Homework requirement:

1. Output Gaussian filter result image
2. Output Canny edge detector result image
3. Output Hough transform result image

Development environment:

Windows 10
Visual Studio 2022 community
OpenCV 4.6.0

Implementation:

Main function:

```
int main() {
    //1
    Mat src = imread("1.jpg");
    Mat srcGray = imread("1.jpg", IMREAD_GRAYSCALE);
    Mat blur = GaussianBlur(srcGray, 5, 10);
    imwrite("1_result_1.jpg", blur);
    Mat canny = Canny(srcGray, 5, 10, 140, 170);
    imwrite("1_result_2.jpg", canny);
    Mat hough = HoughTransform(canny, 20);
    Mat result=Mat::zeros(src.rows,src.cols,CV_8UC3);
    result = DrawImage(src, hough);
    imwrite("1_result_3.jpg", result);
    /*imshow("src", src);
    imshow("blur", blur);
    imshow("canny", canny);
    imshow("hough", result);*/
```

Gaussian blur:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The Gaussian kernel is implement with the above equation.
The kernel is normalized to keep the image intensity.

```
Mat GaussianBlur(Mat src, int size, double sigma) {
    //gaussian function: 1/(2*pi*sigma^2) * e^(-(x^2+y^2)/2*sigma^2)
    //0.707^2 = 0.5
    Mat kernel = Mat::zeros(size, size, CV_64F);
    double s = 2 * sigma * sigma;
    int half_size = size / 2;
    double sum = 0;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            //x^2+y^2=z^2
            double z = sqrt((i - half_size) * (i - half_size) + (j - half_size) * (j - half_size));
            kernel.at<double>(i, j) = (exp(-z / s))/(s*M_PI);
            sum += kernel.at<double>(i, j);
        }
    }
    //normalization
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            kernel.at<double>(i, j) = kernel.at<double>(i, j)/sum;
        }
    }
```

Then we calculate the convolution of padded image and the kernel.

```
//Mat padded_img = Padding(src, half_size, half_size, half_size, half_size);
Mat padded_img=Padding(src, half_size, half_size, half_size, half_size);
Mat result = Mat::zeros(src.rows, src.cols, CV_8UC1);
for (int i = half_size; i < padded_img.rows - half_size; i++) {
    for (int j = half_size; j < padded_img.cols- half_size; j++) {
        double value = 0;
        for (int x = 0; x < size; x++) {
            for (int y = 0; y < size; y++) {
                value += padded_img.at<uchar>(i-half_size+x, j-half_size+y) * kernel.at<double>(x, y);
            }
        }
        result.at<uchar>(i - half_size, j - half_size) = (uchar)value;
    }
}
return result;
```

Implementation of BORDER_REPLICATE padding:

```cpp
Mat Padding(Mat src, int top, int left, int down, int right) {//zero padding
    Mat result = Mat::zeros(src.rows + top + down, src.cols + left + right, CV_8UC1);
    //top left
    for (int i = 0; i < top; i++) {
        for (int j = 0; j < left; j++) {
            result.at<uchar>(i, j) = src.at<uchar>(0, 0);
        }
    }
    //top
    for (int i = 0; i < top; i++) {
        for (int j = left; j < result.cols-right; j++) {
            result.at<uchar>(i, j) = src.at<uchar>(0, j - left);
        }
    }
    //top right
    for (int i = 0; i < top; i++) {
        for (int j = result.cols-right; j < result.cols; j++) {
            result.at<uchar>(i, j) = src.at<uchar>(0, src.cols-1);
        }
    }
    //left
    for (int i = top; i < result.rows-down; i++) {
        for (int j = 0; j < left; j++) {
```

```cpp
//left
for (int i = top; i < result.rows-down; i++) {
    for (int j = 0; j < left; j++) {
        result.at<uchar>(i, j) = src.at<uchar>(i-top, 0);
    }
}
//right
for (int i = top; i < result.rows - down; i++) {
    for (int j = result.cols-right; j < result.cols; j++) {
        result.at<uchar>(i, j) = src.at<uchar>(i-top, src.cols-1);
    }
}
//down left
for (int i = result.rows-down; i < result.rows; i++) {
    for (int j = 0; j < left; j++) {
        result.at<uchar>(i, j) = src.at<uchar>(src.rows-1, 0);
    }
}
//down
for (int i = result.rows - down; i < result.rows; i++) {
    for (int j = left; j < result.cols-right; j++) {
        result.at<uchar>(i, j) = src.at<uchar>(src.rows - 1, j-left);
```

```
//down
for (int i = result.rows - down; i < result.rows; i++) {
    for (int j = left; j < result.cols-right; j++) {
        result.at<uchar>(i, j) = src.at<uchar>(src.rows - 1, j-left);
    }
}
//down right
for (int i = result.rows - down; i < result.rows; i++) {
    for (int j = result.cols-right; j < result.cols; j++) {
        result.at<uchar>(i, j) = src.at<uchar>(src.rows - 1, src.cols - 1);
    }
}

//main area
for (int i = top; i < result.rows-down; i++) {
    for (int j = left; j < result.cols-right; j++) {
        result.at<uchar>(i, j) = src.at<uchar>(i-top, j-left);
    }
}
return result;
```

Canny edge detector:

```
Mat Canny(Mat src, int size, double sigma, int low, int high) {
    Mat result = GaussianBlur(src, size, sigma);
    Mat multichannel = GradientCalculation(result);
    result = NonMaximumSuppression(multichannel);
    result = Linking(result,low,high);
    return result;
}
```

Canny edge detector has 5 steps: noise reduction, gradient calculation, non-maximum suppression, double thresholding, edge tracking by hysteresis. For this implementation we use Gaussian blur as noise reduction method and we combine double thresholding, edge linking into one function Linking().

For gradient calculation, the output is a CV_64FC2 Mat. The first channel stores magnitude and the second one stores the direction vector.

Implementation of gradient calculation:

```cpp
Mat GradientCalculation(Mat src) {
    int sobel_x[3][3] = {{-1,0,1},{-2,0,2},{-1,0,1} };
    int sobel_y[3][3] = { {-1,-2,-1},{0,0,0},{1,2,1} };
    Mat result = Mat::zeros(src.rows, src.cols, CV_64FC2);
    Mat padded_img = Padding(src, 1, 1, 1, 1);
    for (int i = 1; i < padded_img.rows - 1; i++) {
        for (int j = 1; j < padded_img.cols-1; j++) {
            double value_x = 0;
            double value_y = 0;
            for (int x = 0; x < 3; x++) {
                for (int y = 0; y < 3; y++) {
                    value_x += padded_img.at<uchar>(i - 1 + x, j - 1 + y) * sobel_x[x][y];
                    value_y += padded_img.at<uchar>(i - 1 + x, j - 1 + y) * sobel_y[x][y];
                }
            }
            //result.at<Vec2d>(i - 1, j - 1).val[0] = abs(value_x) + abs(value_y);
            result.at<Vec2d>(i - 1, j - 1).val[0] = sqrt(value_x*value_x + value_y*value_y);
            if ((value_x - 0) < 0.01) {
                result.at<Vec2d>(i - 1, j - 1).val[1] = M_PI/2;
            }
            else {
                result.at<Vec2d>(i - 1, j - 1).val[1] = atan(value_y / value_x);//in radiance -pi/2 to pi/2
            }
        }
    }
    return result;//64FC2
}
```

For non-maximum suppression, we first divide direction into 4 cases: 0 degree, 45 degree, 90 degree, 135 degree. Then, we compare current pixel with its two neighbor. If the current pixel is the max value of them, the current value is stored to result. Otherwise, the result pixel is set to 0.

Implementation of non-maximum suppression:

```cpp
Mat NonMaximumSuppression(Mat src) {//CV_64FC2
    Mat channels[2];
    split(src, channels);
    Mat padded_img = Mat::zeros(src.rows + 2, src.cols + 2, CV_8UC1);
    for (int i = 1; i < padded_img.rows -1; i++) {
        for (int j = 1; j < padded_img.cols -1; j++) {
            padded_img.at<uchar>(i, j) = (uchar)channels[0].at<double>(i-1, j-1);
        }
    }
    //imshow("mag", padded_img);
    Mat result = Mat::zeros(channels[0].rows, channels[0].cols, CV_8UC1);
```

```
Mat result = Mat::zeros(channels[0].rows, channels[0].cols, CV_8UC1);
//theta: -pi/2 to pi/2 (rad)
//0: -22.5 to 22.5 (degree)
//45: 22.5 to 67.5 (degree)
// //135: -67.5 to -22.5 (degree)
//90: 67.5 to 90 & -90 to -67.5(degree)
for (int i = 1; i < padded_img.rows - 1; i++) {
    for (int j = 1; j < padded_img.cols - 1; j++) {
        if (channels[1].at<double>(i-1, j-1)<M_PI / 8 && channels[1].at<double>(i-1, j-1) > -M_PI / 8) {//0 degree
            if (padded_img.at<uchar>(i, j) >= padded_img.at<uchar>(i, j - 1) && padded_img.at<uchar>(i, j) >= padded_img.at<uchar>(i, j + 1))
                result.at<uchar>(i - 1, j - 1) = padded_img.at<uchar>(i, j);
        }
        else if (channels[1].at<double>(i-1, j - 1)<3 * M_PI / 8 && channels[1].at<double>(i - 1, j - 1) > M_PI / 8) {//45 degree
            if (padded_img.at<uchar>(i, j) >= padded_img.at<uchar>(i+1, j - 1) && padded_img.at<uchar>(i, j) >= padded_img.at<uchar>(i-1, j + 1))
                result.at<uchar>(i - 1, j - 1) = padded_img.at<uchar>(i, j);
        }
        else if (channels[1].at<double>(i - 1, j - 1)< -M_PI / 8 && channels[1].at<double>(i - 1, j - 1) > -3*M_PI / 8) {//135
            if (padded_img.at<uchar>(i, j) >= padded_img.at<uchar>(i-1, j - 1) && padded_img.at<uchar>(i, j) >= padded_img.at<uchar>(i+1, j + 1))
                result.at<uchar>(i - 1, j - 1) = padded_img.at<uchar>(i, j);
        }
        else {
            if (padded_img.at<uchar>(i, j) >= padded_img.at<uchar>(i-1, j) && padded_img.at<uchar>(i, j) >= padded_img.at<uchar>(i+1, j))
                result.at<uchar>(i - 1, j - 1) = padded_img.at<uchar>(i, j);
        }
    }
}
//imshow("non-max", result);
return result;
```

For edge linking, we have low & high thresholds. If the pixel value is higher than the high threshold, we set result pixel to 255 and push it in a list. We take the front element from the list and check if there is a weak point neighbor. When we find a weak point, we set it 255 on padded and result and push it into the list. Repeat the process until the list is empty.

Implementation of double thresholding & edge linking:

```
Mat Linking(Mat src, int low, int high) {
    //double threshold high & low
    //linking condition:
    //1. value higher than thresh_high
    //2. value between thresh_high and low, and there is a high neighbor
    Mat padded = Padding(src, 1, 1, 1, 1);
    Mat result = Mat::zeros(src.rows, src.cols, CV_8UC1);
    std::list<std::pair<int,int>> strong_points;
    for (int i = 1; i < padded.rows-1; i++) {
        for (int j = 1; j < padded.cols-1; j++) {
            if (padded.at<uchar>(i, j) > high) {//strong
                result.at<uchar>(i - 1, j - 1) = 255;
                std::pair<int, int> coor(i,j);
                strong_points.push_back(coor);
            }
        }
    }
```

```
    while (!strong_points.empty()) {//find strong point's weak neighbor
        int x = strong_points.front().first;
        int y = strong_points.front().second;
        for (int i = -1; i < 2; i++) {
            for (int j = -1; j < 2; j++) {
                if (x + i > 0 && x + i < padded.rows - 1 && y + j>0 && y + j < padded.cols - 1) {//ignore border
                    if (padded.at<uchar>(x + i, y + j) > low && padded.at<uchar>(x + i, y + j) <= high) {//find weak point
                        padded.at<uchar>(x + i, y + j) = 255;
                        result.at<uchar>(x + i-1, y + j-1) = 255;
                        std::pair<int, int> coor(x + i, y + j);
                        strong_points.push_back(coor);
                    }
                }
            }
        }
        strong_points.pop_front();
    }
    return result;
}
```

Hough transform:

We convert Cartesian coordinates to polar coordinates, and we form a 2d container which uses rho and theta as indices. Each pixel votes to the corresponding grid in the container. If the voting value exceeds threshold, we find two point on the Cartesian coordinates space which is on the line corresponding to the point on polar coordinates space. We draw the line on the result image.

Implementation of Hough transform:

```
HoughTransform(Mat src, int thresh_voting) {
//rho = x cos(theta) + y sin(theta)
const double rho_max = sqrt(src.rows * src.rows + src.cols * src.cols);
//const int thresh_voting = 40;
const int theta_size = 180;
std::vector<std::vector<int>> voting((int)(2 * rho_max), std::vector<int>(theta_size, 0));//2d vector

Mat hough = Mat::zeros(src.rows, src.cols, CV_8UC1);
Mat result = Mat::zeros(src.rows, src.cols, CV_8UC3);

for (int i = 0; i < src.rows; i++) {
    for (int j = 0; j < src.cols; j++) {
        if (src.at<uchar>(i, j) == 255) {
            //Cart point -> Polar line(set of points), marks on hough space
            //theta from 0 to 2pi, step: 1 degree
            for (int k = 0; k < 180; k++) {
                int rho = (int)(j * cos(M_PI * k / 180) + i * sin(M_PI * k / 180) + rho_max);//rho always > 0
                voting[rho][k]++;
            }
        }
    }
}
```

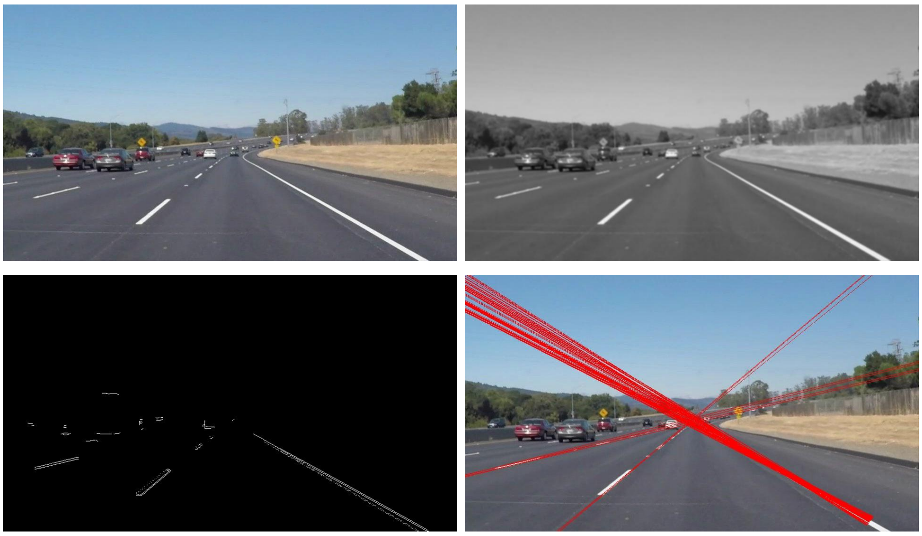
```
//select votings above thresh
for (int i = 0; i < (int)(2 * rho_max); i++) {
    for (int j = 0; j < theta_size; j++) {
        if (voting[i][j] > thresh_voting) {
            //form a line for rho, theta pair
            //select 2 points from line
            Point p1(cos(j * M_PI / 180) * (i-rho_max)+1000 * (-sin(j * M_PI / 180)), sin(j * M_PI / 180) * (i - rho_max) + 1000 * (cos(j * M_PI / 180)));
            Point p2(cos(j * M_PI / 180) * (i - rho_max) - 1000 * (-sin(j * M_PI / 180)), sin(j * M_PI / 180) * (i - rho_max) - 1000 * (cos(j * M_PI / 180)));
            //std::cout << p1 << p2 << "\n";
            //draw a line on result
            line(result, p1, p2, Scalar(0, 0, 255), 2, LINE_8);
        }
    }
}
return result;
```

Implementation of DrawImage():

```
DrawImage(Mat src1, Mat src2) {
Mat result = src1.clone();
for (int i = 0; i < src1.rows; i++) {
    for (int j = 0; j < src1.cols; j++) {
        if (src2.at<Vec3b>(i, j).val[2] > 0) {
            result.at<Vec3b>(i, j).val[0] = src2.at<Vec3b>(i, j).val[0];
            result.at<Vec3b>(i, j).val[1] = src2.at<Vec3b>(i, j).val[1];
            result.at<Vec3b>(i, j).val[2] = src2.at<Vec3b>(i, j).val[2];
        }
    }
}
return result;
```

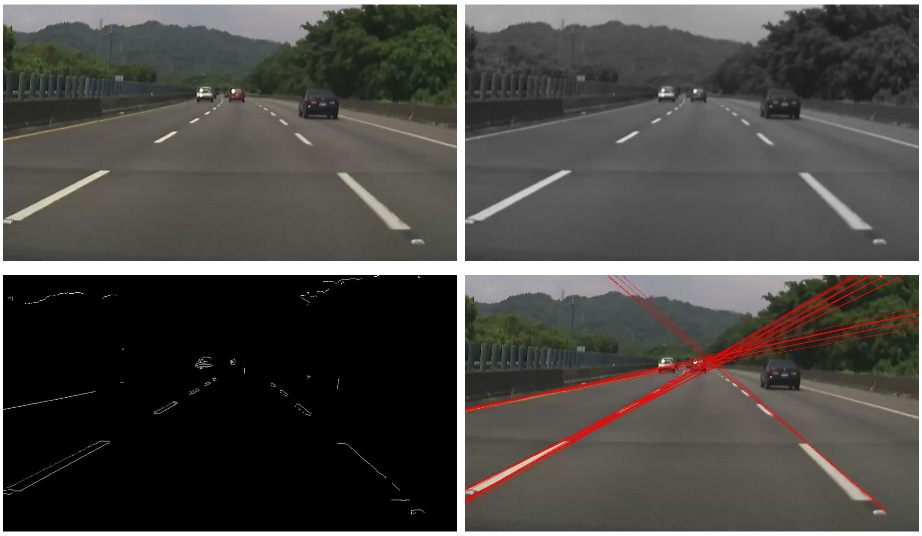
Result:

1.jpg
Parameters:
Gaussian(size=5,sigma=1),
Canny(size=5,sigma=1,low=150,high=200),
Hough(threshold=50)



2.jpg
Parameters:
Gaussian(size=5,sigma=1),
Canny(size=5,sigma=1,low=120,high=150),
Hough(threshold=50)

3.jpg
Parameters:
Gaussian(size=5,sigma=1),
Canny(size=5,sigma=1,low=100,high=150),
Hough(threshold=50)



4.jpg
Parameters:
Gaussian(size=5,sigma=1),
Canny(size=5,sigma=1,low=150,high=200),
Hough(threshold=40)