Computer Vision

"Task 1"
Image Processing Studio

Introduction

In this report, you will be able to know the algorithms that are implemented in the backend of the project using C++. Also, the output samples of each function (an image showing the applied filter or noise, etc...) and finally comments that analyze these results.

Team Members:

Magdy Nasr - Ahmed Emad - Youssef Kadry - Mohab Ghobashy - Mohammed Mostafa

Team No. 3

Tab 1 - Filtering

Page Contents

- Adding Salt & Pepper Noise
- Applying Median Filter
- Adding Gaussian Noise
- Applying Gaussian Filter
- Adding Average Noise
- Applying Box Filter

Algorithms

Adding Salt and Pepper Noise Algorithm:

```
void Add_salt_pepper_Noise(Mat& img, float pSalt, float pPaper)
{
    RNG rng;
    // Calculating the percentage of pixels user want to convert to salt or paper
    int amount1 = img.rows * img.cols * (pSalt/100);
    int amount2 = img.rows * img.cols * (pPaper/100);

    // Choosing random index -in the boundary of the rows and columns of course- and then put to 0 or 255
    for (int counter = 0; counter < amount1; ++counter)
        img.at<uchar>(rng.uniform(0, img.rows), rng.uniform(0, img.cols)) = 255;

    for (int counter = 0; counter < amount2; ++counter)
        img.at<uchar>(rng.uniform(0, img.rows), rng.uniform(0, img.cols)) = 0;
}
```

Adding Gaussian Noise Algorithm:

```
void Add_gaussian_Noise(Mat& img, double mean, double sigma)
{
    Mat noise(img.size(),img.type());
    randn(noise, mean, sigma);
    img += noise;
}
```

Adding Average Noise Algorithm:

```
void add_uniform_noise(Mat& img)
{
    Mat out_img;
    Mat norm_img;
    img.convertTo(img, CV_64FC1);
    norm_img = img / 255;
    Mat noise(norm_img.size(), CV_64FC1);
    randu(noise, 0, 1);
    out_img = norm_img + noise;
    normalize(out_img, out_img, 0.0, 1.0, cv::NORM_MINMAX, CV_64FC1);
    out_img *= 255;
    out_img.convertTo(out_img, CV_8UC1);
    img = out_img;
}
```

Median Filter Algorithm:

```
void medianFilter(Mat& image)
{
    int window[9];
    for (int i = 1; i < image.rows - 1; i++)
        for (int j = 1; j < image.cols - 1; j++)
        {
            window[0] = image.at<uchar>(i - 1, j - 1);
            window[1] = image.at<uchar>(i, j - 1);
            window[2] = image.at<uchar>(i + 1, j - 1);

            window[3] = image.at<uchar>(i - 1, j);
            window[4] = image.at<uchar>(i, j);
            window[5] = image.at<uchar>(i + 1, j);

            window[6] = image.at<uchar>(i - 1, j + 1);
            window[7] = image.at<uchar>(i, j + 1);
            window[8] = image.at<uchar>(i + 1, j + 1);
            insertionSort(window, 9);
            image.at<uchar>(i, j) = window[4];
        }
}
```

Box Filter Algorithm:

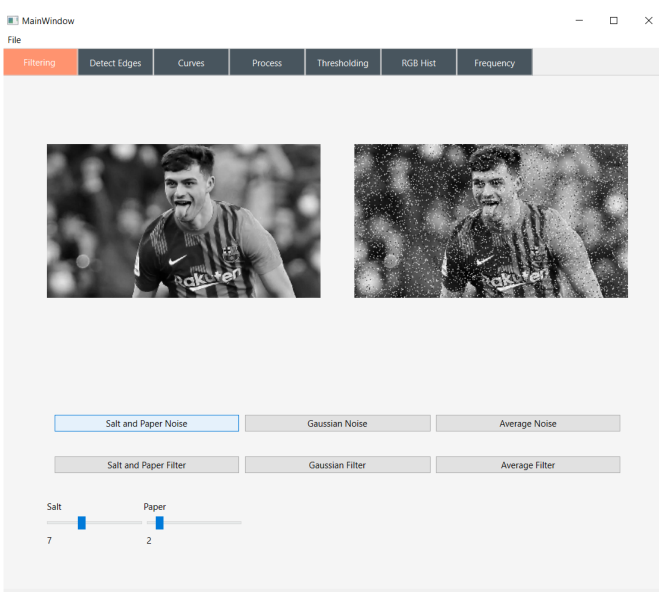
```
void boxFilter(Mat& img, int k_width, int k_height)
{
    if(k_width%2==0){
        k_width=k_width+1;
        k_height=k_height+1;
    }

    Mat pad_img, kernel;
    pad_img = padding(img, k_width, k_height);
    kernel = define_kernel_box(k_width, k_height);

    Mat output = Mat::zeros(img.size(), CV_64FC1);

    for (int i = 0; i < img.rows; i++)
        for (int j = 0; j < img.cols; j++)
            output.at<double>(i, j) = sum(kernel.mul(pad_img(Rect(j, i, k_width, k_height)))).val[0];

    output.convertTo(img, CV_8UC1);
}
```

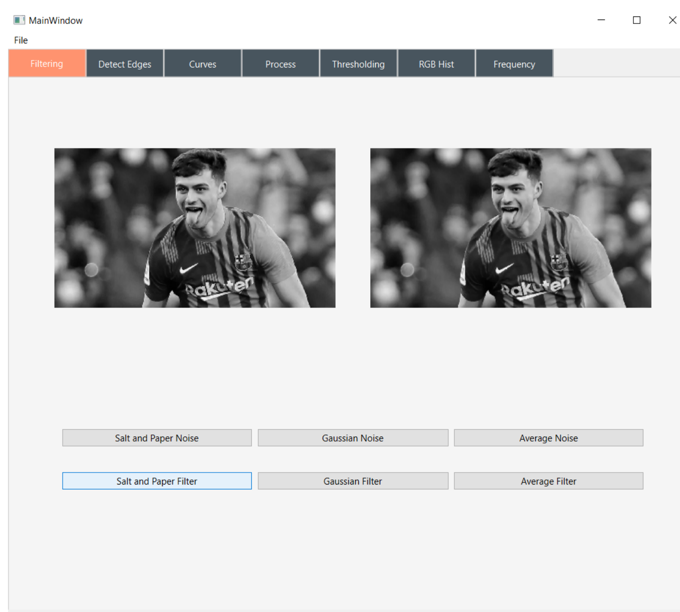
Gaussian Filter Algorithm:

```
void gaussianFilter(Mat& img, int k_w, int k_h, int sigma)
{
    if(k_w%2==0){
        k_w=k_w+1;
        k_h=k_h+1;
    }

    Mat pad_img, kernel;
    pad_img = padding(img, k_w, k_h);
    kernel  = define_kernel_gaussian(k_w, k_h, sigma);

    Mat output = Mat::zeros(img.size(), CV_64FC1);

    for (int i = 0; i < img.rows; i++)
        for (int j = 0; j < img.cols; j++)
            output.at<double>(i, j) = sum(kernel.mul(pad_img(Rect(j, i, k_w, k_h)))).val[0];

    output.convertTo(img, CV_8UC1);
}
```

Samples

1- Adding Salt and Pepper Noise and applying Median Filter:

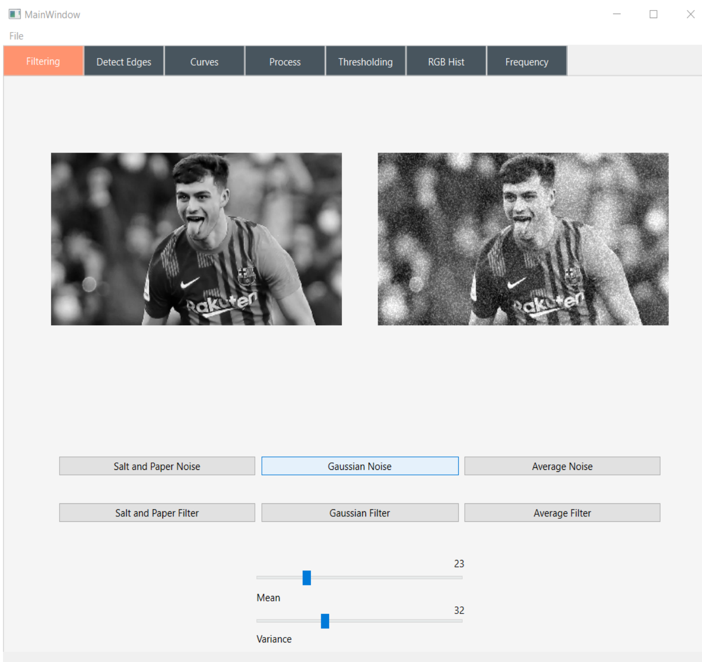


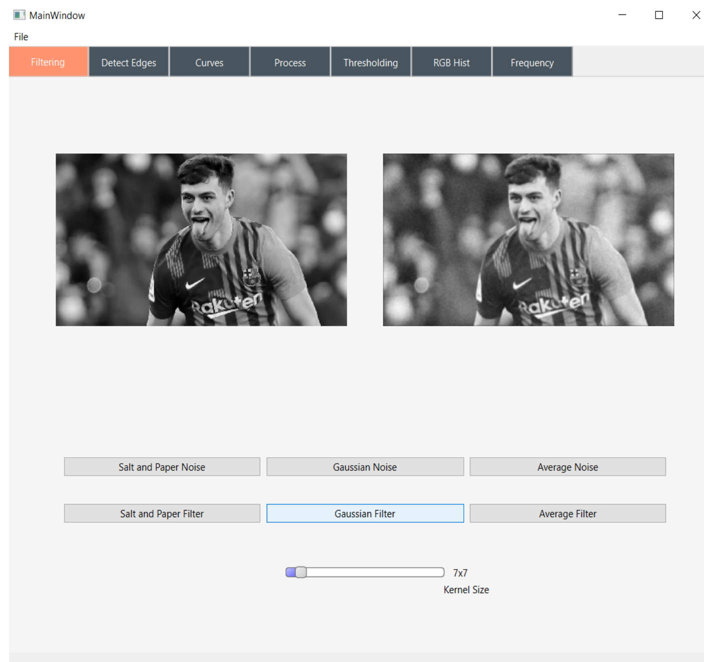


Output Analysis

We notice the salt noise is higher than paper as the user determined the ratio to be 7:2 for the salt. Also after applying the median filter the image is back to its original state.

2- Adding Gaussian Noise and Applying Gaussian Filter:

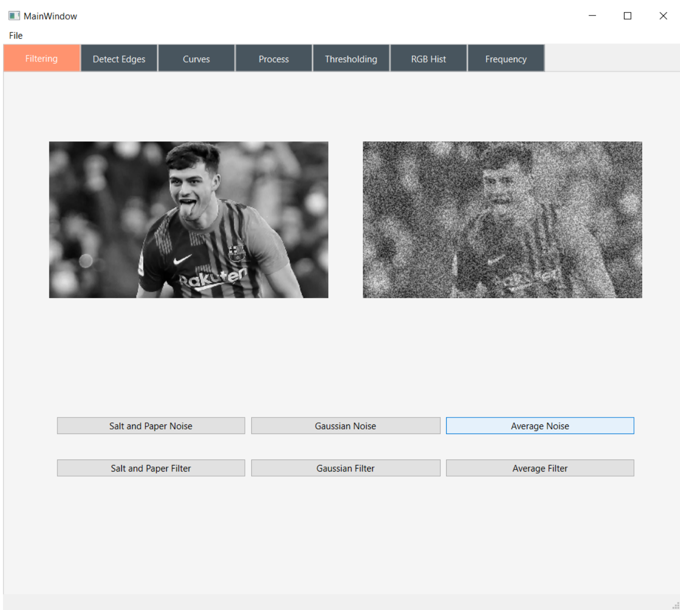


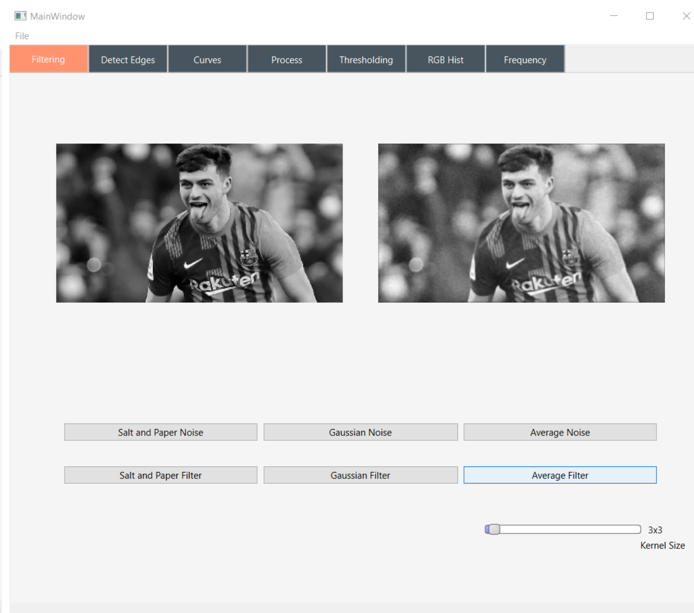


Output Analysis

Here the user chose the mean of gaussian noise equal to 23 and the variance equal to 32. Also, the user chose a kernel 7x7 which worked efficiently.

3- Adding Average Noise and Applying Box Filter:





Output Analysis

Here the user added average noise to the picture and then applied a box filter with kernel size 3x3.

Tab 2 - Edge Detection

Page Contents

- Sobel Edge Detection
- Roberts Edge Detection
- Prewitt Edge Detection
- Canny Edge Detection

Algorithms

In the first 3 Edge Detection Methods :

- First We create the kernel/mask:

```
int(*(getArray)(std::string mode, std::string direction))[3]{
    if (mode == "Sobel") {
        if (direction == "horizontal") {
            static int mask[3][3] = { {-1,-2,-1},{0,0,0},{1,2,1} };
            return mask;
        }
        else {
            static int mask[3][3] = { {-1,0,1},{-2,0,2},{-1,0,1} };
            return mask;
        }
    }
if (mode == "Roberts") {
    if (direction == "horizontal") {
        static int mask[3][3] = { {0,0,0},{0,-1,0},{0,0,1} };
        return mask;
    }
    else {
        static int mask[3][3] = { {0,0,0},{0,1,0},{0,0,-1} };
        return mask;
    }
}
else if(mode =="Prewitt")  {
        if (direction == "horizontal") {
            static int mask[3][3] = { {-1,-1,-1},{0,0,0},{1,1,1} };
            return mask;
        }
        else {
            static int mask[3][3] = { {-1,0,1},{-1,0,1},{-1,0,1} };
            return mask;
        }
    }
}
```

- Convolution over image with the mask :

```
cv::Mat masking(cv::Mat image,int mask[3][3])
{
        cv::Mat temImage = image.clone();
        for (int i = 1; i < image.rows - 1; i++)
        {
            for (int j = 1; j < image.cols - 1; j++)
            {

                    int pixel1 = image.at<uchar>(i - 1, j - 1) * mask[0][0];
                    int pixel2 = image.at<uchar>(i, j - 1) * mask[0][1];
                    int pixel3 = image.at<uchar>(i + 1, j - 1) * mask[0][2];

                    int pixel4 = image.at<uchar>(i - 1, j) * mask[1][0];
                    int pixel5 = image.at<uchar>(i, j) * mask[1][1];
                    int pixel6 = image.at<uchar>(i + 1, j) * mask[1][2];

                    int pixel7 = image.at<uchar>(i - 1, j + 1) * mask[2][0];
                    int pixel8 = image.at<uchar>(i, j + 1) * mask[2][1];
                    int pixel9 = image.at<uchar>(i + 1, j + 1) * mask[2][2];

                    int sum = pixel1 + pixel2 + pixel3 + pixel4 + pixel5 + pixel6 + pixel7 + pixel8 + pixel9;
                    if (sum < 0)
                        sum = 0;

                    if (sum > 255)
                        sum = 255;

                    temImage.at<uchar>(i, j) = sum;

            }
        }
        return temImage;
}
```

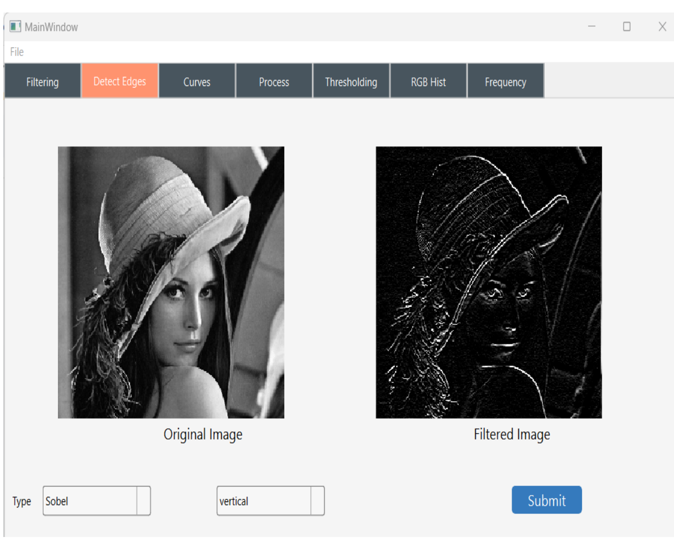
Canny Edge Detection Algorithm :

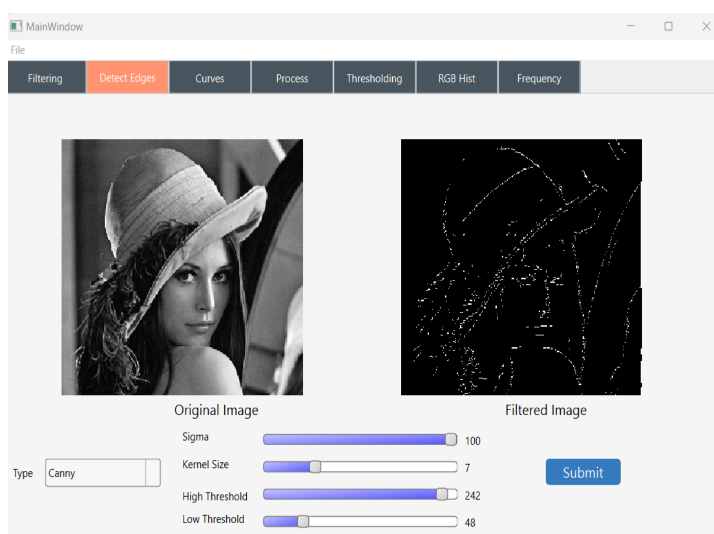
```
cv::Mat CannyEdgeDetection(cv::Mat image, int segma, int lowThreshold, int highThreshold, int KernalSize) {
    if(KernalSize%2==0) KernalSize=KernalSize+1;
    cv::Mat Blured, magnitude, direction, result;
    int(*maskH)[3];
    int(*maskV)[3];
    //Gussian Bluring
    GaussianBlur(image, Blured, cv::Size(KernalSize, KernalSize), segma, segma);
    maskH = getArray("Sobel", "horizontal");
    maskV = getArray("Sobel", "vertical");
    //Sobel Edge detection in both vertical and horizontal directions
    cv::Mat gradientx = masking(Blured, maskH);
    cv::Mat gradienty = masking(Blured, maskV);
    gradientx.convertTo(gradientx, CV_32F);
    gradienty.convertTo(gradienty, CV_32F);
    cartToPolar(gradientx, gradienty, magnitude, direction, true);
    non_max_suppression(magnitude, direction, result);
    inRange(result, cv::Scalar(lowThreshold), cv::Scalar(highThreshold), result);
    return result;
};
```

- Non max Suppression

```cpp
void non_max_suppression(cv::Mat& magnitude,cv::Mat& direction,cv::Mat& result) {
    // Create a copy of the magnitude matrix
    result = magnitude.clone();

    // Suppress non-maximum points
    for (int y = 1; y < magnitude.rows - 1; y++) {
        for (int x = 1; x < magnitude.cols - 1; x++) {
            // Calculate the angle of the gradient at this pixel
            float angle = direction.at<float>(y, x) * 180.0 / CV_PI;

            // Wrap the angle around 180 degrees
            if (angle < 0) {
                angle += 180;
            }

            // Find the two neighboring pixels along the gradient direction
            int x1, y1, x2, y2;
            if (angle < 22.5 || angle >= 157.5) {
                x1 = x2 = x;
                y1 = y - 1;
                y2 = y + 1;
            }
            else if (angle < 67.5) {
                x1 = x - 1;
                y1 = y - 1;
                x2 = x + 1;
                y2 = y + 1;
            }
            else if (angle < 112.5) {
                x1 = x - 1;
                y1 = y;
                x2 = x + 1;
                y2 = y;
            }
            else {
                x1 = x - 1;
                y1 = y + 1;
                x2 = x + 1;
                y2 = y - 1;
            }

            // Suppress the point if its magnitude is smaller than either of its neighbors
            float mag = magnitude.at<float>(y, x);
            float mag1 = magnitude.at<float>(y1, x1);
            float mag2 = magnitude.at<float>(y2, x2);
            if (mag < mag1 || mag < mag2) {
                result.at<float>(y, x) = 0;
            }
        }
```

Samples





Output Analysis

Picking the Edge detection method we want and adding parameters then observe output.

Tab 3 - Histogram

Page Contents

- Distribution Curve
- Histogram

Algorithm

Distribution Curve:

```
cv::Mat DistributionCal(cv::Mat histogram) {

        int num_bins = histogram.rows;
        Mat curve_image(400, 512, CV_8UC3, Scalar(0, 0, 0));
        Mat normalized_histogram;
        normalize(histogram, normalized_histogram, 0, 400, NORM_MINMAX, -1, Mat());

        vector<Point> curve_points(num_bins);
        for (int i = 0; i < num_bins; i++) {
            curve_points[i] = Point(2 * i, curve_image.rows - normalized_histogram.at<float>(i));
        }
        const Point *pts = (const Point*)Mat(curve_points).data;
        int npts = Mat(curve_points).rows;

        polylines(curve_image, &pts, &npts, 1, false, Scalar(255, 0, 0), 2);

        return curve_image;

};
```

Histogram:

```
Mat calc_histogram(Mat image) {
    Mat hist;
    hist = Mat::zeros(256, 1, CV_32F);
    image.convertTo(image, CV_32F);
    double value = 0;
    for (int i = 0; i < image.rows; i++)
    {
        for (int j = 0; j < image.cols; j++)
        {
            value = image.at<float>(i, j);
            hist.at<float>(value) = hist.at<float>(value) + 1;
        }
    }
    return hist;
}
```

Samples


MainWindow
File
Filtering
Detect Edges
Curves
Process
Thresholding
RGB Hist
Frequency
Distribution Curve
Histogram

Tab 4 - Processing

Page Contents

- Image Normalization
- Histogram Equalization

Algorithms

Image Normalization:

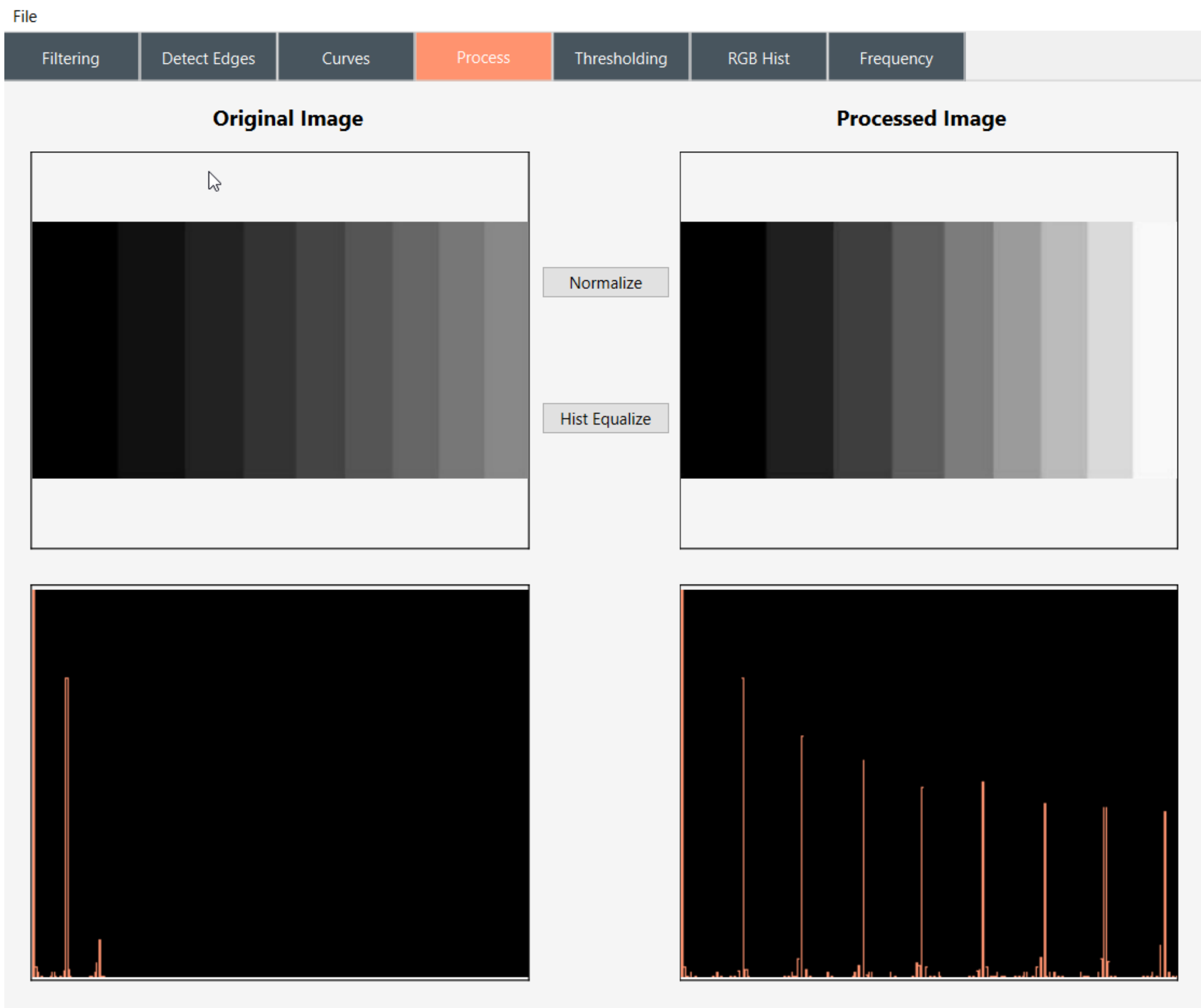
```
void ProcessImg::normalize(Mat& img, int minVal, int maxVal){
    int oldMin = 99999, oldMax = 0;
    for (int rowCounter = 0; rowCounter < img.rows; rowCounter++){
        for (int colCounter = 0; colCounter < img.cols; colCounter++){
            const unsigned char& pixelVal = img.at<unsigned char>(rowCounter,colCounter);
            if(pixelVal>oldMax) oldMax = pixelVal;
            if(pixelVal<oldMin) oldMin = pixelVal;
        }
    }
    if(minVal==oldMin && maxVal==oldMax){return ;}
    for (int rowCounter = 0; rowCounter < img.rows; rowCounter++){
            for (int colCounter = 0; colCounter < img.cols; colCounter++){
                unsigned char& pixelVal = img.at<unsigned char>(rowCounter,colCounter);
                pixelVal = ((pixelVal-oldMin) * (maxVal-minVal) / (oldMax-oldMin)) + minVal;
            }
        }
}
```

Histogram Equalization:

```
void ProcessImg::histEqualize(Mat& img){
    int n_bins = 256;
    QPair<vector<int>, int> data = calcHist(img); // getting histogram array and the minima in pair
    vector<int> hist = data.hist;
    int minVal = data.minValue;
    float scale = (n_bins - 1.f) / (img.rows*img.cols - hist[minVal]);
    vector<int> lut(n_bins, 0);
    int i = minVal+1;
    int sum = 0;
    for (; i < hist.size(); ++i) {          ⚠ Comparison of integers of different signs: 'int' and
        sum += hist[i];
        // the value is saturated in range [0, max_val]
        lut[i] = max(0, min(int(round(sum * scale)), 255));
    }

    for (int rowCounter = 0; rowCounter < img.rows; rowCounter++){
        for (int colCounter = 0; colCounter < img.cols; colCounter++){
            unsigned char& pixelVal = img.at<unsigned char>(rowCounter,colCounter);
            pixelVal = lut[pixelVal];
        }
    }

}
```

Samples

Image Normalization:

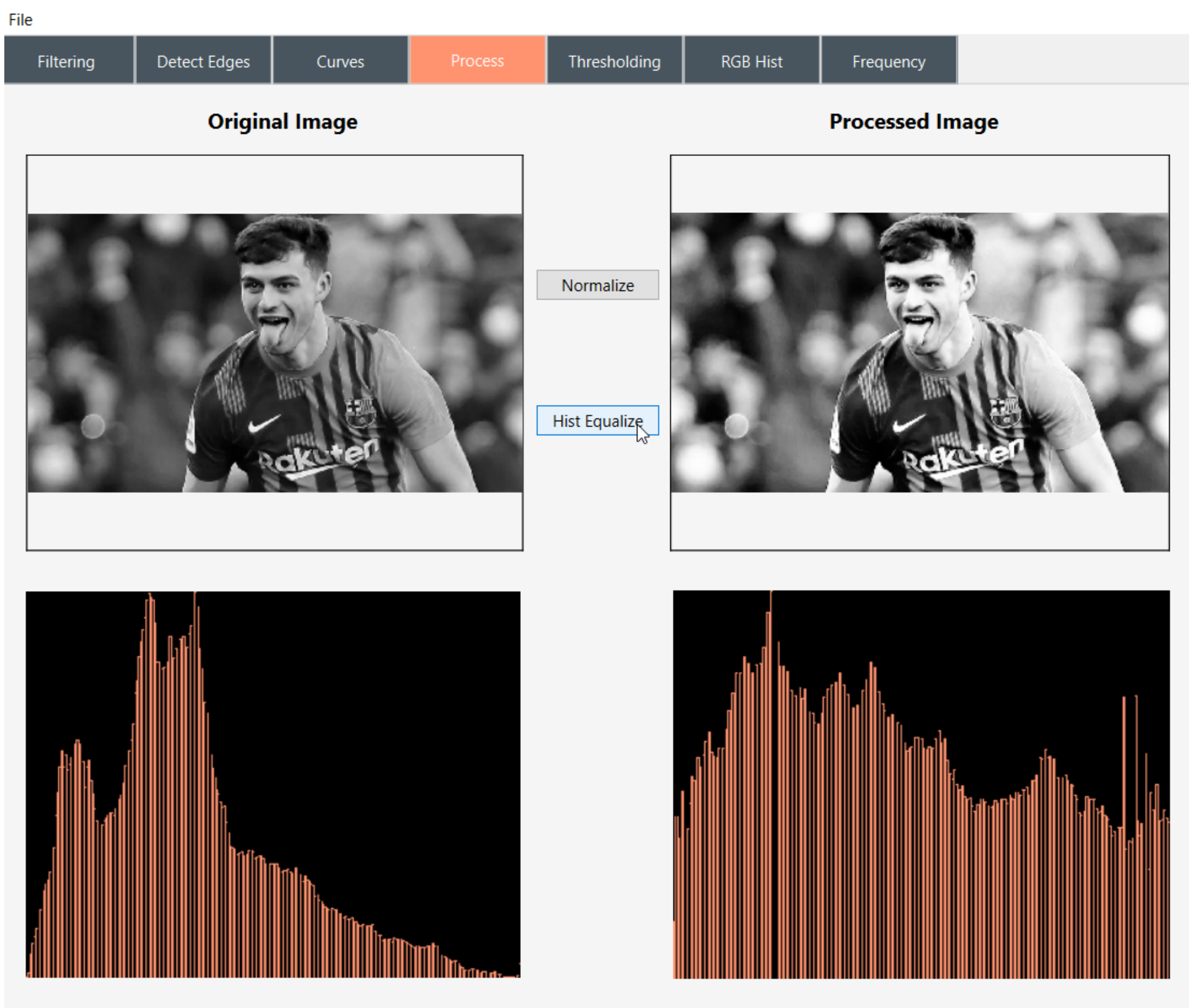


Output Analysis

Normalization effect is to scale the intensity values of the image so that they fall within a certain range (contrast stretching).

Samples

Histogram Equalization:



Output Analysis

Histogram equalization improves the contrast of an image by redistributing its pixel intensities. This can help to improve the visual appearance of an image or make it more suitable for further processing.

Tab 5 - (Threshold)

Page Contents

- Global Threshold
- Local Threshold

Algorithms

Global Thresholding:

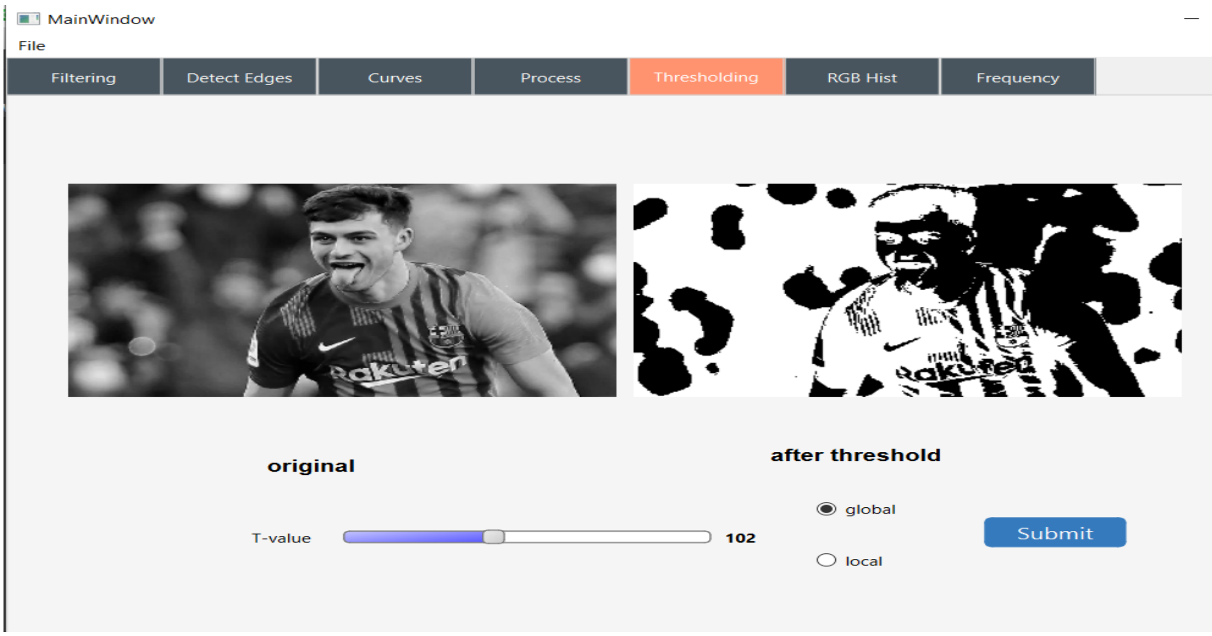
```
}
void Threshold(Mat& input_image, Mat& output_image,int threshold ) {
    output_image.create(input_image.rows, input_image.cols, CV_8UC1);
    for (int i = 0; i < input_image.rows; i++) {
        for (int j = 0; j < input_image.cols; j++) {
            if (input_image.at<uchar>(i, j) > threshold) {
                output_image.at<uchar>(i, j) = 0;
            } else {
                output_image.at<uchar>(i, j) = 255;
            }
        }
    }
}
```

Local Thresholding:

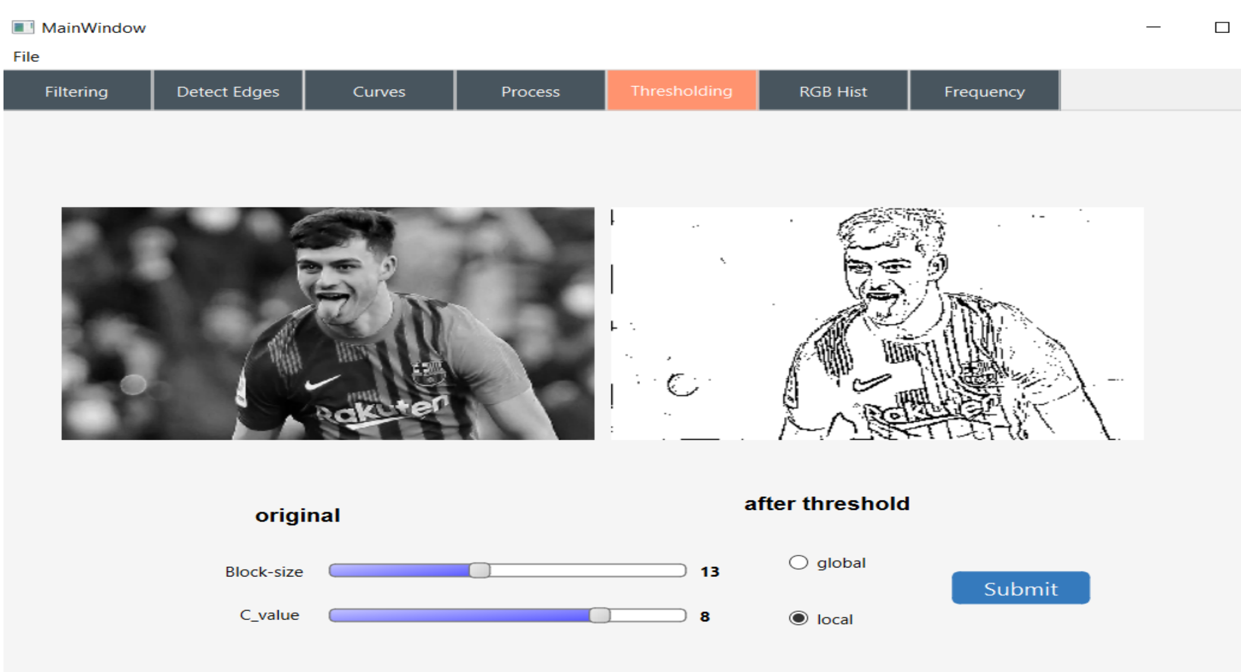
```
void localThreshold( cv::Mat& input, cv::Mat& output, int blockSize, double C) {
    if(blockSize%2==0){
        blockSize=blockSize+1;
    }

    vector<int> pixelValues(blockSize * blockSize);
    output.create(input.rows, input.cols, CV_8UC1);

    for (int y = 0; y < input.rows; y++)
        {
            for (int x = 0; x < input.cols; x++)
            {
                // Get the neighborhood around the pixel
                int index = 0;
                for (int j = -blockSize / 2; j <= blockSize / 2; j++)
                {
                    for (int i = -blockSize / 2; i <= blockSize / 2; i++)
                    {
                        // Check if the pixel is within the image boundaries
                        int px = x + i;
                        int py = y + j;
                        if (px >= 0 && px < input.cols && py >= 0 && py < input.rows)
                        {
                            pixelValues[index++] = input.at<uchar>(py, px);
                        }
                    }
                }

                // Compute the local threshold using the mean and constant C
                int threshold = (int)round(cv::mean(pixelValues)[0] - C);

                // Apply the threshold to the pixel
                if (input.at<uchar>(y, x) >= threshold)
                {
                    output.at<uchar>(y, x) = 255;
                }
                else
                {
                    output.at<uchar>(y, x) = 0;
                }
            }
        }
}
```

Samples

Global Threshold:



Local Threshold:



Output Analysis

Global thresholding can be used to segment an image into foreground and background regions, which is a fundamental task in image processing and computer vision. This can be useful for a wide range of applications such as object detection, recognition, and tracking. In our application we choose Threshold value with a slider and apply it directly to the image.

The output of local thresholding is a binary image that separates an input image into foreground and background regions based on local intensity characteristics. It is a powerful technique that can be used for a wide range of image processing applications, including object detection, recognition, and tracking.

A constant value C from the local mean helps to adjust the threshold level to be more adaptive to the local image characteristics, which can improve the accuracy of foreground/background segmentation in local thresholding.

Tab 6 - (RGB hist)

Page Contents

- RGB Histogram
- RGB Distribution Function
- RGB Cumulative Distribution Function

Algorithms

RGB Histogram

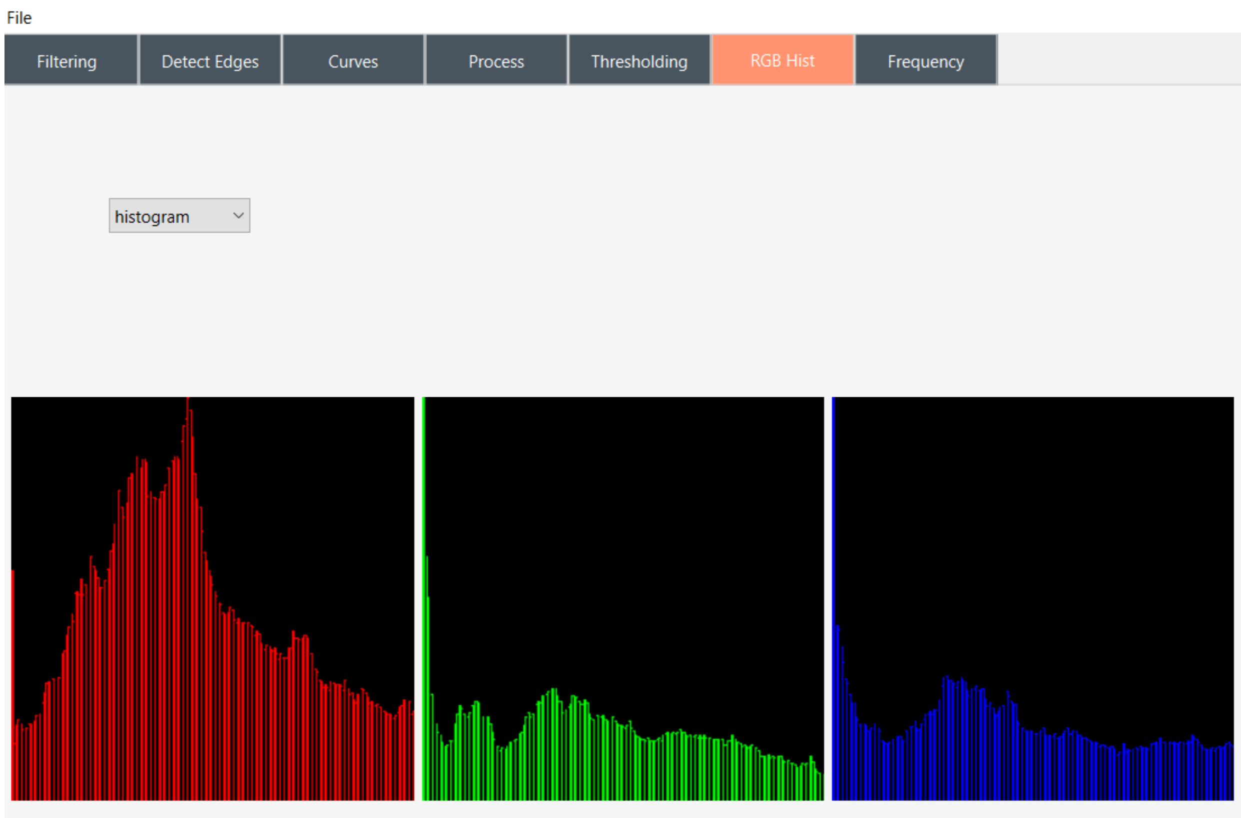
```
};
std::tuple<cv::Mat, cv::Mat,cv::Mat> splitChannels(cv::Mat image){
    std::vector<Mat> channels;
    Mat red,green,blue;
    split(image, channels);
    red=channels[2];
    green=channels[1];
    blue=channels[0];
    red=calc_histogram(red);
    green=calc_histogram(green);
    blue=calc_histogram(blue);
    red=plot_histogram(red,255);
    green=plot_histogram(green,0,255);
    blue=plot_histogram(blue,0,0,255);
    return std::make_tuple(red,green,blue);


}
```

Df and CDF Plotting:

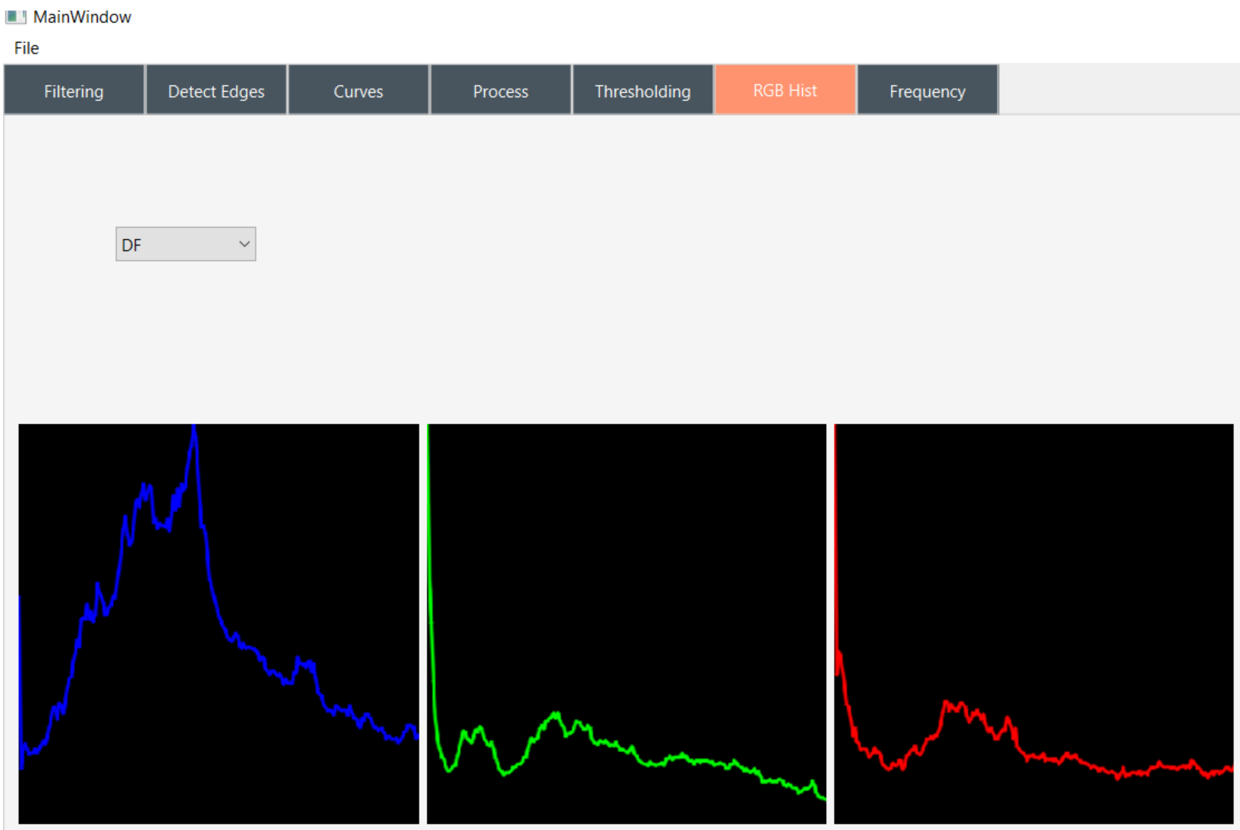
```cpp
std::tuple<cv::Mat, cv::Mat,cv::Mat> plot_rgb_distribution_function(Mat image,string mode) {
    vector<Mat> bgr_planes;
    split(image, bgr_planes);

    const int num_bins = 256;
    const int hist_height = 400;
    const int hist_width = 512;
    const int bin_width = cvRound(static_cast<double>(hist_width) / num_bins);

    // Create histograms for each color channel
    Mat b_hist, g_hist, r_hist;
    calcHist(&bgr_planes[0], 1, 0, Mat(), b_hist, 1, &num_bins, 0);
    calcHist(&bgr_planes[1], 1, 0, Mat(), g_hist, 1, &num_bins, 0);
    calcHist(&bgr_planes[2], 1, 0, Mat(), r_hist, 1, &num_bins, 0);

    // Create separate images for each histogram
    Mat b_DF(hist_height, hist_width, CV_8UC3, Scalar(0, 0, 0));
    Mat g_DF(hist_height, hist_width, CV_8UC3, Scalar(0, 0, 0));
    Mat r_DF(hist_height, hist_width, CV_8UC3, Scalar(0, 0, 0));

    if(mode=="cumulative"){
        Mat cumulative_r,cumulative_b,cumulative_g;
        r_hist.copyTo(cumulative_r);
        g_hist.copyTo(cumulative_b);
        b_hist.copyTo(cumulative_g);


    for(int j=1;j<num_bins;j++){


        cumulative_r.at<float >(j)+=cumulative_r.at<float>(j-1);
        cumulative_b.at<float >(j)+=cumulative_b.at<float>(j-1);
        cumulative_g.at<float >(j)+=cumulative_g.at<float>(j-1);
    }
    normalize(cumulative_b, b_hist, 0, hist_height, NORM_MINMAX, -1, Mat());
    normalize(cumulative_g, g_hist, 0, hist_height, NORM_MINMAX, -1, Mat());
    normalize(cumulative_r, r_hist, 0, hist_height, NORM_MINMAX, -1, Mat());
    }
    else{

        // Normalize the histograms
        normalize(b_hist, b_hist, 0, hist_height, NORM_MINMAX, -1, Mat());
        normalize(g_hist, g_hist, 0, hist_height, NORM_MINMAX, -1, Mat());
        normalize(r_hist, r_hist, 0, hist_height, NORM_MINMAX, -1, Mat());

    }


    // Plot the histograms
    for (int i = 1; i < num_bins; i++) {
        line(b_DF, Point(bin_width * (i - 1), hist_height - cvRound(b_hist.at<float>(i - 1))),
            Point(bin_width * i, hist_height - cvRound(b_hist.at<float>(i))), Scalar(255, 0, 0), 2, LINE_AA);
        line(g_DF, Point(bin_width * (i - 1), hist_height - cvRound(g_hist.at<float>(i - 1))),
            Point(bin_width * i, hist_height - cvRound(g_hist.at<float>(i))), Scalar(0, 255, 0), 2, LINE_AA);
        line(r_DF, Point(bin_width * (i - 1), hist_height - cvRound(r_hist.at<float>(i - 1))),
            Point(bin_width * i, hist_height - cvRound(r_hist.at<float>(i))), Scalar(0, 0, 255), 2, LINE_AA);

    }

return std::make_tuple(r_DF,g_DF,b_DF);
}
```

Samples

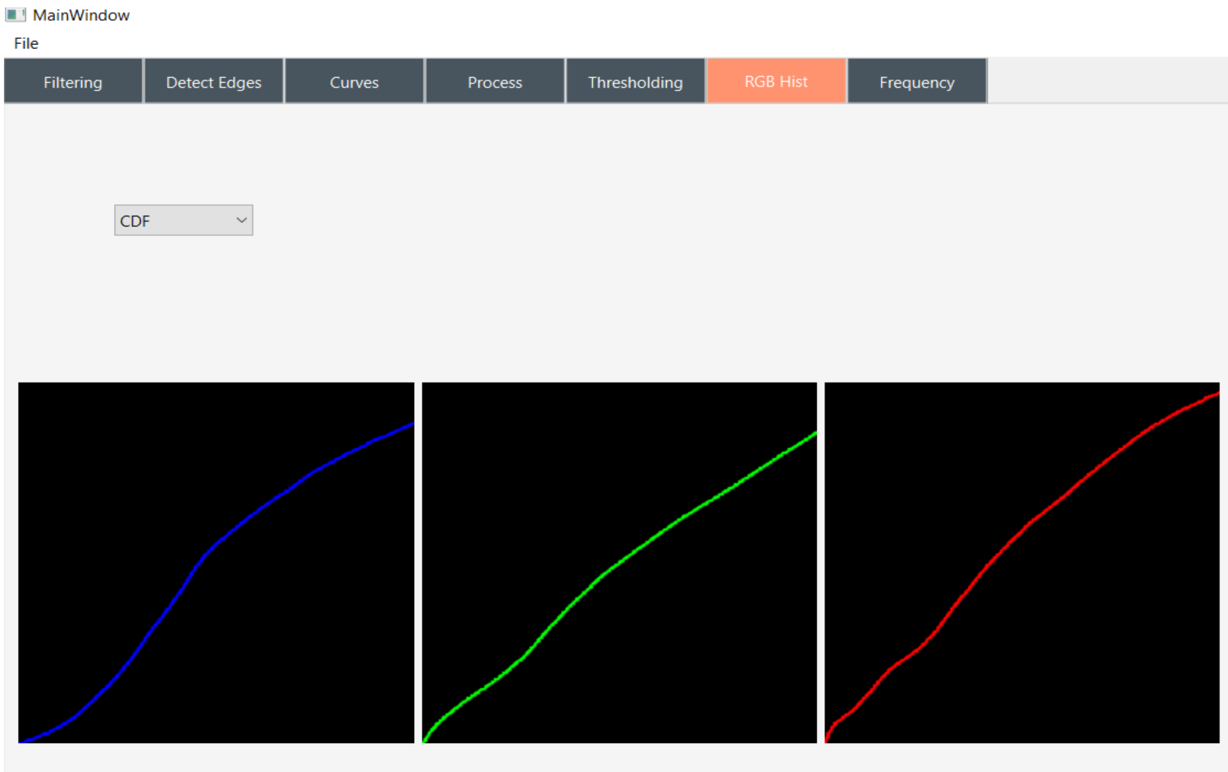
RGB Histogram:



RGB DF:


MainWindow
File
Filtering
Detect Edges
Curves
Process
Thresholding
RGB Hist
Frequency
DF

RGB CDF:


MainWindow
File
Filtering
Detect Edges
Curves
Process
Thresholding
RGB Hist
Frequency
CDF

Tab 7 - Frequency Filters

Page Contents

- Low-Pass Filter
- High-Pass Filter
- Hybrid Image

Algorithms

Calculate Fourier Transform for an Image:

```
void fourierShift(Mat& Img) {
    Img = Img(Rect(0, 0, Img.cols & -2, Img.rows & -2));
    int cx = Img.cols / 2;
    int cy = Img.rows / 2;
    // Split image into quadrants
    Mat q0(Img, Rect(0, 0, cx, cy));
    Mat q1(Img, Rect(cx, 0, cx, cy));
    Mat q2(Img, Rect(0, cy, cx, cy));
    Mat q3(Img, Rect(cx, cy, cx, cy));
    // Swap quadrants
    Mat tmp;
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);
    // Swap other quadrants
    q1.copyTo(tmp);
    q2.copyTo(q1);
    tmp.copyTo(q2);
}

// Convert the image to complex numbers by applying dft
Mat calcDFT(Mat& img) {
    // Mat padded = adjustSize(img);
    // copy the source image, on the border add zero values
    Mat planes[] = { Mat_<float>(img), Mat::zeros(img.size(), CV_32F) };
    Mat complex_img;
    merge(planes, 2, complex_img);
    // create a complex matrix (Fourier transform)
    dft(complex_img, complex_img);
    fourierShift(complex_img);

    return complex_img;
}
```

Create Filter:

```
Mat createFilter(Mat& complex_img, float distance, string filterType) {
    Mat filter(complex_img.size(), CV_32F, Scalar(1));
    Point center = Point(complex_img.rows / 2, complex_img.cols / 2);
    float radius;
    for (int i = 0; i < complex_img.rows; i++) {
        for (int j = 0; j < complex_img.cols; j++) {
            radius = sqrt(pow((i - center.x), 2.0) + pow((j - center.y), 2.0));

            if (filterType == "lowpass"){
                if (radius > distance)
                    filter.at<float>(i, j) = 0;
            }
            else if (filterType == "highpass") {
                if (radius < distance)
                    filter.at<float>(i, j) = 0;
            }
        }
    }

    return filter;
}
```

Apply Filter and Inverse Fourier:

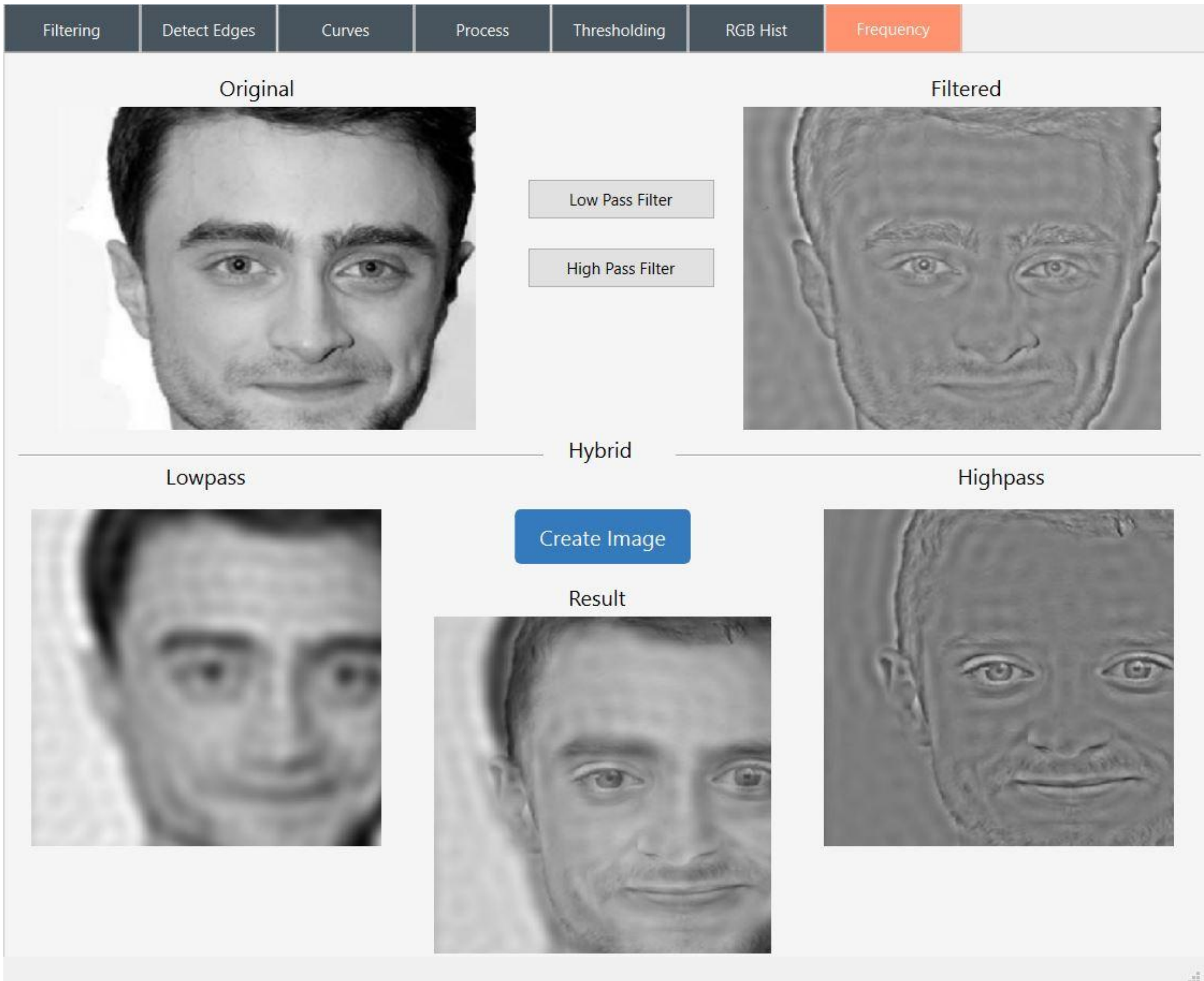
```
Mat applyFilter(Mat& complex_img, Mat& filter) {
    Mat output_img;
    Mat planes_filter[] = { Mat_<float>(filter.clone()), Mat_<float>(filter.clone()) };

    Mat planes_dft[] = { Mat_<float>(complex_img), Mat::zeros(complex_img.size(), CV_32F) };
    split(complex_img, planes_dft);

    Mat planes_out[] = { Mat::zeros(complex_img.size(), CV_32F), Mat::zeros(complex_img.size(), CV_32F) };
    planes_out[0] = planes_filter[0].mul(planes_dft[0]);
    planes_out[1] = planes_filter[1].mul(planes_dft[1]);

    merge(planes_out, 2, output_img);

    return output_img;
}
```

```
void ifft(Mat& complex_img) {
    fourierShift(complex_img);
    dft(complex_img, complex_img, DFT_INVERSE | DFT_REAL_OUTPUT);
    normalize(complex_img, complex_img, 0, 1, NORM_MINMAX);
}
```

Hybrid Image:

```
void MainWindow::on_submitThreshold_2_clicked()
{
    // first image
    cvtColor(img->getOriginalImage(), img->getImage("hyprid"), COLOR_BGR2GRAY);
    Mat imglow = img->getImage("hyprid");
    cv::resize(imglow, imglow, Size(512, 512), 0, 0);
    Mat complex_img1 = calcDFT(imglow);
    Mat filter1 = createFilter(complex_img1, 20, "lowpass");
    Mat output1 = applyFilter(complex_img1, filter1);

    // second image
    cvtColor(img2->getOriginalImage(), img2->getImage("hyprid"), COLOR_BGR2GRAY);
    Mat imghigh = img2->getImage("hyprid");
    cv::resize(imghigh, imghigh, Size(512, 512), 0, 0);
    Mat complex_img2 = calcDFT(imghigh);
    Mat filter2 = createFilter(complex_img2, 20, "highpass");
    Mat output2 = applyFilter(complex_img2, filter2);

    // create hybrid image
    Mat hybrid;
    add(output1, output2, hybrid);
    ifft(hybrid);
    ifft(output1);
    ifft(output2);

    // prepare to view
    double  minVal,  maxVal;
    minMaxLoc(output1,  &minVal,  &maxVal);
    output1.convertTo(output1,  CV_8U,  255.0/(maxVal  -  minVal),  -minVal);
    minMaxLoc(output2,  &minVal,  &maxVal);
    output2.convertTo(output2,  CV_8U,  255.0/(maxVal  -  minVal),  -minVal);
    minMaxLoc(hybrid,  &minVal,  &maxVal);
    hybrid.convertTo(hybrid,  CV_8U,  255.0/(maxVal  -  minVal),  -minVal);

    // view all images
    showImg(output1, ui->originalImg_freqfilters2, QImage::Format_Grayscale8, this->origWidth, this->origHeight);
    showImg(output2, ui->originalImg_freqfilters3, QImage::Format_Grayscale8, this->origWidth, this->origHeight);
    showImg(hybrid, ui->resulthybrid, QImage::Format_Grayscale8, this->origWidth, this->origHeight);
}
```

Samples



Output Analysis

Low pass filter removes the high frequency parts from the image making it smoother (more blurry), while high pass filter removes the low frequency parts detecting the edges of the image.

Hybrid image is a composition of 2 images after applying low pass filter on one and highpass filter on the other. Making the lowpass filtered image be more obvious when you look at it from a long distance, and the high pass filtered image be more obvious when you look at it from a close distance.