

Report – Assignment 1

Syam Sundar Herle

Neelam Tikone

Gulshan Madhwani

PART 1: WATERMARKING

Part 1.1: Magnitude

The following input image is given to find the magnitude (Spectrogram) of the image.



The input image is read and changed to (0 - 255) matrix and fed into FFT function to get real and imaginary part of same dimension of the input image. Using below function a magnitude view of the image is produced.

$$E(i, j) = \sqrt{R(i, j)^2 + I(i, j)^2}$$

The magnitude view of the input image is,



Part 1.2 :

For reducing the noise in image noise1.png, we tried to hard-code the value of real and imaginary pixel where the value of $\sqrt{R^2 + I^2} > 1$ and we changed the pixel having this condition true to the neighboring pixel value. We made cautious change by not changing the center of the real and imaginary value. Once this is done IFT is done to get noise-free image. The output of part 1.2 is given below,



Part 1.3 (a) Adding watermark:

Here we have made some design decision, following are the parameters we fixed for this problem,

$$R(\text{radius}) = 128$$

$$\alpha = 5$$

$$l = 16$$

For adding watermark in real-part we created random binary bits of length l for integer N (given as parameter) we found 16 distinctive points in real using simple logic of circle ($x = c_x + r * \cos(\vartheta)$ and $y = c_y + r * \sin(\theta)$ for symmetric part $x = c_x + r * \cos(\vartheta + \pi)$ and $y = c_y + r * \sin(\theta + \pi)$) and modified the pixel value using the below formulae,

$$R(i, j) = R(i, j) + \alpha |R(i, j)| v_i$$

Where the v_i is the one of bit of binary vector formed by $srand(N)$ function, the binary vector v is of length l , and the resultant real and imaginary part is sent to the IFT and the water marked image is produced, the following is the output for integer $N=300$. Where the watermark forms almost invisible pattern distortion.



When the adding watermark fails

The change in any magic parameter (like decrease in radius) produces more distortion of the image. If the value of $\alpha = 120$ the distortion is quite visible in the image.



Part 1.3 (b) Detecting Watermark

For detecting the watermark, we tried to extract the changed real part of the image in form of vector c , and we once again created a binary vector v of length l for given integer N and we

tried to calculate the correlation coefficient of both the vectors c and v . The correlation coefficient is calculated using the following equation,

$$cr = \frac{\sum (c_i - \mu_c)(v_i - \mu_v)}{\sqrt{var(c)var(v)}}$$

If the correlation coefficient is greater than threshold t we will declare the image is watermarked of integer N .

Design parameters (magic parameters):

The design parameters we fixed for this problem are

$$R(\text{radius}) = 128$$





$$\alpha = 5$$





$$l = 16$$

$$t = 0.3$$

Qualitative Results:

We tried to watermark certain images (4) and we came through some good results,

Input	Output	Integer	Watermarked?
		700	True (t=0.322)
		300	True (t=0.620995)

		550	True (t=0.68132)
		3050	True (t=0.668103)

We observed the quality of the image is preserved and no distortion is seen (except the last one) by using the designed parameters.

Quantitative results:

For the purpose of quantitative result's we check whether the above 4 images has been watermarked of some random numbers and we record the count of false positive,

Image 1 was watermarked with integer $N=700$, the false positive count is 19.

Image 2 was watermarked with integer $N=300$, the false positive count is 13.

Image 3 was watermarked with integer $N=550$, the false positive count is 16.

Image 4 was watermarked with integer $N=3050$, the false positive count is 9.

Scenarios for failure of detecting watermark

When the design parameter in detection is different of adding watermark part. The algorithm fails to detect the watermark of the same integer N . If the r is decreased in adding as well as detection part the algorithm gives high false positive. Generally, the change of parameter results in increase in false positive count.

Output Part1.3 (b) : A correlation value will be printed and a Boolean will be printed denoting watermarked or not

How to run the program

Part 1.1 : `./watermark 1.1 input.png output.png`

Part 1.2 : `./watermark 1.2 input.png output.png`

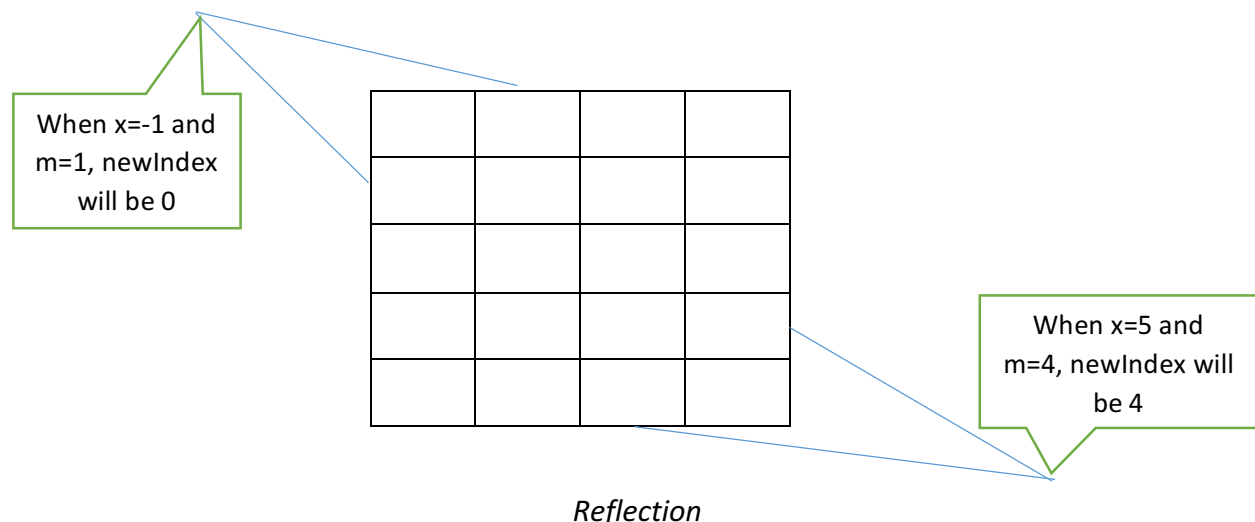
Part 1.3 (add): `./watermark 1.3 input.png output.png add N`

Part 1.3 (check): `./watermark 1.3 input.png output.png check N`

2.1 Convolution using 2D kernel:

2D kernel is created using `createGaussKernel()` function. Convolution on the image is performed in `convolve_general()` function. Boundary conditions are handled using a function called `border()` which takes two input as an index and count for row/columns. Boundary conditions are handled using reflection.

Reflection – Let 'x' be the index and 'm' is the size of row/column and if the image goes beyond left or top boundary, new index is calculated as $-x - 1$ and if the index goes beyond right and bottom boundary, new index is calculated as $(2 \times m) - x - 1$.



2.2 Convolution using separable kernel:

We have hard-coded the values for Sobel matrices and divide the 2-D matrix into row and column matrix of 1D using the function `getRowFilter` and `getColFilter` functions. `getRowFilter()`

just extracts the last row of the input matrix and `getColFilter()` extracts the last column. `convolution_separable()` function is used to convolve the input using separable matrices. Initially convolved using column filter and intermediate result is stored in *SDoublePlane intermediate* and then *intermediate* is convolved using row filter. Boundary conditions are handled using reflection as explained above.

To run this part, uncomment line number 684 and 685 in `detect.cpp` file.

How to run the code:

Part 2:

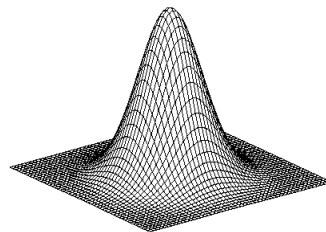
- Open terminal and go to the directory "Part 2". Make sure it contains "Makefile".
- Type `make <enter>`.
- To run the code, type `./detect <image path>`.
- The output will be generated in the same folder; an image file named `detected.png` and a text file called `detected.txt`.

2.3 Edge Detection:

As the very first step in this process, we convolve the image using the Gaussian filter which is obtained using the function `createGaussKernel(double (&gaussKernel)[5][5], double sigma)`

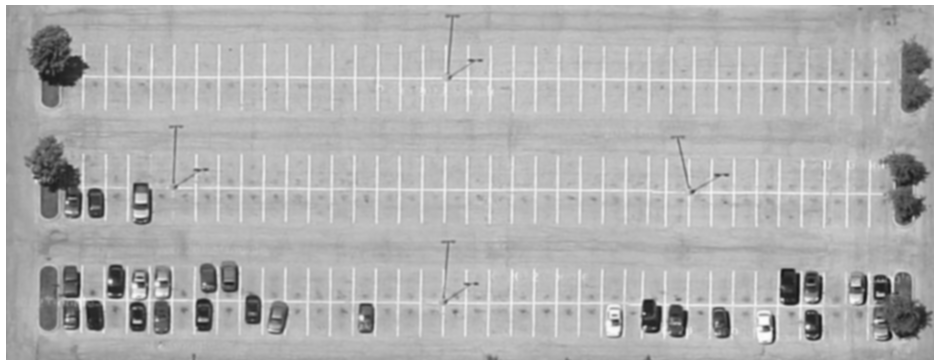
The above function creates a Gaussian kernel of the size `[5][5]` and it takes the sigma value to be considered as an argument to the function. Once, the Gaussian kernel is applied to the image, we remove the noise which is present in the image which is why we blur the image. The Gaussian Kernel is created using the formula:

$$h_{\sigma}(u, v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{2\sigma^2}}$$



(Gaussian)

The image obtained after applying the Gaussian kernel is:



Gaussian Kernel on SRSC.png

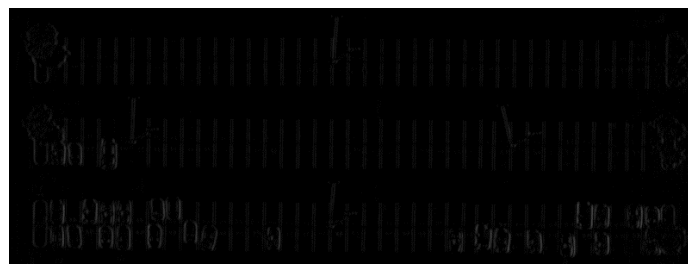
Once we apply the Gaussian kernel, we calculate the gradient of the image using the **sobel operators**. The S_x is used to find the gradient in x direction and S_y for Y direction

$$\frac{1}{8} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

S_x

$$\frac{1}{8} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

S_y



(SRSC.png after applying Sobel gradient filter in x direction (Sx))



(SRSC.png after applying Sobel gradient filter in y direction(Sy))

Once we have the image gradients using the sobel operator. We get the two Matrices *sobelxImg* and *sobelyImg* consisting of the gradients in the x and y direction.

As we have the gradients in both the directions, we get the image with the edges after using the function *find_edges(sobelxImg1,sobelyImg1)*

The function *find_edges* takes two arguments, which are: *sobelxImg1*, The gradients obtained after the Sobel operator in x direction and *sobelyImg1*, The gradients obtained after the Sobel operator in y direction.

The addTwo function *addTwo(input_gX,input_gY)* takes in input these two gradients and generates a matrix which produces an output of the addition of these two gradients. Hence, we get the edges in the image.



Image obtained after adding the two gradients

Computing orientation and Magnitude:

Once we have the Matrix containing the edges of the image, we calculate the orientation and magnitude of the image and store it in the Orientation and the Magnitude Matrices where Magnitude of the image gives us the edge strength.

The formula used to calculate Magnitude is:

$$\|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$$

The formula used to calculate orientation is:

$$\theta = \tan^{-1} \left(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x} \right)$$

Once we have the orientation and magnitude value for each pixel, categorize the orientation in two directions, 0 and 90 for simplicity purpose. The pixels whose orientation is between *45 and 135* or between *225 and 315*, the Orientation for that pixel is updated as *90* and similarly, The pixels whose orientation is lesser than *45 and greater than 315* or between *225 and 135*, the Orientation for that pixel is updated as *0*.

Once we have updated the orientation into two categories, we find the orientation of the entire image by finding the ratio of the total number of pixels having angle value 0 and total number of pixels having angle value 90. This orientation value is then used to decide that whether the Template in the horizontal direction needs to be used or the Template in the vertical direction needs to be used.

For choosing the template, the threshold I have considered is 0.7 i.e. if the ratio value is above 0.7, I use the vertical car template to detect the cars parked in the parking space else if the value is 0.7 or below, I use the horizontal car template to detect the cars parked in the parking space.

Edge Thinning:

For thinning of edges, we first check the magnitude of that pixel. If we see that the magnitude of the pixel is lesser than 21, we update the greyscale of the image as 0 else we follow the following steps:

Check orientation of that pixel. If the pixel has the orientation angle 90, check the magnitude values of the two pixels in the right as well as two pixels in the left of that particular pixel and the greyscale value of the pixel having the highest magnitude is updated as 255 in its SdoublePlane Matrix and rest of the 4 pixels value is updated as 0.

Similarly, if the pixel has the orientation angle 0, check the magnitude values of the upper two as well as lower two of that particular pixel and the greyscale value of the pixel having the highest magnitude is updated as 255 in its SdoublePlane Matrix and rest of the 4 pixels value is updated as 0. Hence we choose the pixel which has the highest magnitude.

The image obtained after Edge Thinning is:

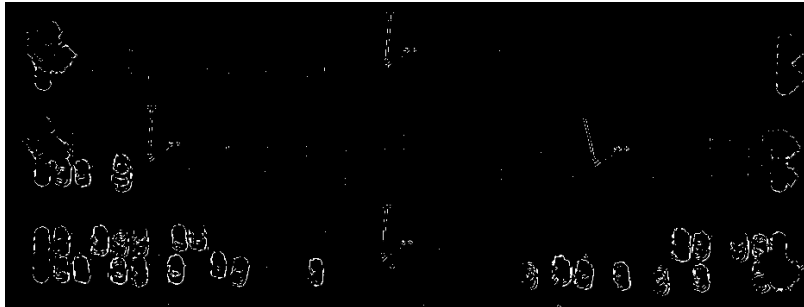


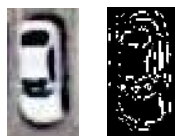
Image obtained after thinning of edges SRSC.png

Once we have thinned the edges, we used the function *match_template* to match the edges of the Image and the template.

I have used two templates for matching:



Horizontal car Template (temp2.png)



Vertical car Template (temp.png)

As mentioned above, if the orientationVal of the image obtained is above 0.7, we use the vertical car template else we use the horizontal car template.

The matching technique I am using to find the edges of the car in the image is pixel to pixel matching. I match each pixel in the template to each pixel in the image. If both the pixels are having the value 255 i.e. if both the pixels are white then I increase the count by 1. Hence, The following thresholds I have kept for matching the horizontal and the vertical templates:

Threshold of the Vertical car Template:

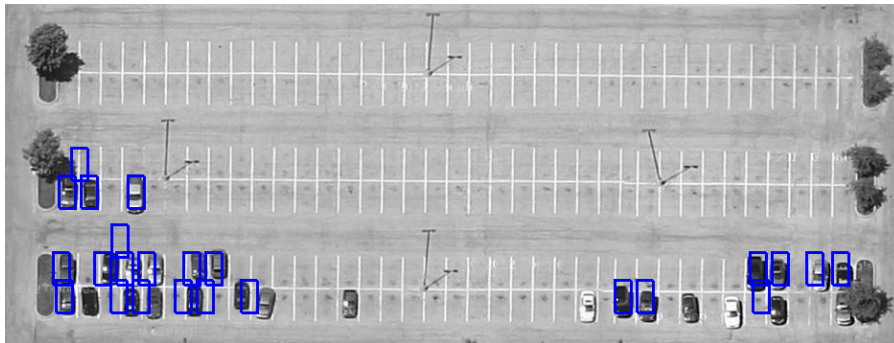
If the total count of the matched pixels are greater than 22, then I consider it as a match and the co-ordinates for which I get the matched images are updated as the co-ordinates to plot the box.

Threshold of the Horizontal car Template:

If the total count of the matched pixels are greater than 7, then I consider it as a match and the co-ordinates for which I get the matched images are updated as the co-ordinates to plot the box.

Results

For SRSC.png:



Edges Detected SRSC.png

Total cars: 28

Total detected (True Positives): 20

Total undetected (False Negative): 8

False Positives: 3

Accuracy: $20/28 = 71.42\%$

It does not work when:

Some of the black cars were not detected because the edges did not have magnitude high enough to get brightened during edge thinning process. Hence they were set as 0 and did not have enough pixels to get detected.

It works when:

The cars are light colored and the windows and edges of the cars are detected properly

Informatics.png:



Edges Detected Informatics.png

Total cars: 128

Total detected (True Positives): 127

Total undetected (False Negative): 1

False Positives: 13

Accuracy: $127/128 = 99.21\%$

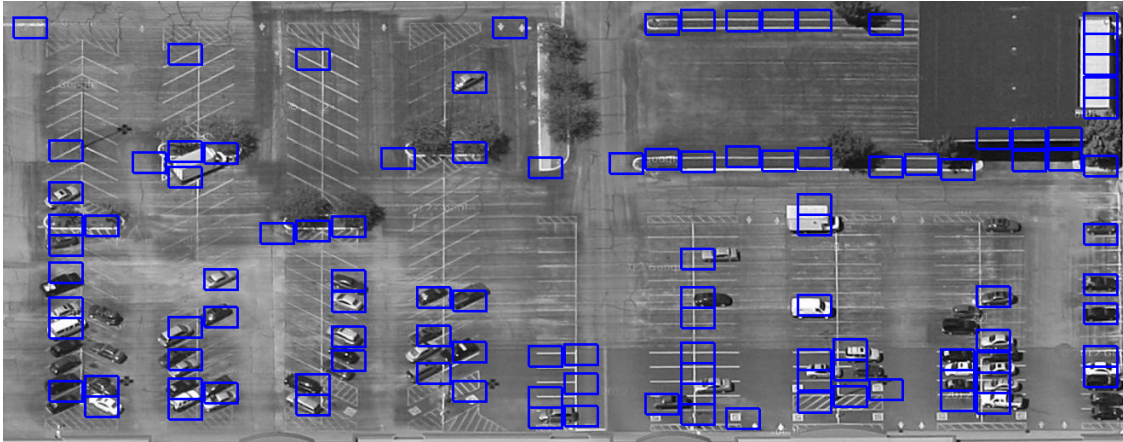
It does not work when:

It successfully detects the blank space where the cars are not parked. Some trees had edges which high magnitude pixels. Hence it could not be neglected and those were brightened during edge thinning process.

It works when:

The cars are light colored and the windows and edges of the cars are detected properly

Plaza.png



Edges Detected Plaza.png

Total cars: 66

Total detected (True Positives): 56

Total undetected (False Negative): 10

False Positives: 54

Accuracy: $56/66 = 84.84\%$

It does not work when:

We can see that there are too many false positives obtained in the image which is because the pixel strength in that part of the parking lot was too high to be neglected. Similarly, the magnitude of the pixels over the building was too high to be neglected, Hence during the edge thinning process, those pixels were brightened.

It works when:

The cars are light colored and the windows and edges of the cars are detected properly

How can we improve?

- Using certain corner detection algorithms and matching the corners of the images.
- Hough Transform can be used to detect the orientation using which we can make decision of whether to use the horizontal or the vertical car template.
- SIFT algorithm can be used on the edge pixels for a scale invariant detection and we can detect using the keypoint descriptors