**CV ASSIGNMENT-1 PROJECT REPORT**

**iganjoo-islammdl-molasia**

**PART-2**

Run the code on command line:

make all

./detect.cpp “input\_filename”.png

**1.** **General Convolution:**

The method convolve\_general() accepts two inputs; namely, the input image matrix and a filter(in our case gaussian filter). To handle the image borders, we performed reflection:

· Dimensions of actual image = (r, c)

· Create a new matrix *input\_convolve* of dimensions (r+2, c+2)

· Copy the original image in the new matrix.

· Reflect the first and last row and column of the original image and store it in the new matrix.

· Now, in order to populate the corner four pixels of the new matrix *input\_convolve*, simply copy the border pixels of the original image in the new border pixels.

· Now convolve the matrix *input\_convolve* with the 2d kernel and store the output in a matrix with the same dimensions as the input image.



Result on Informatics.png

**2. Separable Convolution:**

Convolve with two 1-d kernels –

· We start by creating two 1-d Gaussian kernels *Hx* and *Hy*.

· We pass the input image and the two Gaussian kernels as input to the method *convolve\_separable()*.

· We again use ‘Reflection’ to handle the border pixels as described above.

· Note, here, as we use 1-d kernels, we do not need to add the 4 corner pixel values in the *input\_convolve* matrix.

· We first convolve the matrix with *row\_filter(Hx)* and then convolve the result of this convolution with the *column filter(Hy)* and store the output of the final convolution in the matrix *output\_final*.

**Note:** Both step1 and 2 gives similar results and were equally fast. However, there would be significant difference in speed for kernels of larger size. In order to convolve the input image for any kernel we should reflect the entire image, which is computationally expensive. In our case, since we are using a 3\*3 gaussian filter, we haven’t reflected the entire image.



Results on Informatics.png

**3. Convolution with sobel operator:**

Convolve with a sobel operator –

· We call the method ‘*sobel\_gradient\_filter()*’ and pass the input image and a Boolean to the function.

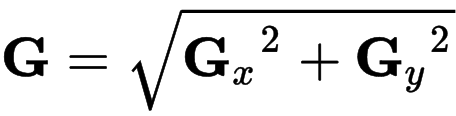
· Now, we create two 3\*3 sobel filters Gx and Gy to get the magnitude and orientation matrix corresponding to x and y directions.

· We pass the input image to the method ‘*reflectImage()*’ which performs reflection on the input image and returns the matrix ‘*input\_convolve*’.

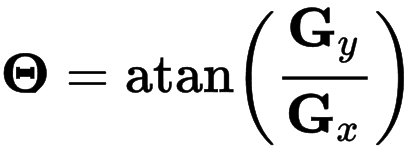
· We now pass the new input matrix and Gx to the method *outputGradientOperator()*, and store the gradient matrix in *output\_Gx*.

· Similarly we get the gradient matrix *output\_Gy*.

· Now we combine the two gradient approximations to get the final magnitude matrix using the formula :

 , where Gx and Gy are values at every pixel in the matrices output\_Gx[][] and output\_Gy[][].

· Now using the formula :



we calculate the orientation at each pixel and store it in the matrix orientation[][].

· We check if the calculated angle falls in one of the below ranges and then assign it the corresponding angle value:

22.5 – 67.5 : **45**

67.5 – 112.5 : **90**

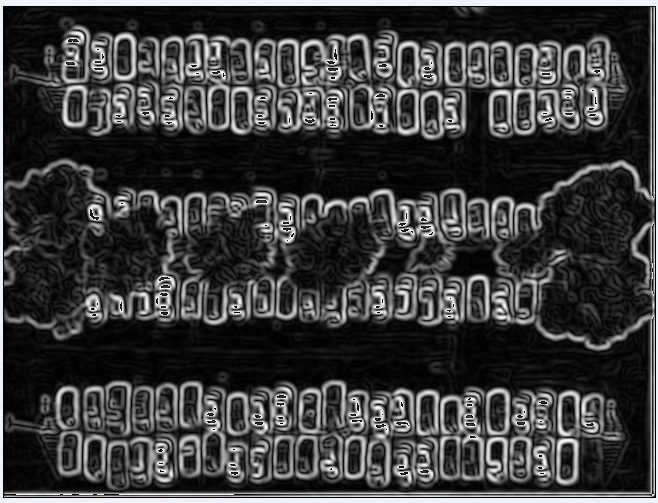
112.5 – 157.5 : **135**

157.5 – 202.5 || 0 – 22.5 || 337.5 – 360 : **0**

202.5 – 247.5 : **45**

247.5 – 292.5 : **90**

If a particular value is less than 0, we convert it into a positive.



Now, as part of edge detection, we implemented Canny Edge detector.

**CANNY EDGE DETECTOR** :

We perform the following steps as part of canny edge detection :

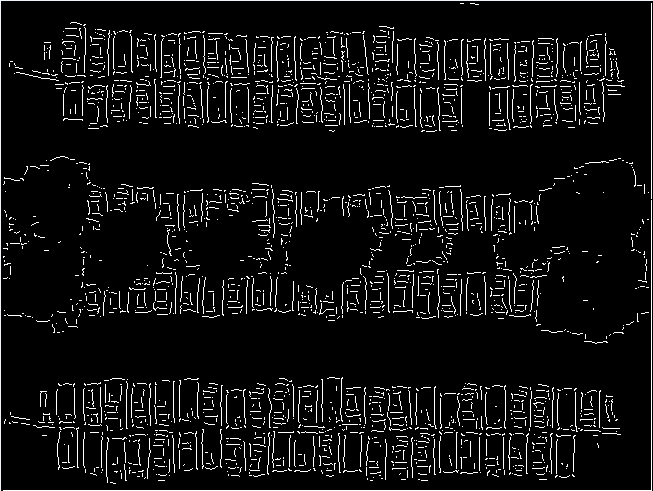
* We convolve the input image with ***gaussian kernel*** to remove the noise.
* In our case, we convolved the image multiple times(8) instead of convolving just once before applying non-maximum suppression as it gave us the best detection rate.
* Now we apply ***sobel*** on the convoluted image(steps described above in part3).
* We now call the method ***nonMaxSuppression()***, and pass it the magnitude matrix, orientation matrix and the final output matrix.
* In *nonMaxSuppression(),* we check the orientation of the current pixel in the magnitude matrix, and according to the angle, we compare the value of that pixel with its two adjacent pixel values. If the magnitude of the pixel under consideration is greater than the adjacent pixel values, we add it to the final output matrix.

Eg : if the current pixel coordinates are (r,c), and the orientation is 0 degrees, then the two adjacent pixels’ coordinates will be (r, c-1) and (r, c+1).

* The next step is to perform ***hysteresis thresholding*** where we pass the output obtained from non-maximum suppression. Here, we set two threshold values, low and high. If the current pixel value is above the high threshold, we simply copy it in the output matrix, if it is greater than the lower threshold and less than the upper threshold, we check if the pixel is connected to another pixel with value greater than the upper threshold in the following manner:

1. We check the neighbors of the current pixel in a ***3\*3 region***, and return true if the value of any one of its neighboring pixels is greater than the upper threshold and add the current pixel in the output matrix.
2. If no such adjoining pixel is found, then we increase the size of the region to ***5\*5*** and search the area. If a pixel with value higher than the upper threshold is found, we return true and add the current pixel in the output matrix.

* We now get an output image with distinct edges.



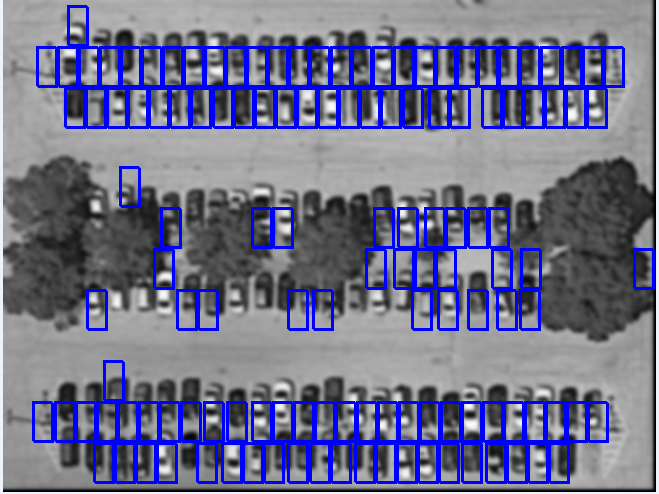
**CAR DETECTION ALGORITHM:**

In order to detect the cars, we devised the following algorithm:

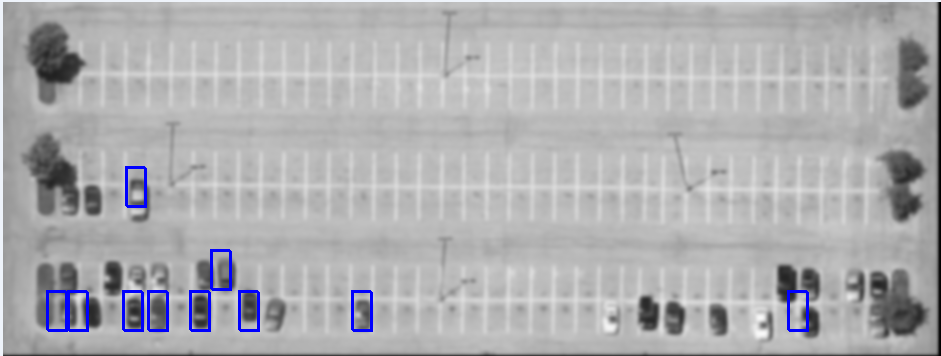
* The *isCar()* function is used to detect cars in the above canny edge output.
* The function takes as input, the output of canny edge and the coordinate of a pixel (r, c), and is called for all the pixel coordinates in the image matrix.
* We use below two parameters to check the presence of a car starting from the row and pixel.

1. In a 22\*42 window, starting from (r, c) we count the number of pixels with gradient zero. Note that we only count the zeroes in the perimeter of the 22\*42 rectangle and not the area.
2. If the count of zeroes from point 1) exceeds a threshold, we count the number of pixels with gradient 255 in the area of 20\*40 starting from (r+1, c+1).
3. If the count of white pixels from point 2) exceeds a threshold we return the count of white pixels, indicating the presence of a car, else we return 0.
4. The count of white pixels from point 3) is used to detect confidence of each car which is calculated as:

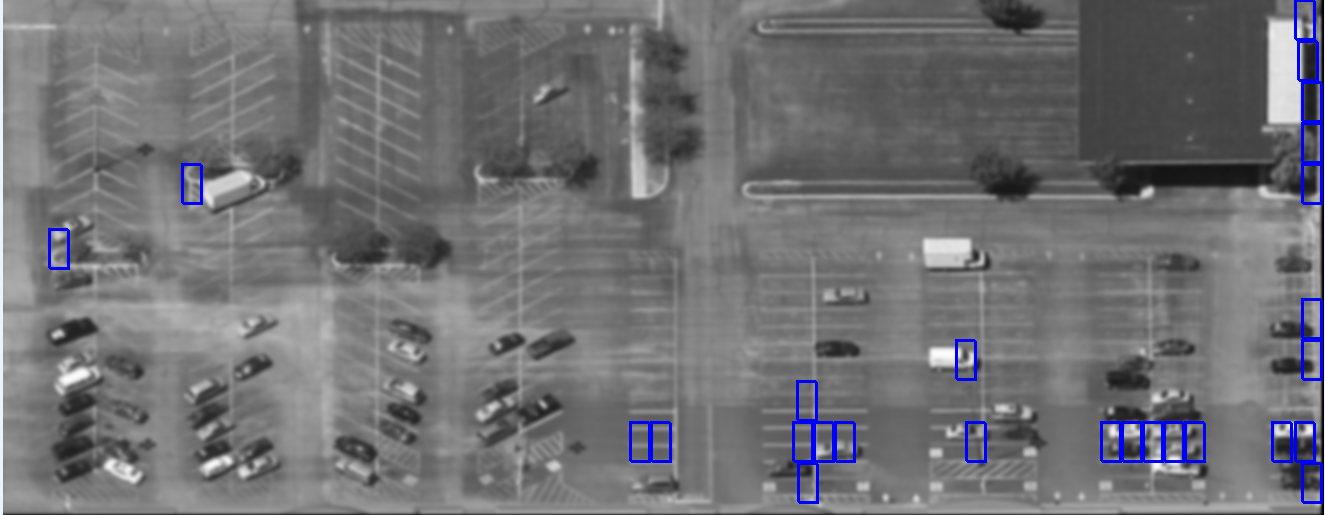
After applying the above algorithm, we get the below result:



Result on Informatics.png



Result on SRSC.png



Result on Plaza.png

**NOTE on RESULTS:** We ran a1-eval.py provided to us for checking accuracy of our results, but we are getting a precision of 0. This may be possible because the code is looking for exact same coordinates in the ground truth file and detected.txt.

**FUTURE IMPROVEMENTS:**

* The algorithm devised for car detection can be made more robust to detect cars of all dimensions.
* Currently, it only detects cars of dimensions 20\*40 within a window 22\*42.
* The low and high thresholds used are in accordance of our experiment with the current test images, however to get better results for any image, the thresholds should be set dynamically.
* We can experiment with Harris corner detection, Hough transform and template matching to get better results in future.