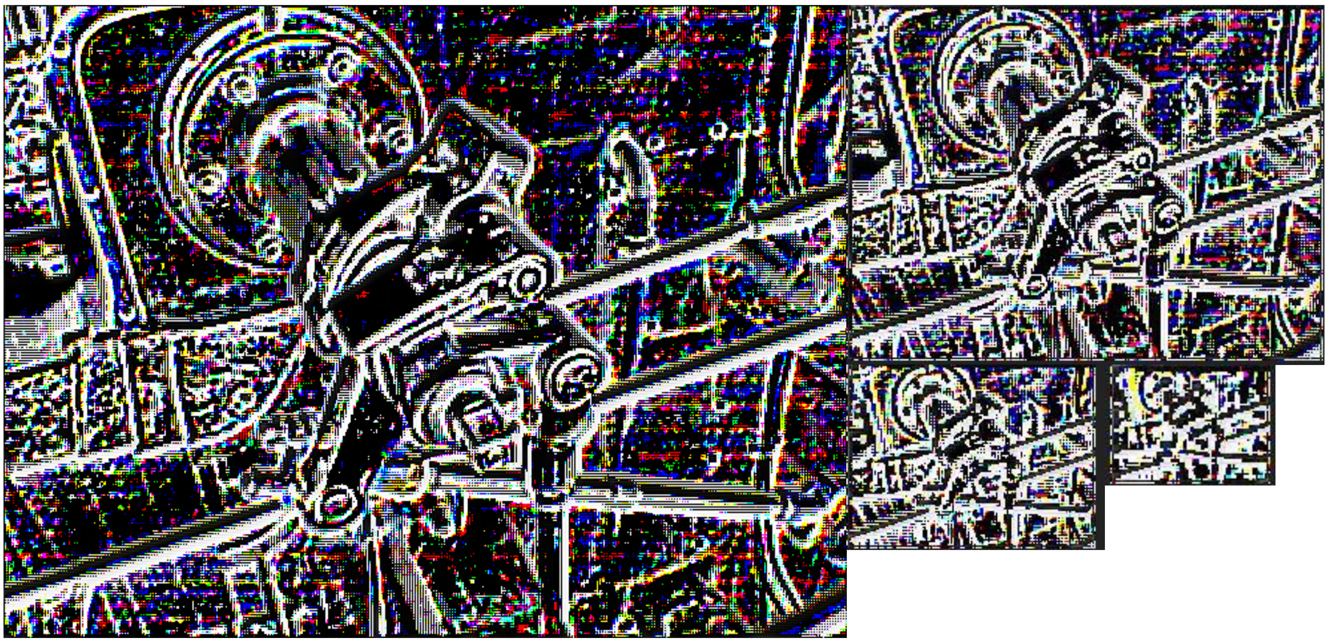
This loop inside the function iterates to create each level of the Gaussian pyramid, starting with the original image and then progressively smoothing and down scaling the image for each subsequent level.



```
Mat downsample(Mat& src)
{
    int width = src.cols / 2;
    int height = src.rows / 2;
    Mat dst(height, width, src.type());

    uchar* srcData = src.data;
    uchar* dstData = dst.data;

    for(int i = 0; i < height; i++)
    {
        for(int j = 0; j < width; j++)
        {
            for(int k = 0; k < 3; k++)
            {
                dstData[(i * width + j) * 3 + k] = srcData[((i * 2) * (width * 2) + (j * 2)) * 3 + k];
            }
        }
    }

    return dst;
}
```

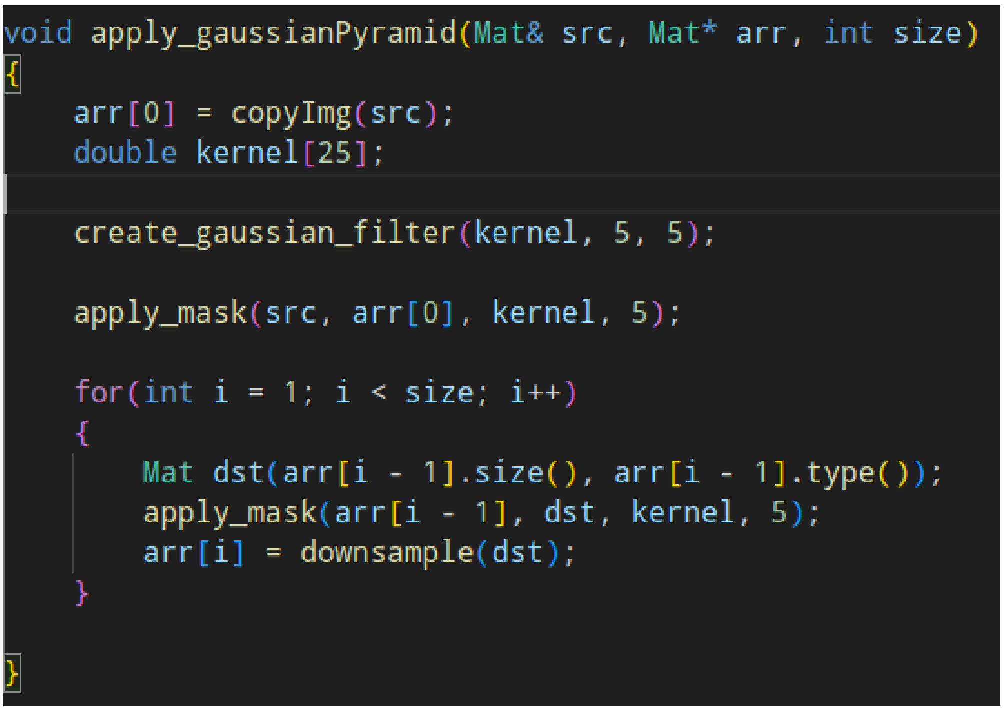
The down-sample function is designed to reduce the size of an input RGB image by a factor of 2 in both the horizontal and vertical dimensions. It takes an input image src and returns a down-sampled version of that image.

Task 6:



Images above represent the 4 level laplacian pyramid applied on input image.Explanation of whole process is provided below!

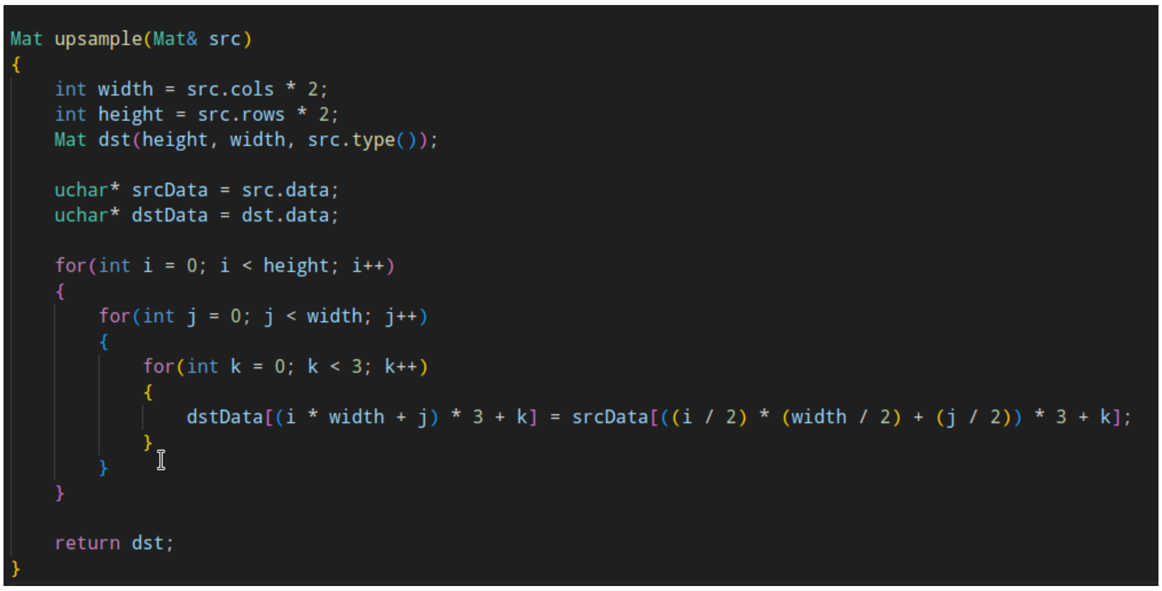
In the first part gaussian pyramid should be applied on input image and result have to be saved:



```
void apply_gaussianPyramid(Mat& src, Mat* arr, int size)
{
    arr[0] = copyImg(src);
    double kernel[25];

    create_gaussian_filter(kernel, 5, 5);

    apply_mask(src, arr[0], kernel, 5);

    for(int i = 1; i < size; i++)
    {
        Mat dst(arr[i - 1].size(), arr[i - 1].type());
        apply_mask(arr[i - 1], dst, kernel, 5);
        arr[i] = downsample(dst);
    }

}
```

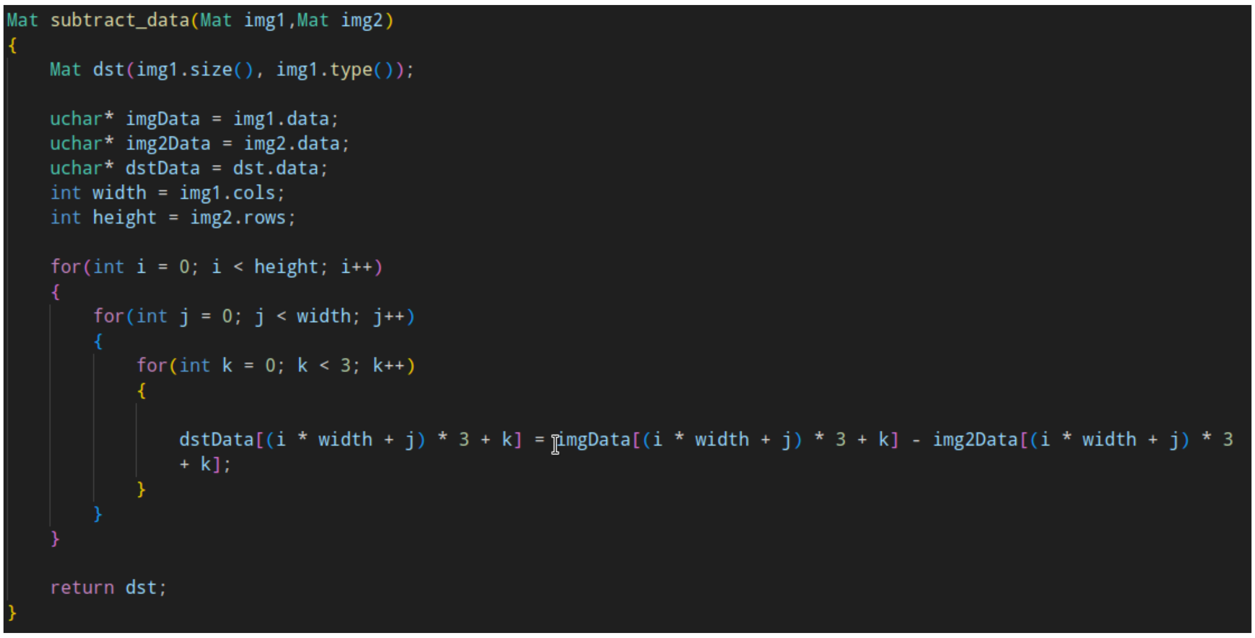
In second part i level of gaussian pyramid should be up-scaled (up-sampled) and corresponding pixels have to be subtracted:

Up-sample



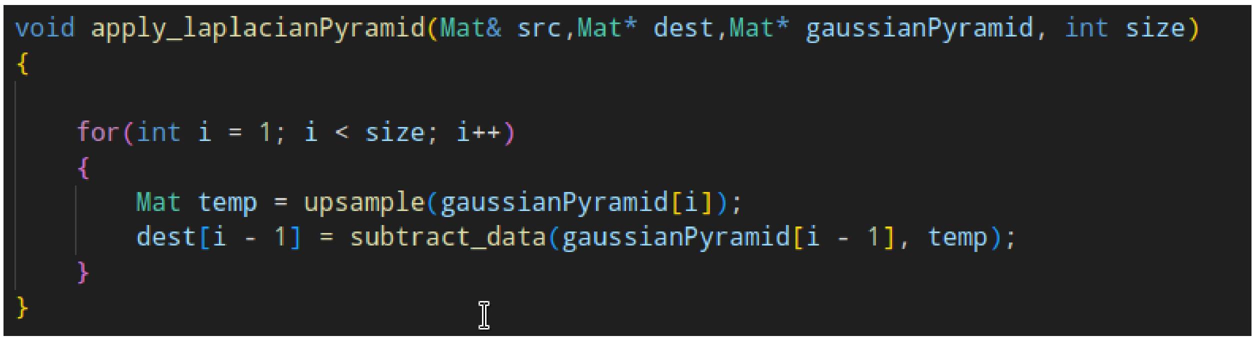
```
Mat upsample(Mat& src)
{
    int width = src.cols * 2;
    int height = src.rows * 2;
    Mat dst(height, width, src.type());

    uchar* srcData = src.data;
    uchar* dstData = dst.data;

    for(int i = 0; i < height; i++)
    {
        for(int j = 0; j < width; j++)
        {
            for(int k = 0; k < 3; k++)
            {
                dstData[(i * width + j) * 3 + k] = srcData[((i / 2) * (width / 2) + (j / 2)) * 3 + k];
            }
        }
    }

    return dst;
}
```

Subtraction



```
Mat subtract_data(Mat img1,Mat img2)
{
    Mat dst(img1.size(), img1.type());

    uchar* imgData = img1.data;
    uchar* img2Data = img2.data;
    uchar* dstData = dst.data;
    int width = img1.cols;
    int height = img2.rows;

    for(int i = 0; i < height; i++)
    {
        for(int j = 0; j < width; j++)
        {
            for(int k = 0; k < 3; k++)
            {

                dstData[(i * width + j) * 3 + k] = imgData[(i * width + j) * 3 + k] - img2Data[(i * width + j) * 3
                + k];
            }
        }
    }

    return dst;
}
```

Taking previous two facts into account the Laplacian pyramid can be represented as follows:



```
void apply_laplacianPyramid(Mat& src,Mat* dest,Mat* gaussianPyramid, int size)
{

    for(int i = 1; i < size; i++)
    {
        Mat temp = upsample(gaussianPyramid[i]);
        dest[i - 1] = subtract_data(gaussianPyramid[i - 1], temp);
    }
}
```

<u>Student Name:</u> Javid Guliyev

<u>Student ID:</u> 12235421