CMPE 362

DIGITAL IMAGE PROCESSING

HOMEWORK 2 REPORT

Instructor: Aslı GENÇTAV
Name - Surname: Neslihan Pelin METİN
Student Number: 71047171244

Introduction

The main purpose of this project is to learn about template matching and image filtering in the frequency domain in the field of image processing. It also aims to test what we have learned in the classes using Python.

Literature Review

◊ Template Image: It is the image that is desired to be found in the large image and is smaller compared to the main(large) image.

◊ Mean Value of an Image: It is the average of pixel values of the image, and it is calculated by this formula where P is an image:

$$P_{mean} = \frac{1}{m * n} \sum_{x=1}^{m} \sum_{y=1}^{n} P(x, y)$$

◊ Template Matching: It allows the location of the template image to be found in the main image by comparing the template image with the main image.

1. Correlation: It is a similarity measure between the main image and the template image. It is the filtered version of the main image with a template image and can also be calculated using the following formula.

$$\sum_{x=1}^{m} \sum_{y=1}^{n} P(x, y) * T(x, y)$$

2. Zero-Mean Correlation: It is a different form of *correlation*. Unlike the other, in this, the average value of the template image is subtracted from all pixels in the template image. Then the new image created is used in the *correlation* formula as can be seen below.

$$\sum_{x=1}^{m} \sum_{y=1}^{n} P(x, y) * (T(x, y) - T_{mean})$$

3. Sum of Squared Difference (SSD): It is a measure of dissimilarity between two images. It is computed by summing the squared differences between the pixel values of the template and the target image at each position. A lower SSD value indicates greater similarity between the template and the target image.

$$\sum_{x=1}^{m} \sum_{y=1}^{n} (P(x, y) - T(x, y))^2$$

4. Normalized Cross-Correlation (NCC): It is a measure of similarity between two images. It is computed by dividing the correlation between the template and the target image by the product of their standard deviations.

$$\frac{\sum_{x=1}^{m} \sum_{y=1}^{n} (P(x, y) - P_{mean}) * (T(x, y) - T_{mean})}{\left[\left(\sum_{x=1}^{m} \sum_{y=1}^{n} (P(x, y) - P_{mean})^2 \right) * \left(\sum_{x=1}^{m} \sum_{y=1}^{n} (T(x, y) - T_{mean})^2 \right) \right]^{0.5}}$$

◊ Image Filtering in Frequency Domain: This technique is used to increase the sharpness or smoothness of the images.

1. Butterworth Low Pass Filter: Increases the smoothness of the images. It can be calculated as in the following image where *D(u,v)* denotes Euclidean distance of (*u*, *v*) to the center (h/2, *w*/2) where h and *w* are the height and width of filter *H*. Also, D_0 and *n* define the shape of the filter H.

$$H_{LowPass}(u, v) = \frac{1}{1 + [D(u, v)/D_0]^{2*n}}$$

2. Butterworth High Pass Filter: It is the inverse operation of the Butterworth Low Pass Filter that we mentioned above. Increases the sharpness of the images. And can be calculated with the formula below.

$$H_{HighPass} = 1 - H_{LowPass}$$

Problem Statement & Implementation

◊ Part 1

In this question, we were asked to find the location of a smaller image on a main image by using 4 main techniques that we explain in *Literature Review.* These methods are called *Correlation, Zero-Mean Correlation, Sum of Squared Difference*, and *Normalized Cross-Correlation*. I will explain how I implemented them one by one. But first I must define some other methods to be able to explain them more easily.

- *CalculateMean(im):* Calculates the mean of the image given as a parameter.

```
def calculateMean(im):        # this method helps to calculate the mean of the parameter image
  i = 0
  j = 0
  sum = 0
  result = 0

  while(i < len(im)):           # calculating the sum of pixel values
    while(j < len(im[0])):
      sum += im[i][j]
      j += 1
    i += 1
    j = 0

  result = sum /(len(im) * len(im[0]))    # dividing the sum with the height and width of the image
  return result                          #returning the result
```

- *newP(P, T):* Resizes the image so that we skip doing processes to pixels that are closer to the boundary of the image.

```
def newP(P, T):     # reshapes the image given in the parameter
  x = len(P)
  y = len(P[0])
  i = (len(T)-1)//2
  j = (len(T[0])-1)//2
  newP = P[i:x-i,j:y-j]
  return newP
```

- *ssdMeasure(P, T):* Calculates the sum of squared difference between two images of the same size.

```
def ssdMeasure(P, T):                    # computes SSD between P and T that are same size
    P = P.astype(np.float64)             # converting images to float64 datatype
    T = T.astype(np.float64)

    ssd = np.sum((P - T) ** 2)           # compute SSD between P and T
    return ssd
```

- *stdForNcc(im,mim):* Calculates the standard deviation of the given image and the mean of the image as a parameter.

```
import math
def stdForNcc(im,mim):      #calculates the standard deviation of the given image and the mean of the image as a parameter
  i = 0
  j = 0
  sum = 0

  while(i < len(im)):
    while(j < len(im[0])):
      sub = abs(im[i][j]- mim)
      sum += math.pow((sub),2)
      j += 1
    i += 1
    j = 0

  sum = math.pow(sum,0.5)
  return sum
```

1. Correlation

To implement the *correlationMeasure (P, T),* I used the filtering property of *Correlation* after changing the types of images to the *np.float(64)* type. Then by using *newP* method, the main image is resized and

applied *cv2.filter2d* function which filters the image with the kernel that you give as a parameter. At this point, we have completed template matching, so the program shows the new image. *Correlation* measures the similarity as we stated before and we must find pixels with the maximum value. To do this, we got help from *cv2.minMacLoc* which gives the locations of the pixels with minimum and maximum values. After that, by using *cv2.rectangle* we are marking the area with the maximum value. Finally, we are showing the image that is marked on the screen.

```
# this method calculates the correlation measure and prints it
# then it finds the highest pixel values and shows it on the image in a rectangle shape
def correlationMeasure(P,T):
  img = P.copy()
  P = np.float64(P)                  # converting images to float64 datatype
  T = np.float64(T)
  nP = newP(P,T)                     # reshape the main image
  result = cv2.filter2D(src = nP, ddepth=-1, kernel=T)   # calculates the correlation measure by filtering the image by the template
  plt.imshow(result, cmap = 'jet')                      # show the correlation measure
  plt.title('Correlation')
  plt.colorbar()
  plt.show()

  w, h = T.shape[::-1]
  min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)    # find the location of maximum & minimum values
  top_left = max_loc                                            # since correlation is a similarity measure choose max value's location
  bottom_right = (top_left[0] + w, top_left[1] + h)
  cv2.rectangle(img,top_left, bottom_right, 0, 2)         # draw the rectangle on the image
  plt.imshow(img,cmap = 'gray')                           # and print it
  plt.title('Detected Point')
  plt.show()
```

2. *Zero-Mean Correlation*

As I mentioned before, *Zero-Mean Correlation* is almost the same process as *Correlation.* The only remarkable difference is that we are calculating the mean of the *Template image* and simply subtracting from all the pixels of the template image itself using *calculateMean* method. Apart from that, all stages are the same.

```
# this method calculates the zero mean correlation measure and prints it
# then it finds the highest pixel values and shows it on the image in a rectangle shape
def zeroMeanCorrelationMeasure(P,T):
  img = P.copy()
  P = np.float64(P)             # converting images to float64 datatype
  T = np.float64(T)

  tmean = calculateMean(T)
  T = T - tmean                 # calculates new T by substracting its mean from it

  nP = newP(P,T)                # reshape the main image
  result = cv2.filter2D(src = nP, ddepth=-1, kernel=T)   # calculates the correlation measure by filtering the image by the template
  plt.imshow(result, cmap = 'jet')                      # show the zero mean correlation measure
  plt.title('Zero Mean Correlation')
  plt.colorbar()
  plt.show()

  w, h = T.shape[::-1]
  min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)    # find the location of maximum & minimum values
  top_left = max_loc                                            # since correlation is a similarity measure choose max value's location
  bottom_right = (top_left[0] + w, top_left[1] + h)
  cv2.rectangle(img,top_left, bottom_right, 0, 2)         # draw the rectangle on the image
  plt.imshow(img,cmap = 'gray')                           # and print it
  plt.title('Detected Point')
  plt.show()
```

3. *Sum of Squared Difference*

In this method, we determine the *height* and *width* of the input and *template images* using the *shape* attribute of *numpy* arrays. Then, we are creating a new *result* image of the same size as the input image to store the SSD measure. Next, for each pixel in the input image, it extracts a patch of the input image that corresponds to the template, computes the SSD measure between the patch and the template using the function *ssdMeasure*, and stores the result in the corresponding location in the *resulting* image. Afterward, the code normalizes the result array to be in the range [0, 1] and we reshape it using the function *newP*. Consequently, it finds the location of the minimum value in the result array using the *cv2.minM*axLoc function. We draw a rectangle around the location of the minimum value in the input image using *the cv2.rectangle* function and display the resulting image using *plt.imshow()*.

```python
def sumOfSquaredDifferenceMeasure(P, T):
    img = P.copy()

    input_height, input_width = P.shape                # get size of input and template images
    template_height, template_width = T.shape

    result = np.zeros_like(P, dtype=np.float64)        # initialize output image

    margin_y = template_height // 2                      # define the region where we will compare the input image and the template
    margin_x = template_width // 2


    for y in range(margin_y, input_height - margin_y):
        for x in range(margin_x, input_width - margin_x):
            Pi = P[y - margin_y : y + margin_y + 1, x - margin_x : x + margin_x + 1]    # get the patch of the input image that corresponds to the template
            val = ssdMeasure(Pi, T)                                                    # compute SSD between P and T
            result[y, x] = val                                                      # set the corresponding pixel in the output image

    result = (result - np.min(result)) / (np.max(result) - np.min(result))    # normalize output image to be in the range [0, 1]

    result = (result * 255).astype(np.uint8)    # convert output image to 8-bit unsigned integer
    nP = newP(result,T)                             # reshape the output image

    plt.imshow(nP, cmap = 'jet')                    # show the sum of squared difference measure
    plt.title("Sum of Squared Difference")
    plt.colorbar()
    plt.show()

    w, h = T.shape[::-1]
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(nP)    # find the location of minimum values
    top_left = min_loc                                  # since correlation is a dissimilarity measure choose min value's location
    bottom_right = (top_left[0] + w, top_left[1] + h)
    cv2.rectangle(img,top_left, bottom_right, 0, 2)         # draw the rectangle on the image
    plt.imshow(img,cmap = 'gray')                           # and print it
    plt.title('Detected Point')
    plt.show()
```

4. *Normalized Cross-Correlation*

Firstly, we normalize the template and the image to have zero mean and standard deviation. In next, we computed the cross-correlation between the normalized template and the normalized image using the *filter2D* function from the *OpenCV* library. The resulting cross-correlation map is then normalized by dividing it by the product of the standard deviations of the template and the image. Also, the location of the maximum value in the cross-correlation map is used to determine the position of the template in the image. By using *cv2.rectangle* we are marking the area with the maximum value. Finally, we are showing the image that is marked on the screen.

```python
def normalizedCrossCorrelationMeasure(P,T):
  img = P.copy()
  nP = newP(P,T)
  P = np.float64(nP)             # converting images to float64 datatype
  T = np.float64(T)
  tmean = calculateMean(T)      # calculates mean of image T
  sumt = stdForNcc(T, tmean)    # calculates standard deviation of image T

  T = T - tmean
  T_norm = T / (sumt)         # calculates normalized version of image T

  pmean = calculateMean(P)      # calculates mean of image P
  sump = stdForNcc(P, pmean)    # calculates standard deviation of image P

  P = P - pmean
  P_norm = P / (sump)         # calculates normalized version of image P

  result = cv2.filter2D(src = P_norm, ddepth=-1, kernel=T_norm)   # calculates the normalized cross correlation measure by filtering the image by the template

  result = (result )/(sumt*sump)         # calculates normalized version of resulting image

  plt.imshow(result, cmap = 'jet')                    # show the result
  plt.title('Normalized Cross-Correlation')
  plt.colorbar()
  plt.show()

  w, h = T.shape[::-1]
  min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)    # find the location of maximum & minimum values
  top_left = max_loc                                            # since ncc is a similarity measure choose max value's location
  bottom_right = (top_left[0] + w, top_left[1] + h)
  cv2.rectangle(img,top_left, bottom_right, 0, 2)         # draw the rectangle on the image
  plt.imshow(img,cmap = 'gray')                           # and print it
  plt.title('Detected Point')
  plt.show()
```

◊ Part 2

This question wants us to perform Butterworth Low/High Pass Filtering on an image to see how the parameters are affecting the filter.

1. *butterworthLowpassFilter(img, D0, n)*

To implement this method we begin by applying the Fourier transform to the input image using *np.fft.fft2()*. Then, the Fourier transform is shifted using *np.fft.fftshift()*. And we create a new filter H as a 2D array of zeros, with the same dimensions as the input image. For each pixel in the filter, the distance D from the center of the filter is computed using the distance formula. After that, we apply the Butterworth low-pass filter formula to each pixel of the filter to compute its value. The filtering result in the frequency domain is computed by multiplying the shifted Fourier transform Fshift by the filter H.

In the end, we apply the inverse Fourier transform to the filtered result using *np.fft.ifft2(),* and the absolute value of the resulting array is computed using *np.abs()* so that the resulting image can be found. As a final remark, we show the magnitude of each process using *plt.imshow()* and *plt.show()*.

```
# computes and prints the magnitudes and filtered result
def butterworthLowpassFilter(img, D0, n):
  F = np.fft.fft2(img)              # applying fourier transform

  plt.imshow(np.log(1 + np.abs(F)), cmap = 'jet')       # printing the magnitude of fourier transform
  plt.title("Fourier Transform Magnitude")
  plt.colorbar()
  plt.show()

  Fshift = np.fft.fftshift(F)             # shifting the fourier transform

  plt.imshow(np.log(1 + np.abs(Fshift)), cmap = 'jet')       # printing the magnitude of shifted fourier transform
  plt.title("Shifted Fourier Transform Magnitude")
  plt.colorbar()
  plt.show()

  M,N = img.shape
  H = np.zeros((M,N), dtype = np.float64)         # creating new filter and filling it with zeros

  for i in range(M):
    for j in range(N):
      D = np.sqrt((i-M/2)**2 + (j-N/2)**2)        # applying the Butterworth Low Pass Filter Formula
      H[i,j] = 1/ (1 + (D / D0)** (2*n))        # and filling into the filter H

  plt.imshow(H, cmap='gray')       # printing the magnitude of filter H
  plt.title("Filter H")
  plt.show()

  Gshift = Fshift * H         # finding the filtering result in frequency domain

  plt.imshow(np.log(1 + np.abs(Gshift)), cmap = 'gray') # printing the filtering result in frequency domain
  plt.title("Filtering Result in Frequency Domain")
  plt.show()

  G = np.fft.ifftshift(Gshift)              # inverse shifting the filtering result in frequency domain
  g = np.abs(np.fft.ifft2(G))               # and then applying inverse fourier transform to it => this is our result

  plt.imshow(g, cmap='gray')                # showing the result image
  plt.title("Filtering Result in Spatial Domain")
  plt.show()
```

2. *butterworthHighpassFilter(img, D0, n)*

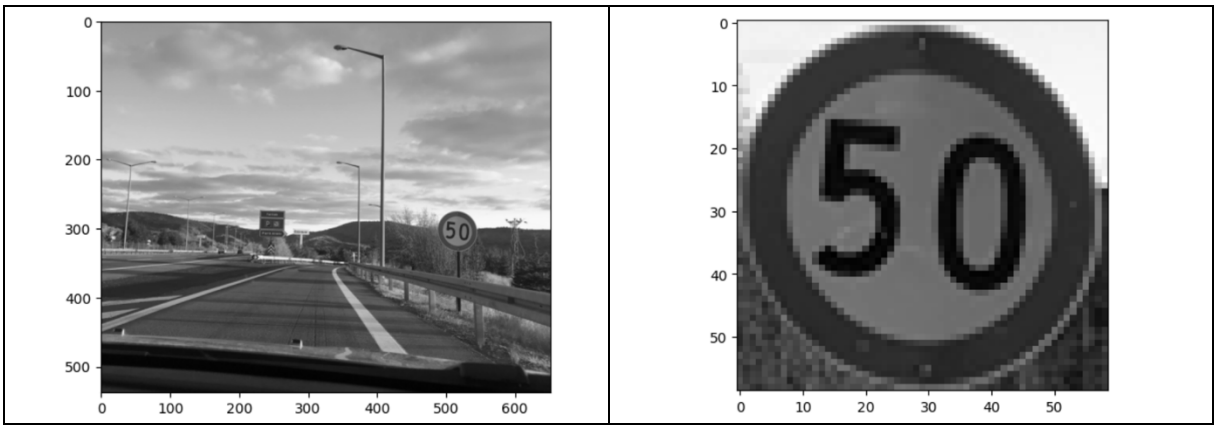
At this point, we know that this method is written almost the same with the method above. The only difference is that to apply the function of Butterworth high pass filter, we must subtract the calculated value of the low pass filter from 1. The rest is composed of the same steps.

```
# computes and prints the magnitudes and filtered result
def butterworthHighpassFilter(img, D0, n):
  F = np.fft.fft2(img)              # applying fourier transform

  plt.imshow(np.log(1 + np.abs(F)), cmap = 'jet')       # printing the magnitude of fourier transform
  plt.title("Fourier Transform Magnitude")
  plt.colorbar()
  plt.show()

  Fshift = np.fft.fftshift(F)             # shifting the fourier transform

  plt.imshow(np.log(1 + np.abs(Fshift)), cmap = 'jet')       # printing the magnitude of shifted fourier transform
  plt.title("Shifted Fourier Transform Magnitude")
  plt.colorbar()
  plt.show()

  M,N = img.shape
  H = np.zeros((M,N), dtype = np.float64)         # creating new filter and filling it with zeros

  for i in range(M):
    for j in range(N):
      D = np.sqrt((i-M/2)**2 + (j-N/2)**2)        # applying the Butterworth High Pass Filter Formula
      H[i,j] = 1- 1 / (1 + (D / D0)** (2*n))        # and filling into the filter H

  plt.imshow(H, cmap='gray')       # printing the magnitude of filter H
  plt.title("Filter H")
  plt.show()

  Gshift = Fshift * H         # finding the filtering result in frequency domain

  plt.imshow(np.log(1 + np.abs(Gshift)), cmap = 'gray') # printing the filtering result in frequency domain
  plt.title("Filtering Result in Frequency Domain")
  plt.show()

  G = np.fft.ifftshift(Gshift)              # inverse shifting the filtering result in frequency domain
  g = np.abs(np.fft.ifft2(G))               # and then applying inverse fourier transform to it => this is our result

  plt.imshow(g, cmap='gray')                # showing the result image
  plt.title("Filtering Result in Spatial Domain")
  plt.show()
```

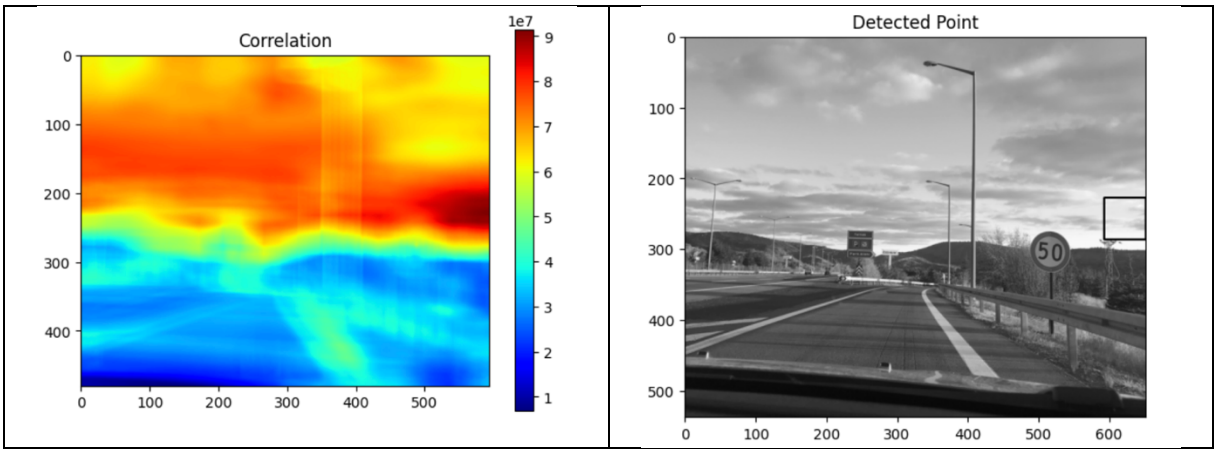
Testing

◊ *Part 1a*

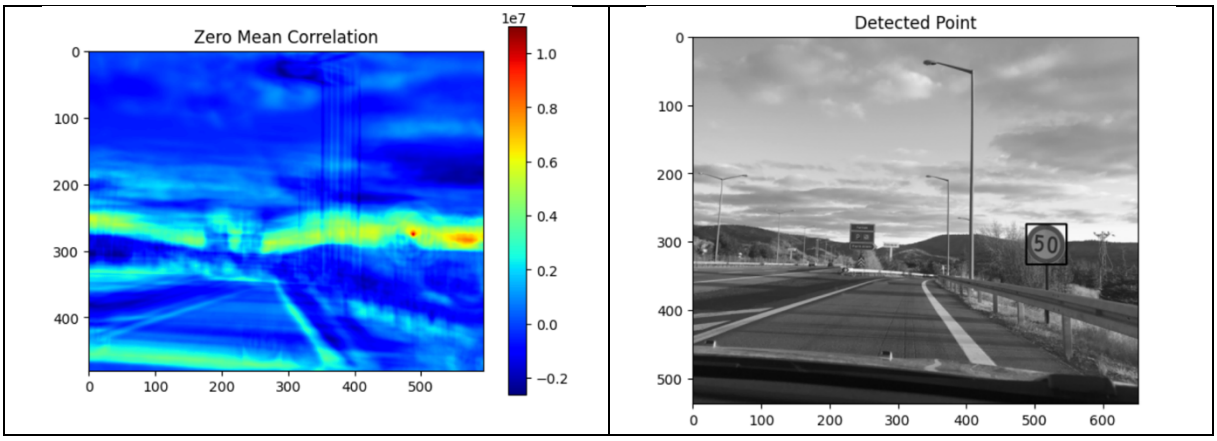
In these outputs, the measures of Normalized Cross-Correlation and Zero-Mean Correlation looks almost the same, but their values shown in the color bar are different. And the method Correlation could not find the patch we were looking for. Other methods find the correct places.



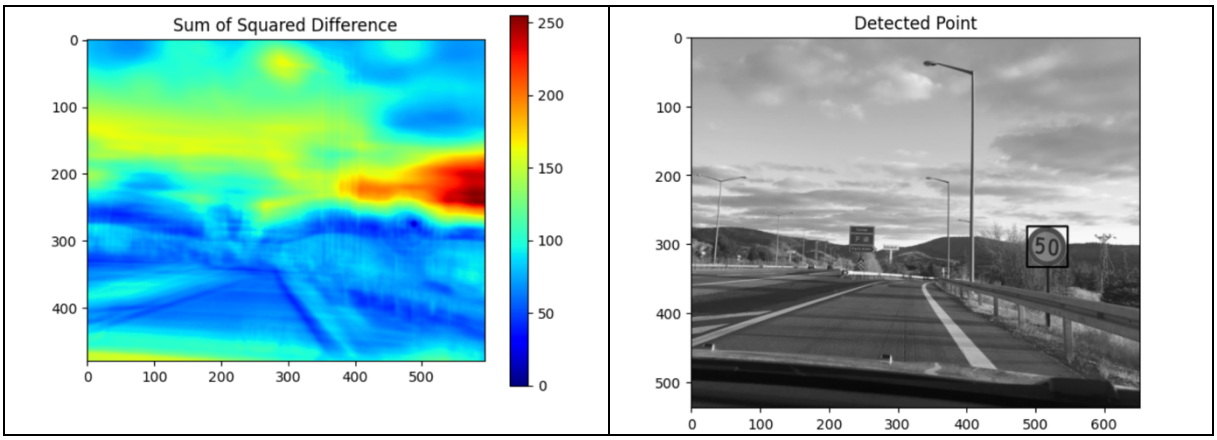
(Main and Template images of Part 1a)



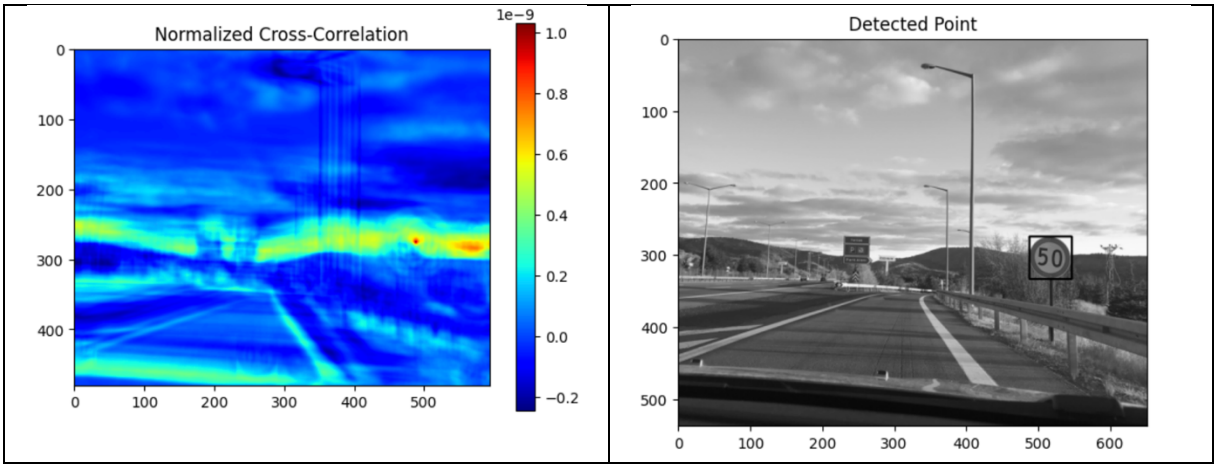
(Correlation Measure Results)



(Zero-Mean Correlation Measure Results)



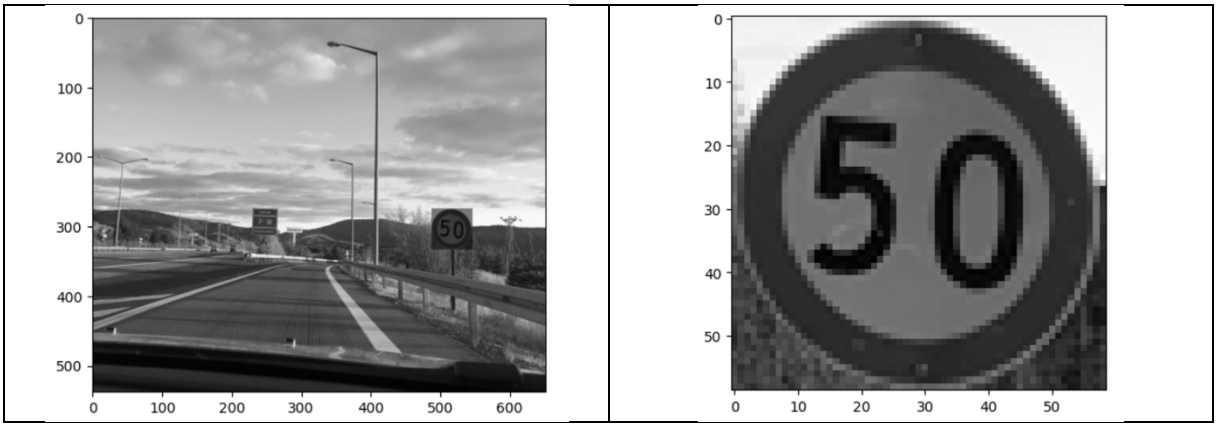
(Sum of Squared Difference Measure Results)



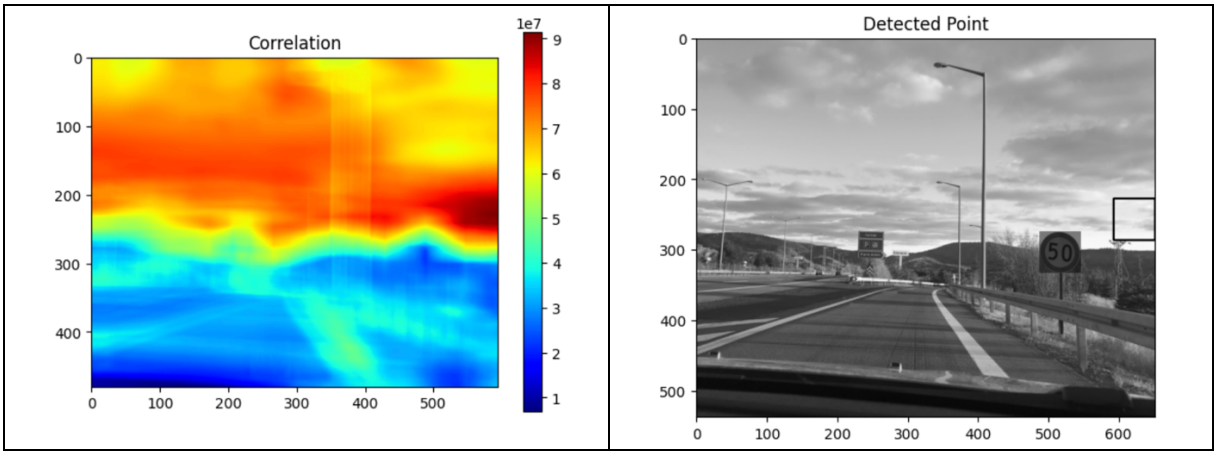
(Normalized Cross-Correlation Measure Results)

◊ *Part 1b*

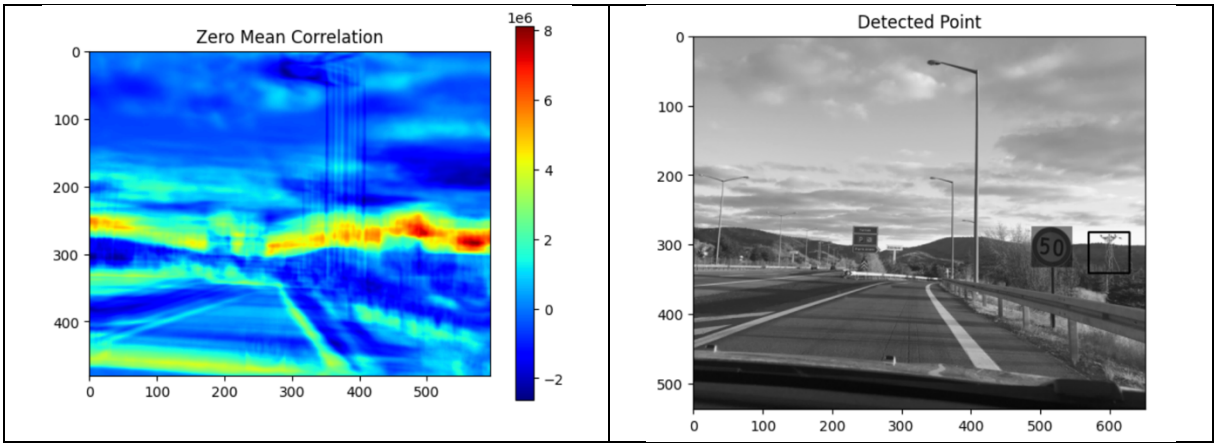
In this question *template image* is not in the main image. That's why all methods find it near to patch in the main image but cannot find the template. As you can see in the outputs below, the rectangles are so close to the patch. However, no one finds the correct place.



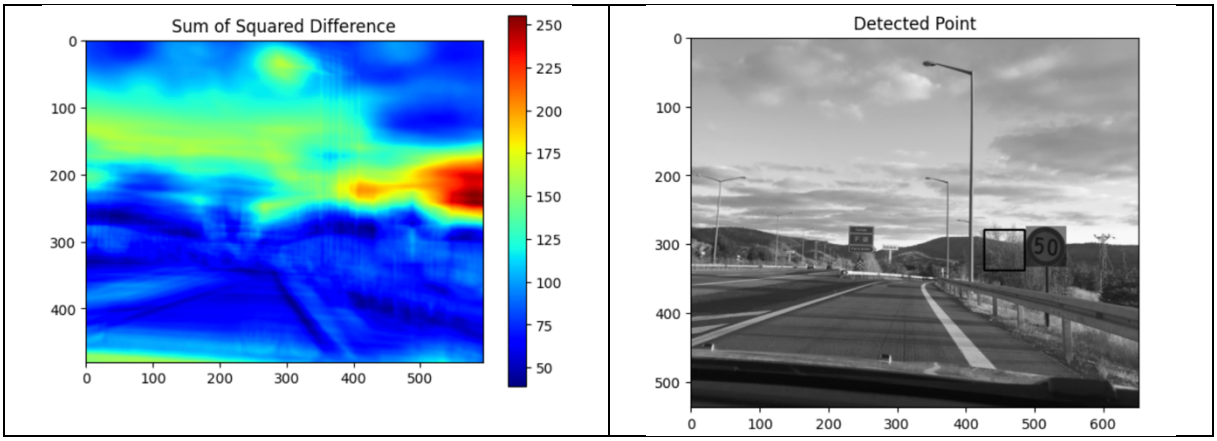
(Main and Template images of Part 1b)



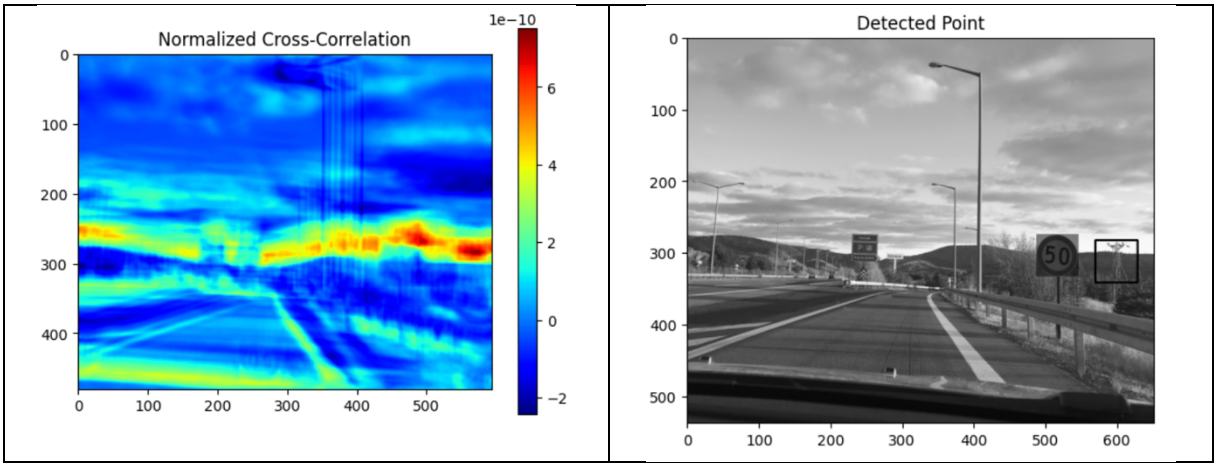
(Correlation Measure Results)



(Zero-Mean Correlation Measure Results)



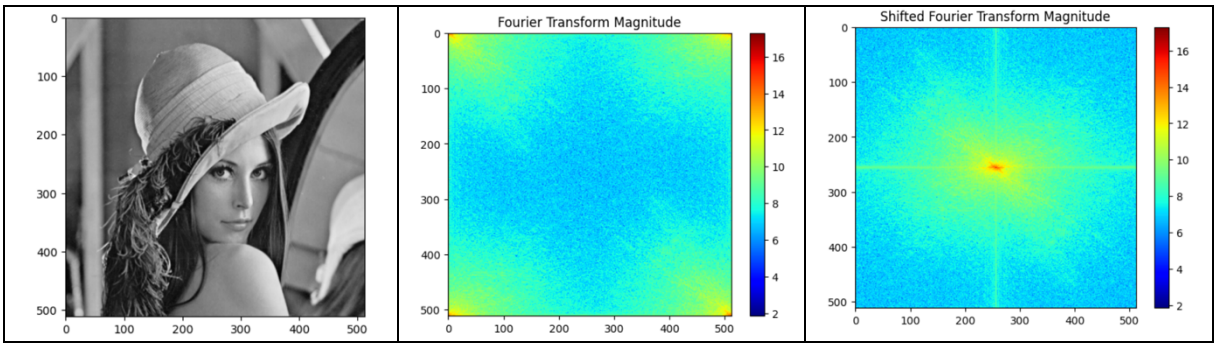
(Sum of Squared Difference Measure Results)



(Normalized Cross-Correlation Measure Results)

◊ *Part 2a*

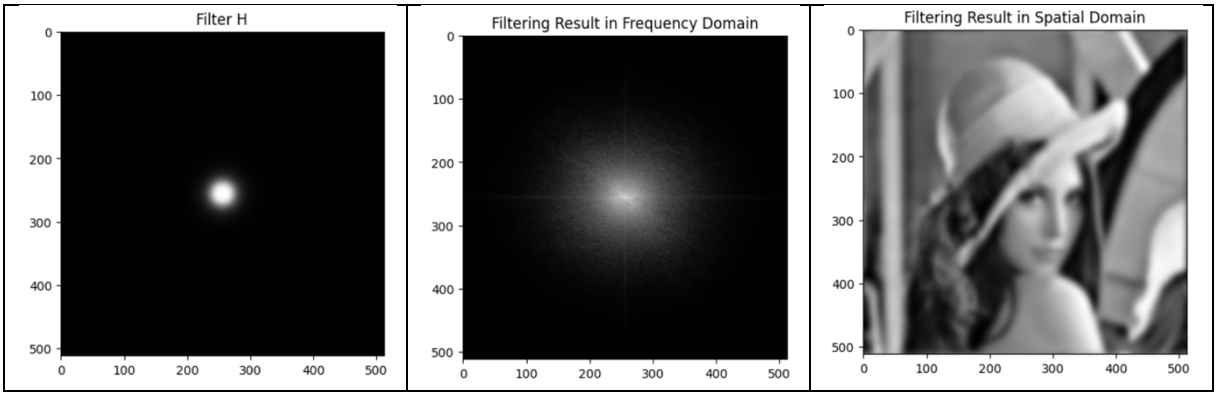
In Part 2, since the image used in a and b are the same, its Fourier transform magnitude and shifted Fourier transform always be like this:



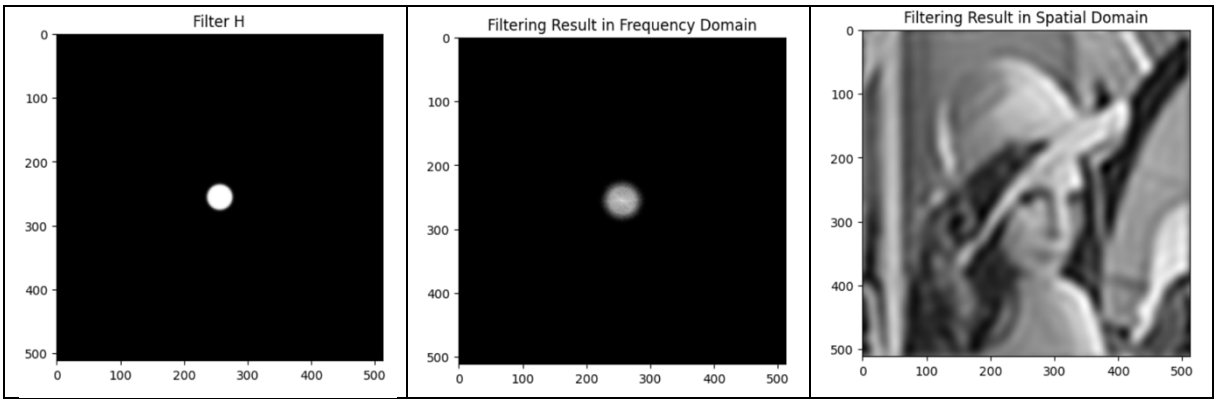
When we have a smaller value n, the filter H and filtering result in frequency domain have a smoother shift from white to black. However, if we increase the value of n, their shift from white to black gets smaller and sharper. Also, the one with smaller n value have smoother image surface and the one with bigger n value does not seems high quality if we compare the two filtering results in spatial domain.

If we check D_0, we can see that while D_0 is increasing, the radius of filter H gets bigger and so we have a closer output to the original image.

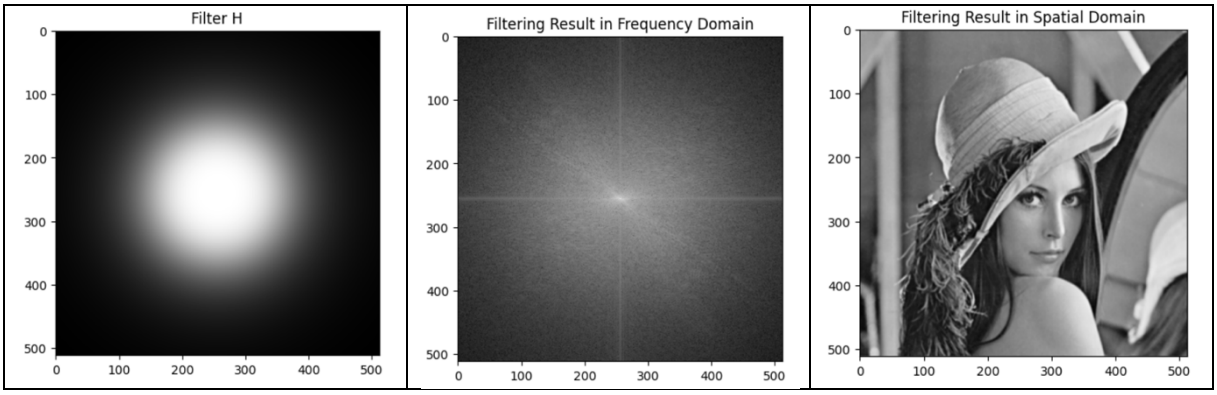
Finally, my outputs are:



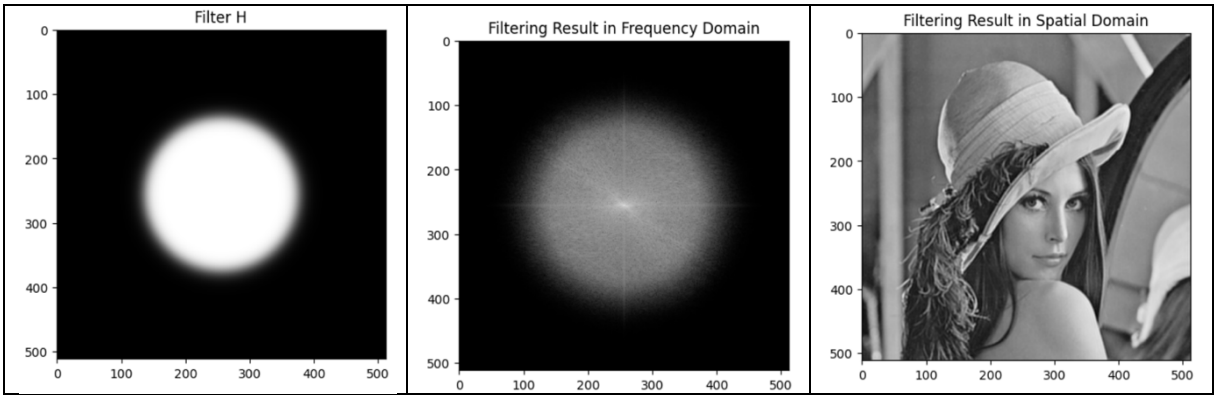
(D_0= 20, n =2)



(D_0= 20, n =10)



(D_0= 120, n =2)



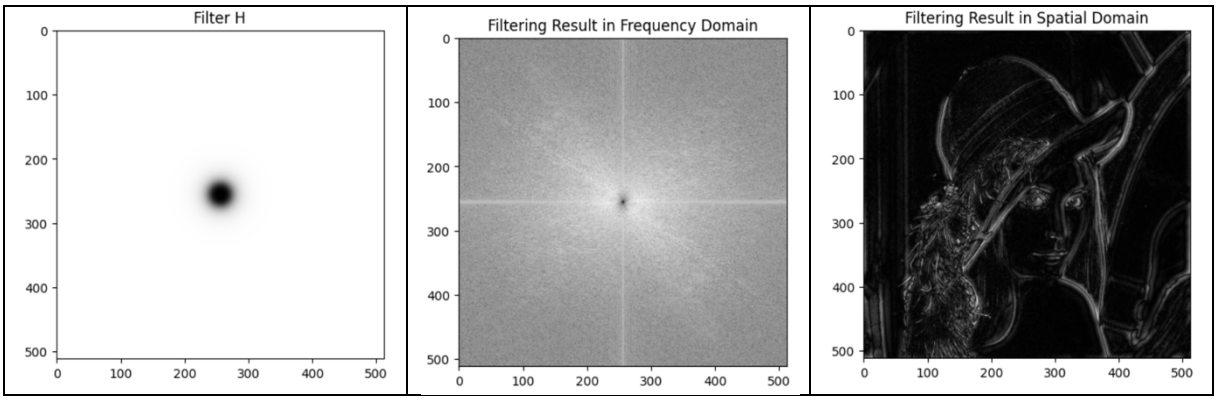
(D_0= 120, n =10)

◊ *Part 2b*

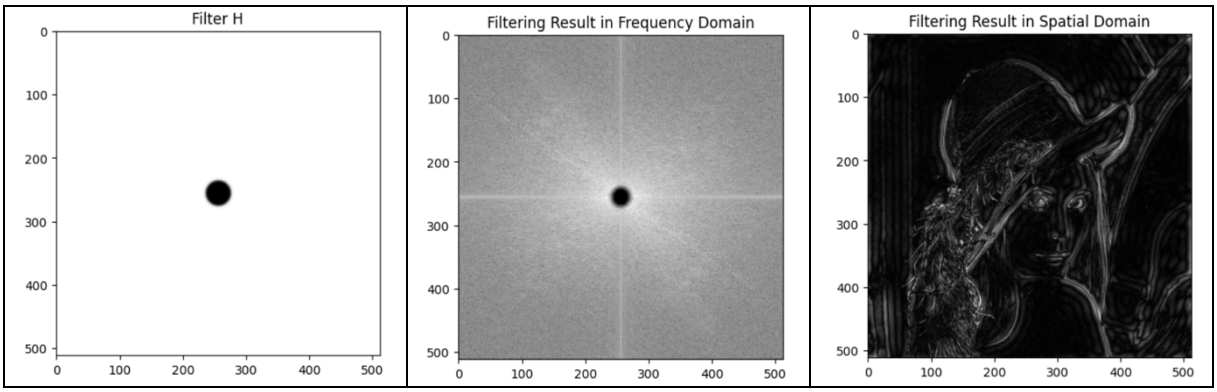
We mentioned about Butterworth High Pass Filter was the opposite process of Butterworth Low Pass Filter. And we can observe that the filter H's have opposite magnitude when their D_0 and n values are same.

The n values are still controls over the smoothness and quality of images. Moreover, D_0 has the same operation as in the Butterworth Low Pass Filter.

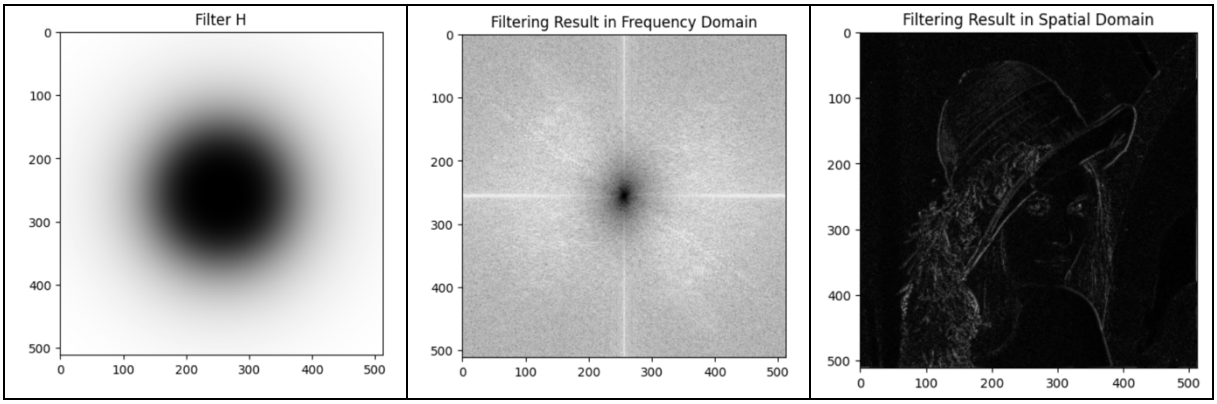
And my outputs are shown in the below:



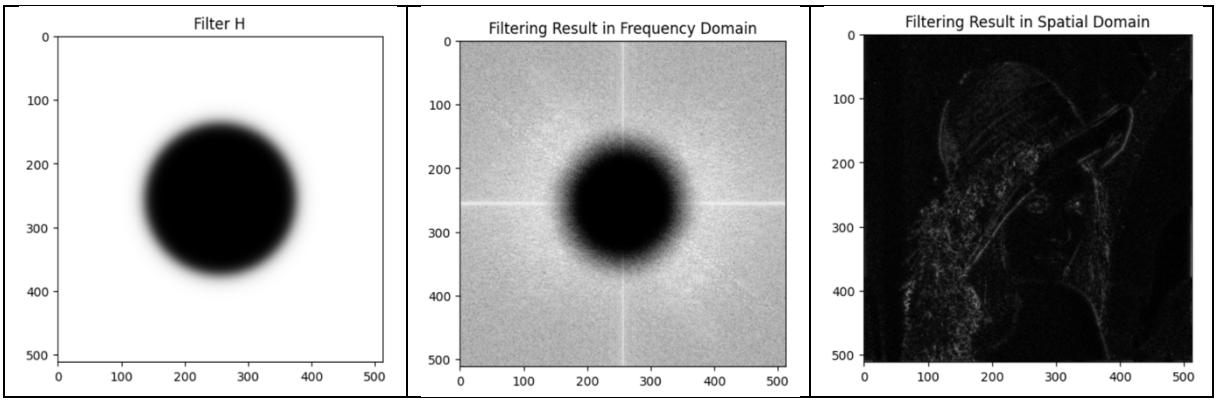
(D0= 20, n =2)



(D0= 20, n =10)



(D0= 120, n =2)



(D0= 120, n =10)

Conclusion

In my opinion, this project was challenging not because of its hardness but because of the limited sources. I had to search on the web but most of the sources were not clear enough or completely out of what I was looking for. So, I had a hard time. But also, the slides of the course helped me on some topics even though it is difficult to understand them. Apart from these, I think this project made me understand the concepts better. Plus, while I was writing the code it was somehow enjoyable and I am so happy that I could make this project finished. As a final remark, I hope I can use what I have learned in this project in the future.