CSE 573 Project 1

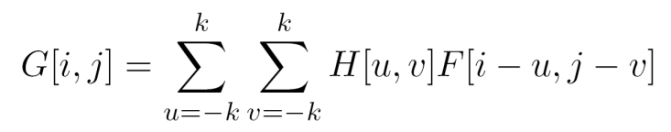
* NIKHIL SRIHARI

50291966

nikhilsr

**1. EDGE DETECTION:**

**Theory:** Our Image F exists as a 2D grayscale matrix. We use the convolution operator on this image matrix F, to find the edges in this image. The Convolution operator is mathematically defined as:

****

where, F is the input image, H is the convolution kernel, G is the output image, i and j are the co-ordinates of the current pixel, u and v are the co-ordinates of the kernel, and the kernel size is 2k+1( with the co-ordinates (0,0) being at the center of the kernel). Depending on which Kernel is being used, different effects are achieved on the image. For example, when the kernel is a box kernel, averaging of the current pixel with its neighbors is achieved, producing a blurred image. With a Gaussian kernel, this blurring effect is better and the weightage is more logically distributed – The weightage is highest at the central pixel and decreases as you go farther away from it.

To find edges, we need to find the points in the image where there is a significant change in the intensity values. We can do this by differentiating the Image(or the blurred/smoothed image) wrt to the x and y axis. But a convolution operation is associative. So, we have

d(F\*G)=d(F)\*G=F\*d(G).

The differentiation of the Guassian operation is called the Sobel operator. Hence, Ox = F\*Sx and Oy = F\*Sy give the edges in the image F wrt to the x and y axis respectively. These individual edges can then be combined to produce the final output image by,

O = ((Ox^2+Oy^2)^0.5)

I have developed 3 solutions for this task, each differing from the other in the preprocessing step. The brief explanation of these solutions is available below. I have, however due to space constraint, only attached the code and output images for the 3rd (best) solution here. The other codes are available in my code submission.

**Code 1:** This code implementation sees a simple straightforward application of Sobel Convolution on the input image to produce the edges. The image is taken and then immediately Sx and Sy are convoluted to produce the Ox and Oy images. These images are further normalized to yield more visually apparent images. And then, these 2 are merged together to give the final output.

**Code 2:** In this code implementation, I first convolve the Image F with a Gaussian kernel with sigma = 0.133. This results in a blurred image which is now convoled with Sx and Sy kernels to obtain Ox and Oy images. The rest is the same as Solution/Code 1 above.

**Code 3:** In this implementation, I first sharpen the input image F twice, by performing this operation: S = F\*I – F\*G, where S is the sharpened image, F is the input image, G is the Gaussian kernel and I is the scaled impulse kernel, twice. This sharpened image is now convolved with Sx and Sy to obtain Ox and Oy images. The rest is the same as Solution/Code 1 above. The code is below:

**from** cv2 **import** imread, imwrite

**import** numpy **as** np

**import** math **as** math

imageLocation = './task1.png'

**def** readImage(imageLocation):

img = imread(imageLocation,0)

**return** img

**def** writeImage(img, outputFileName):

imwrite('output/Task1\_Sol3\_'+outputFileName+'.jpg', img)

**return** 1

**def** getBoxIntensityDoublerKernel():

boxIntensityDoublerKernel = np.array(

[ [0,0,0],

[0,2,0],

[0,0,0] ]

)

**return** boxIntensityDoublerKernel

**def** getSobelX():

sobelX = np.array(

[ [1,0,-1],

[2,0,-2],

[1,0,-1] ]

)

normalizedSobelX = sobelX / 8

**return** normalizedSobelX

**def** getSobelY():

sobelY = np.array(

[ [-1,-2,-1],

[0,0,0],

[1,2,1] ]

)

normalizedSobelY = sobelY / 8

**return** normalizedSobelY

**def** getGaussianKernel(kernelSize, sigma=0.1):

**if** (kernelSize%2==0):

kernelSize=kernelSize+1

kernelSizeHalf = (int)(kernelSize/2)

l=-1\*kernelSizeHalf

h=kernelSizeHalf

kernel = np.zeros((kernelSize, kernelSize))

sum=0

**for** kernel\_i **in** np.arange( l, h+1 ):

**for** kernel\_j **in** np.arange( l, h+1 ):

kernel[kernel\_i+kernelSizeHalf][kernel\_j+kernelSizeHalf] = (1/( 2\*(math.pi)\*(sigma\*\*2)\*( (math.e)\*\*( ( ((kernel\_i/(2\*kernelSize))\*\*2)+((kernel\_j/(2\*kernelSize))\*\*2) )/( 2\*(sigma\*\*2) ) ) ) ))

sum = sum + kernel[kernel\_i+kernelSizeHalf][kernel\_j+kernelSizeHalf]

**for** kernel\_i **in** np.arange( l, h+1 ):

**for** kernel\_j **in** np.arange( l, h+1 ):

kernel[kernel\_i+kernelSizeHalf][kernel\_j+kernelSizeHalf] = (kernel[kernel\_i+kernelSizeHalf][kernel\_j+kernelSizeHalf]) / sum

**return** kernel

**def** convolve(img, kernel):

kernelSize = (int)((kernel.shape[0])/2)

convoluted\_img\_abs = img.copy()

convoluted\_img = np.zeros((convoluted\_img\_abs.shape[0], convoluted\_img\_abs.shape[1]))

**for** img\_i **in** np.arange( kernelSize, ((img.shape[0])-kernelSize) ):

**for** img\_j **in** np.arange( kernelSize, ((img.shape[1])-kernelSize) ):

temp = 0

**for** kernel\_i **in** np.arange(-1, 2):

**for** kernel\_j **in** np.arange(-1, 2):

kernel\_coordinate = kernel[kernel\_i+kernelSize, kernel\_j+kernelSize]

img\_coordinate = img[img\_i-kernel\_i, img\_j-kernel\_j]

temp = temp + (img\_coordinate\*kernel\_coordinate)

convoluted\_img\_abs[img\_i][img\_j] = abs(temp)

convoluted\_img[img\_i][img\_j] = temp

**return** convoluted\_img, convoluted\_img\_abs

**def** eliminateZeros\_method1(convoluted\_img):

outImg = np.zeros((convoluted\_img.shape[0], convoluted\_img.shape[1]))

minVal = (convoluted\_img[0,0])

maxVal = (convoluted\_img[0,0])

**for** convoluted\_img\_i **in** np.arange( 0, convoluted\_img.shape[0] ):

**for** convoluted\_img\_j **in** np.arange( 0, convoluted\_img.shape[1] ):

**if** (minVal>(convoluted\_img[convoluted\_img\_i, convoluted\_img\_j])):

minVal=(convoluted\_img[convoluted\_img\_i, convoluted\_img\_j])

**if** (maxVal<(convoluted\_img[convoluted\_img\_i, convoluted\_img\_j])):

maxVal=(convoluted\_img[convoluted\_img\_i, convoluted\_img\_j])

**for** convoluted\_img\_i **in** np.arange( 0, convoluted\_img.shape[0] ):

**for** convoluted\_img\_j **in** np.arange( 0, convoluted\_img.shape[1] ):

outImg[convoluted\_img\_i, convoluted\_img\_j] = 255.0 \* ( (convoluted\_img[convoluted\_img\_i, convoluted\_img\_j] - minVal) / (maxVal - minVal) )

**return** outImg

**def** eliminateZeros\_method2(convoluted\_img):

outImg = np.zeros((convoluted\_img.shape[0], convoluted\_img.shape[1]))

maxVal = (convoluted\_img[0,0])

**for** convoluted\_img\_i **in** np.arange( 0, convoluted\_img.shape[0] ):

**for** convoluted\_img\_j **in** np.arange( 0, convoluted\_img.shape[1] ):

**if** (maxVal<convoluted\_img[convoluted\_img\_i, convoluted\_img\_j]):

maxVal=convoluted\_img[convoluted\_img\_i, convoluted\_img\_j]

**for** convoluted\_img\_i **in** np.arange( 0, convoluted\_img.shape[0] ):

**for** convoluted\_img\_j **in** np.arange( 0, convoluted\_img.shape[1] ):

outImg[convoluted\_img\_i, convoluted\_img\_j] = 255.0 \* ( convoluted\_img[convoluted\_img\_i, convoluted\_img\_j] / maxVal )

**return** outImg

**def** mergeSobelXYValues(convoluted\_img\_sobelX, convoluted\_img\_sobelY):

convoluted\_img = np.zeros((convoluted\_img\_sobelX.shape[0], convoluted\_img\_sobelX.shape[1]))

**for** convoluted\_img\_i **in** np.arange( convoluted\_img.shape[0] ):

**for** convoluted\_img\_j **in** np.arange( convoluted\_img.shape[1] ):

convoluted\_img[convoluted\_img\_i, convoluted\_img\_j] = math.sqrt( ((convoluted\_img\_sobelX[convoluted\_img\_i, convoluted\_img\_j])\*\*2) + ((convoluted\_img\_sobelY[convoluted\_img\_i, convoluted\_img\_j])\*\*2) )

**return** convoluted\_img

**def** main():

# Reading and displaying the Input Image

img = readImage(imageLocation)

# Scaled Intensity Kernel

boxIntensityDoublerKernel = getBoxIntensityDoublerKernel()

convoluted\_img\_boxIntensityDoublerKernel, convoluted\_img\_boxIntensityDoublerKernel\_abs = convolve(img, boxIntensityDoublerKernel)

# Blur the image first

gaussianKernel = getGaussianKernel(5, 0.13301)

convoluted\_img\_gaussianKernel, convoluted\_img\_gaussianKernel\_abs = convolve(img, gaussianKernel)

# Find the difference between Scaled Intensity Kernel convolution of the Image and Blurred image

img = np.zeros((convoluted\_img\_boxIntensityDoublerKernel.shape[0], convoluted\_img\_boxIntensityDoublerKernel.shape[1]))

**for** i **in** range(0, convoluted\_img\_boxIntensityDoublerKernel.shape[0]):

**for** j **in** range(0, convoluted\_img\_boxIntensityDoublerKernel.shape[1]):

img[i][j]=convoluted\_img\_boxIntensityDoublerKernel[i][j]-convoluted\_img\_gaussianKernel[i][j]

writeImage(img, 'img')

# Box Intensity Double Kernel of Image

boxIntensityDoublerKernel = getBoxIntensityDoublerKernel()

convoluted\_img\_boxIntensityDoublerKernel, convoluted\_img\_boxIntensityDoublerKernel\_abs = convolve(img, boxIntensityDoublerKernel)

# Blur the image first

gaussianKernel = getGaussianKernel(5, 0.13301)

convoluted\_img\_gaussianKernel, convoluted\_img\_gaussianKernel\_abs = convolve(img, gaussianKernel)

# Find the difference between Box Intensity Double Kernel of Image and Blurring the image first

img = np.zeros((convoluted\_img\_boxIntensityDoublerKernel.shape[0], convoluted\_img\_boxIntensityDoublerKernel.shape[1]))

**for** i **in** range(0, convoluted\_img\_boxIntensityDoublerKernel.shape[0]):

**for** j **in** range(0, convoluted\_img\_boxIntensityDoublerKernel.shape[1]):

img[i][j]=convoluted\_img\_boxIntensityDoublerKernel[i][j]-convoluted\_img\_gaussianKernel[i][j]

writeImage(img, 'img')

# Starting Edge Detection using Sobel X Operator

sobelX = getSobelX()

convoluted\_img\_sobelX, convoluted\_img\_sobelX\_abs = convolve(img, sobelX)

#No point in writing convoluted\_img\_sobelX, as it has negative values, which cant be written by imwrite. It just messes everything up.

#So i have taken a snapshot of this as it runs

writeImage(convoluted\_img\_sobelX\_abs, 'convoluted\_img\_sobelX\_abs')

convoluted\_img\_sobelX\_elZerosM1 = eliminateZeros\_method1(convoluted\_img\_sobelX\_abs)

writeImage(convoluted\_img\_sobelX\_elZerosM1, 'convoluted\_img\_sobelX\_elZerosM1')

convoluted\_img\_sobelX\_elZerosM2 = eliminateZeros\_method2(convoluted\_img\_sobelX\_abs)

writeImage(convoluted\_img\_sobelX\_elZerosM2, 'convoluted\_img\_sobelX\_elZerosM2')

# Starting Edge Detection using Sobel Y Operator

sobelY = getSobelY()

convoluted\_img\_sobelY, convoluted\_img\_sobelY\_abs = convolve(img, sobelY)

#No point in writing convoluted\_img\_sobelY, as it has negative values, which cant be written by imwrite. It just messes everything up.

#So i have taken a snapshot of this as it runs

writeImage(convoluted\_img\_sobelY\_abs, 'convoluted\_img\_sobelY\_abs')

convoluted\_img\_sobelY\_elZerosM1 = eliminateZeros\_method1(convoluted\_img\_sobelY\_abs)

writeImage(convoluted\_img\_sobelY\_elZerosM1, 'convoluted\_img\_sobelY\_elZerosM1')

convoluted\_img\_sobelY\_elZerosM2 = eliminateZeros\_method2(convoluted\_img\_sobelY\_abs)

writeImage(convoluted\_img\_sobelY\_elZerosM2, 'convoluted\_img\_sobelY\_elZerosM2')

# Constructing final image by merging Sobel X and Sobel Y's output images

final\_convoluted\_img = mergeSobelXYValues(convoluted\_img\_sobelX\_abs, convoluted\_img\_sobelY\_abs)

writeImage(final\_convoluted\_img, 'final\_convoluted\_img')

main()

**Output Images:**

Fig 1.1. F \*Sx Fig 1.2. F\*Sx with method 1 Zeros Elimination

Fig 1.3. F\*Sx with method 2 Zeros Elimination Fig 1.4. F \* Sy

Fig 1.5. F\*Sy with method 1 Zeros Elimination Fig 1.6. F\*Sy with method 2 Zeros Elimination



Fig 1.7. Final edge detection Image obtained by merging F\*Sx and F\*Sy’s magnitude.

**2. Keypoint Detection:**

**Theory:** This question deals with keypoint detection. Simple keypoint detection just needs one to take the Laplacian of an image. It is immune to rotation and intensity and view point. But this doesn’t take into account the scale variation. Hence, a more elegant solution to extract (and later match) keypoints is to use SIFT. SIFT stands for Scale Invariant Feature Transform. Here, we only partially implement SIFT. The 1st step involves creating a scale space – It consists of a series of Gaussian blurred image. And then the original image is resized and Gaussian blurred again and so on and son. Each of these blurred image sets are called Octaves. Once the scale space is create, we now need to the Laplacian of Gaussian(LoG). There, however, exists an alternative which is computationally much cheaper. It is the difference of Gaussian(DoG). So the 2nd step involves calculating the DoG space in each Octave. The 3rd step involved comparing these DoG space images to find their local maxima and minima. These local minimas and maxima gives an indication about the location of keypoints. This is the extend to which we have implemented here

**Code:** As mentioned above, first we create the Scale Space. This is a 4 X 5 list (4 Octaves, 5 Images per Octave). Then we create the DoG Space using this scale space. Its 4 X 4. Finally we find the local minima and maxima which act as our keypoints. This is 4 X 2. Finally, so, we have 8 images.

**from** cv2 **import** imread, imwrite

**from** math **import** sqrt, pi **as** CONSTANT\_PI, e **as** CONSTANT\_E

**import** numpy **as** np

**from** cv2 **import** imshow, waitKey, destroyAllWindows

imageLocation = './task2.jpg'

gaussianSigma\_startingValue = (1/(sqrt(2)))

gaussianSigma\_WithinOctaveRatio = sqrt(2)

gaussianSigma\_AcrossOctaveRatio = 2

numberOfOctaves = 4

imagesPerOctave = 5

gaussianKernelSize = 7

**def** readImage(imageLocation):

img = imread(imageLocation,0)

**return** img

**def** writeImage(img, outputFileName):

imwrite('output/Task2\_Sol1\_'+outputFileName+'.jpg', img)

**return** 1

**def** displayImage(img, imgTitle='No Title'):

imshow(imgTitle, img)

waitKey(0)

destroyAllWindows()

**def** getGaussianKernel(kernelSize, sigma):

**if** (kernelSize%2==0):

kernelSize=kernelSize+1

kernelSizeHalf = (int)(kernelSize/2)

l=-1\*kernelSizeHalf

h=kernelSizeHalf

kernel = np.zeros((kernelSize, kernelSize))

sum=0

**for** kernel\_i **in** np.arange( l, h+1 ):

**for** kernel\_j **in** np.arange( l, h+1 ):

kernel[kernel\_i+kernelSizeHalf][kernel\_j+kernelSizeHalf] = (1/( 2\*(CONSTANT\_PI)\*(sigma\*\*2)\*( (CONSTANT\_E)\*\*( ( ((kernel\_i)\*\*2)+((kernel\_j)\*\*2) )/( 2\*(sigma\*\*2) ) ) ) ))

sum = sum + kernel[kernel\_i+kernelSizeHalf][kernel\_j+kernelSizeHalf]

**for** kernel\_i **in** np.arange( l, h+1 ):

**for** kernel\_j **in** np.arange( l, h+1 ):

kernel[kernel\_i+kernelSizeHalf][kernel\_j+kernelSizeHalf] = (kernel[kernel\_i+kernelSizeHalf][kernel\_j+kernelSizeHalf]) / sum

**return** kernel

**def** convolve(img, kernel):

kernelSize = (int)((kernel.shape[0])/2)

convoluted\_img\_abs = img.copy()

convoluted\_img = np.zeros((convoluted\_img\_abs.shape[0], convoluted\_img\_abs.shape[1]))

**for** img\_i **in** np.arange( kernelSize, ((img.shape[0])-kernelSize) ):

**for** img\_j **in** np.arange( kernelSize, ((img.shape[1])-kernelSize) ):

temp = 0

**for** kernel\_i **in** np.arange(-1, 2):

**for** kernel\_j **in** np.arange(-1, 2):

kernel\_coordinate = kernel[kernel\_i+kernelSize, kernel\_j+kernelSize]

img\_coordinate = img[img\_i-kernel\_i, img\_j-kernel\_j]

temp = temp + (img\_coordinate\*kernel\_coordinate)

convoluted\_img\_abs[img\_i][img\_j] = abs(temp)

convoluted\_img[img\_i][img\_j] = temp

**return** convoluted\_img, convoluted\_img\_abs

**def** resize(img, factor=0.5):

resized\_img = np.zeros(( (int)((img.shape[0])\*factor) , (int)((img.shape[1])\*factor) ))

**for** resized\_img\_i **in** range(0, resized\_img.shape[0]):

**for** resized\_img\_j **in** range(0, resized\_img.shape[1]):

a = (int)((1/factor)\*resized\_img\_i)

b = (int)((1/factor)\*resized\_img\_j)

resized\_img[resized\_img\_i][resized\_img\_j] = (int)(img[a][b])

**return** resized\_img

**def** main():

original\_img = readImage(imageLocation)

# Step 1 - Creating Scale Space images

scaleSpace = []

**for** i **in** range(0, numberOfOctaves):

temp=[]

**for** j **in** range(0, imagesPerOctave):

temp.append(None)

scaleSpace.append(temp)

octaveSigma\_startingValue = gaussianSigma\_startingValue / gaussianSigma\_AcrossOctaveRatio

**for** octaveIndex **in** range(0, numberOfOctaves):

octaveSigma\_startingValue = octaveSigma\_startingValue \* gaussianSigma\_AcrossOctaveRatio

sigma = octaveSigma\_startingValue / gaussianSigma\_WithinOctaveRatio

**for** imageIndex **in** range(0, imagesPerOctave):

**if** (imageIndex==0):

**if** (octaveIndex==0):

scaleSpace[0][0]=original\_img

**else**:

scaleSpace[octaveIndex][0] = resize(scaleSpace[octaveIndex-1][0], factor=0.5)

**else**:

sigma = sigma \* gaussianSigma\_WithinOctaveRatio

gaussianKernel = getGaussianKernel(gaussianKernelSize, sigma)

convoluted\_img, convoluted\_img\_abs = convolve(scaleSpace[octaveIndex][imageIndex-1], gaussianKernel)

scaleSpace[octaveIndex][imageIndex] = convoluted\_img

**for** octaveIndex **in** range(0, numberOfOctaves):

**for** imageIndex **in** range(0, imagesPerOctave):

writeImage(scaleSpace[octaveIndex][imageIndex], "ScaleSpace\_"+str(octaveIndex)+"\_"+str(imageIndex))

**print**("Scale Space Created")

# Step 2 - Creating DOG Space images

dogSpace = []

**for** i **in** range(0, numberOfOctaves):

temp=[]

**for** j **in** range(0, imagesPerOctave-1):

temp.append(None)

dogSpace.append(temp)

**for** octaveIndex **in** range(0, len(dogSpace)):

**for** imageIndex **in** range(0, len(dogSpace[0])):

currentDogSpace\_l = (scaleSpace[octaveIndex][imageIndex]).shape[0]

currentDogSpace\_b = (scaleSpace[octaveIndex][imageIndex]).shape[1]

dogSpace[octaveIndex][imageIndex] = np.zeros(( currentDogSpace\_l , currentDogSpace\_b ))

**for** i **in** range(0, currentDogSpace\_l):

**for** j **in** range(0, currentDogSpace\_b ):

diff = scaleSpace[octaveIndex][imageIndex+1][i][j] - scaleSpace[octaveIndex][imageIndex][i][j]

**if** (diff>=0):

dogSpace[octaveIndex][imageIndex][i][j] = diff

**else**:

dogSpace[octaveIndex][imageIndex][i][j] = diff + 256

**for** octaveIndex **in** range(0, len(dogSpace)):

**for** imageIndex **in** range(0, len(dogSpace[0])):

writeImage(dogSpace[octaveIndex][imageIndex], "DOGSpace\_"+str(octaveIndex)+"\_"+str(imageIndex))

**print**("DOG Space Created")

# Step 3 - Identifying maxima/minima in DOG images

maxMinDogSpace = []

**for** i **in** range(0, numberOfOctaves):

temp=[]

**for** j **in** range(0, imagesPerOctave-3):

temp.append(None)

maxMinDogSpace.append(temp)

**for** octaveIndex **in** range(0, len(maxMinDogSpace)):

**for** imageIndex **in** range(0, len(maxMinDogSpace[0])):

maxMinDogSpace[octaveIndex][imageIndex] = np.zeros(( (dogSpace[octaveIndex][0]).shape[0] , (dogSpace[octaveIndex][0]).shape[1] ))

prevImage = dogSpace[octaveIndex][imageIndex]

currentImage = dogSpace[octaveIndex][imageIndex+1]

nextImage = dogSpace[octaveIndex][imageIndex+2]

**for** i **in** range(1, (currentImage.shape[0])-1):

**for** j **in** range(1, (currentImage.shape[1])-1):

currentPixelVal = currentImage[i][j]

compArray = []

k = -1

l = -1

**while**(k<2):

**while**(l<2):

compArray.append(prevImage[i+k][j+l])

compArray.append(currentImage[i+k][j+l])

compArray.append(nextImage[i+k][j+l])

l=l+1

k=k+1

minComp = min(compArray)

maxComp = max(compArray)

**if** (minComp!=0 **and** maxComp!=0 **and** (minComp==currentPixelVal **or** maxComp==currentPixelVal)):

maxMinDogSpace[octaveIndex][imageIndex][i][j] = 255

**for** octaveIndex **in** range(0, len(maxMinDogSpace)):

**for** imageIndex **in** range(0, len(maxMinDogSpace[0])):

writeImage(maxMinDogSpace[octaveIndex][imageIndex], "maxMinDogSpace\_"+str(octaveIndex)+"\_"+str(imageIndex))

**print**("Max and Min values in DOG Space Found")

main()

My Scale Space and DoG images are good. But my final Keypoints from Local Min and Maxima are not accurate for the 3rd and 4th Octave weren’t correct. Anyways here are some Output Images:

**Output Images: (Solutions to 1,2,3(partial))**

**1.**

 Scale Space, Octave 2, Image 1 (375X229)

 Scale Space, Octave 2, Image 2 (375X229)

 Scale Space, Octave 2, Image 3 (375X229)

 Scale Space, Octave 3, Image 1 (187X114)

 Scale Space, Octave 3, Image 2 (187X114)

 Scale Space, Octave 3, Image 3 (187X114)

2.

 DoG Space,Octave 2,Image 1(375X229)

 DoG Space,Octave 2,Image 2(375X229)

 DoG Space,Octave 2,Image 3(375X229)

 DoG Space,Octave 2,Image 4(375X229)

 DoG Space, Octave 3, Image 1 (187X114)

 DoG Space, Octave 3, Image 2 (187X114)

 DoG Space, Octave 3, Image 3 (187X114)



Min and Max Dog Space Octave 1, Image 2

**3A Cursor Detection:**

**Theory:** This project is about template matching. We have been given a template image and we need to find this template in each image. We take the test image and template image and extract the key points from it. We then take this new template image with key points and try to match it with the test image’s key points. This can be done using a variety of methods like SSD, NCC etc. Also, keypoint extraction has been detailed above.

**Code:** This project allows us to use external libraries like OpenCV and numpy to their fullest extent. So – First I read the template image and take its Laplacian. Then I smooth it with a Gaussian convolution. This is now my Template image from now on. I do the same procedure with my test image. Laplacian, Gaussian Convolution and then Image sharpening. The resultant image is my Test Image from now on. Now, I compare these 2 images using the openCV function matchTemplate with the method set as TM\_CCOEFF (I tried the other methods, but this method yielded the best results). Since we use TM\_CCOEFF we have to now find the max values in the output image from matchTemplate. (Ideally, this max value should be above a threshold. I set this threshold to 400000 by trial and error.) It is at this location, that our template exists. So, now we just draw a rectangle around our template to highlight it.

**import** numpy **as** np

**import** argparse, glob

**from** cv2 **import** imread, rectangle, imshow, imwrite, waitKey, destroyAllWindows, resize, Laplacian, GaussianBlur, filter2D, matchTemplate, minMaxLoc

**from** cv2 **import** CV\_8U **as** CONST\_CV\_8U, TM\_CCOEFF **as** CONST\_TM\_CCOEFF, TM\_CCORR **as** CONST\_TM\_CCORR

**from** scipy.ndimage.filters **import** gaussian\_filter

imageLocation\_template = './task3/template.png'

imageLocation\_positiveImages\_prefix = './task3/pos\_'

imageLocation\_positiveImages\_suffix = '.jpg'

imageLocation\_negativeImages\_prefix = './task3/neg\_'

imageLocation\_negativeImages\_suffix = '.jpg'

numberOfPositiveImages = 15

numberOfNegativeImages = 10

**def** readImage(imageLocation):

img = imread(imageLocation,0)

**return** img

**def** writeImage(img, outputFileName):

imwrite('output/Task3\_Sol1\_'+outputFileName+'.jpg', img)

**return** 1

**def** displayImage(img, imgTitle='No Title'):

imshow(imgTitle, img)

waitKey(0)

destroyAllWindows()

**def** getSharpeningKernel():

sharpeningKernel = np.array(

[ [-1,-1,-1],

[-1, 9,-1],

[-1,-1,-1] ]

)

**return** sharpeningKernel

**def** main():

**print**("Starting processing")

img\_template = readImage(imageLocation\_template)

img\_template = resize(img\_template , (13,18))

img\_template = Laplacian(img\_template, CONST\_CV\_8U)

img\_template = GaussianBlur(img\_template, (3,3),0)

sharpeningKernel = getSharpeningKernel()

**for** i **in** range(1, numberOfPositiveImages+1):

imageLocation = imageLocation\_positiveImages\_prefix+str(i)+imageLocation\_positiveImages\_suffix

**print**("Image at '"+imageLocation+"'")

img\_test\_original = readImage(imageLocation)

img\_test = img\_test\_original.copy()

img\_test = Laplacian(img\_test, CONST\_CV\_8U)

img\_test = GaussianBlur(img\_test, (3,3), 0)

img\_test = filter2D(img\_test, -1, sharpeningKernel)

result = matchTemplate(img\_test, img\_template, CONST\_TM\_CCOEFF)

min\_val, max\_val, min\_loc, max\_loc = minMaxLoc(result)

**if** (max\_val>400000):

**print**(" Template Found")

img\_test\_original = np.dstack([img\_test\_original, img\_test\_original, img\_test\_original])

rectangle(img\_test\_original, (max\_loc[0], max\_loc[1]), (max\_loc[0]+img\_template.shape[1], max\_loc[1]+img\_template.shape[0]), (0, 255, 255), 2)

**else**:

**print**(" Template Not Found")

writeImage(img\_test\_original, "Pos"+str(i))

**print**("Positive images processed")

**for** i **in** range(1, numberOfNegativeImages+1):

**if** (i==7):

**continue**

imageLocation = imageLocation\_negativeImages\_prefix+str(i)+imageLocation\_negativeImages\_suffix

**print**("Image at '"+imageLocation+"'")

img\_test\_original = readImage(imageLocation)

img\_test = img\_test\_original.copy()

img\_test = Laplacian(img\_test, CONST\_CV\_8U)

img\_test = GaussianBlur(img\_test, (3,3), 0)

img\_test = filter2D(img\_test, -1, sharpeningKernel)

result = matchTemplate(img\_test, img\_template, CONST\_TM\_CCOEFF)

min\_val, max\_val, min\_loc, max\_loc = minMaxLoc(result)

**if** (max\_val>400000):

**print**(" Template Found")

img\_test\_original = np.dstack([img\_test\_original, img\_test\_original, img\_test\_original])

rectangle(img\_test\_original, (max\_loc[0], max\_loc[1]), (max\_loc[0]+img\_template.shape[1], max\_loc[1]+img\_template.shape[0]), (0, 255, 255), 2)

**else**:

**print**(" Template Not Found")

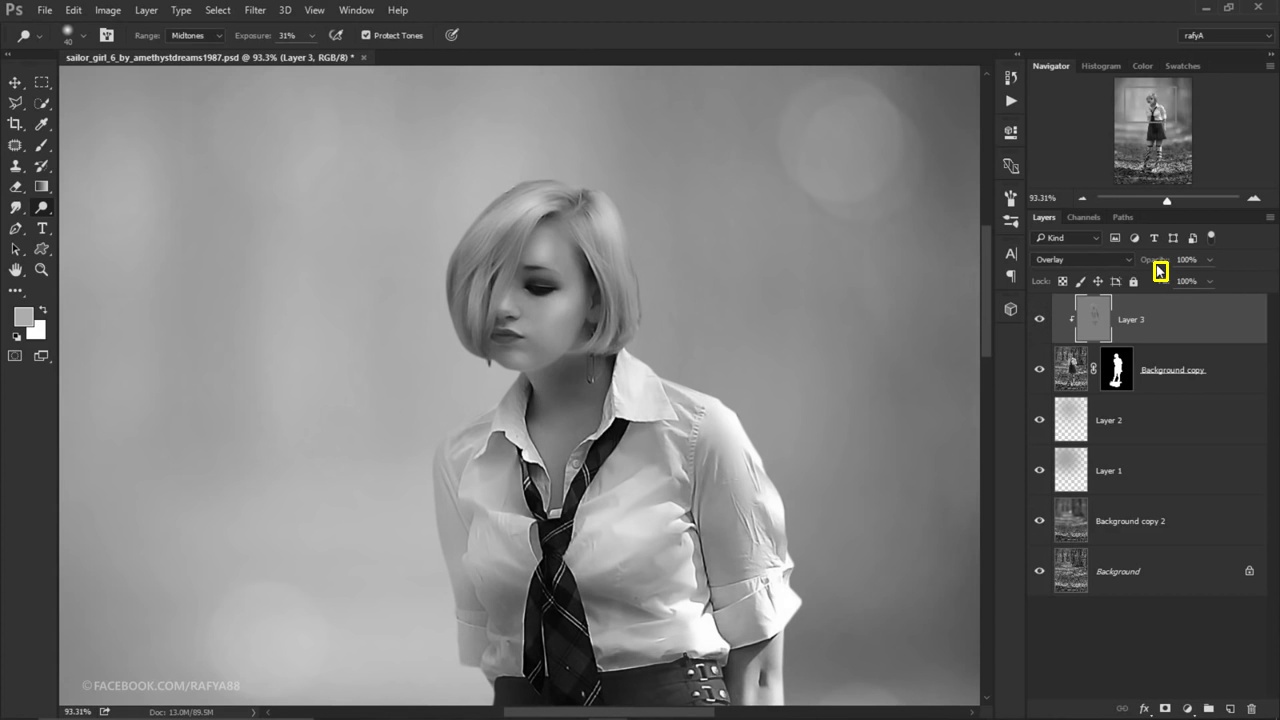
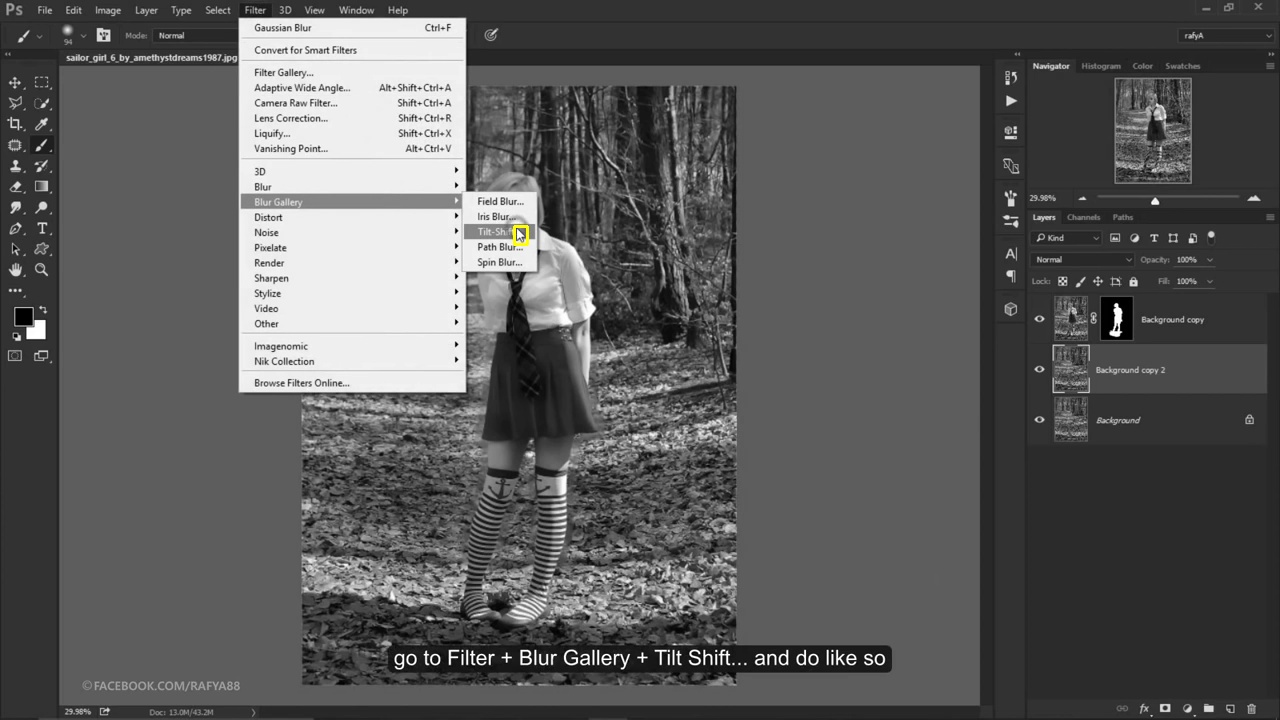
writeImage(img\_test\_original, "Neg"+str(i))

**print**("Negative images processed")

main()

My code has an overall accuracy of **80%**. The image, expected output and my code’s output are tabulated below:

|  |  |  |
| --- | --- | --- |
| **Image Name** | **Expected Output** | **Actual Output** |
| pos\_1.jpg | Found | Found Correctly |
| pos\_2.jpg | Found | Found Correctly |
| pos\_3.jpg | Found | Found Incorrectly |
| pos\_4.jpg | Found | Found Correctly |
| pos\_5.jpg | Found | Found Incorrectly |
| pos\_6.jpg | Found | Found Correctly |
| pos\_7.jpg | Found | Found Correctly |
| pos\_8.jpg | Found | Not Found |
| pos\_9.jpg | Found | Found Correctly |
| pos\_10.jpg | Found | Found Correctly |
| pos\_11.jpg | Found | Found Correctly |
| pos\_12.jpg | Found | Found Incorrectly |
| pos\_13.jpg | Found | Found Correctly |
| pos\_14.jpg | Found | Found Correctly |
| pos\_15.jpg | Found | Found Correctly |
| neg\_1.jpg | Not Found | Not Found |
| neg\_2.jpg | Not Found | Not Found |
| neg\_3.jpg | Not Found | Not Found |
| neg\_4.jpg | Not Found | Found Incorrectly |
| neg\_5.jpg | Not Found | Not Found |
| neg\_6.jpg | Not Found | Not Found |
| neg\_7.jpg | Not Found | Not Found |
| neg\_8.jpg | Not Found | Not Found |
| neg\_9.jpg | Not Found | Not Found |
| neg\_10.jpg | Not Found | Not Found |

**Some Sample Output Images:**

**3B Curson Detection – Bonus Package:**

**Theory:** Same as above but repeated for multiple templates

**Code:** Logic is same as above, just repeated for multiple templates. Also, here we pay closer attention to size of the template as its size is not consistent across the different test images. So, there is some custom logic to handle template image sizes.

**import** numpy **as** np

**import** argparse, glob

**from** cv2 **import** imread, rectangle, imshow, imwrite, waitKey, destroyAllWindows, resize, Laplacian, GaussianBlur, filter2D, matchTemplate, minMaxLoc

**from** cv2 **import** CV\_8U **as** CONST\_CV\_8U, TM\_CCOEFF **as** CONST\_TM\_CCOEFF, TM\_CCORR **as** CONST\_TM\_CCORR

**from** scipy.ndimage.filters **import** gaussian\_filter

imageLocation\_template1 = './task3\_bonus/template1.jpg'

imageLocation\_positiveImages1\_prefix = './task3\_bonus/t1\_'

imageLocation\_positiveImages1\_suffix = '.jpg'

imageLocation\_template2 = './task3\_bonus/template2.jpg'

imageLocation\_positiveImages2\_prefix = './task3\_bonus/t2\_'

imageLocation\_positiveImages2\_suffix = '.jpg'

imageLocation\_template3 = './task3\_bonus/template3.jpg'

imageLocation\_positiveImages3\_prefix = './task3\_bonus/t3\_'

imageLocation\_positiveImages3\_suffix = '.jpg'

imageLocation\_negativeImages\_prefix = './task3\_bonus/neg\_'

imageLocation\_negativeImages\_suffix = '.jpg'

template1\_sizeArray = [30, 20, 25, 20, 30, 22]

template3\_sizeArray = [30, 25, 30, 30, 30, 30]

numberOfPositive1Images = 6

numberOfPositive2Images = 6

numberOfPositive3Images = 6

numberOfNegativeImages = 12

**def** readImage(imageLocation):

img = imread(imageLocation,0)

**return** img

**def** writeImage(img, outputFileName):

imwrite('output/Task3Bonus\_'+outputFileName+'.jpg', img)

**return** 1

**def** displayImage(img, imgTitle='No Title'):

imshow(imgTitle, img)

waitKey(0)

destroyAllWindows()

**def** getSharpeningKernel():

sharpeningKernel = np.array(

[ [-1,-1,-1],

[-1, 9,-1],

[-1,-1,-1] ]

)

**return** sharpeningKernel

**def** main():

**print**("Starting Template 1 processing")

sharpeningKernel = getSharpeningKernel()

**for** i **in** range(1, numberOfPositive1Images+1):

img\_template = readImage(imageLocation\_template1)

**if** (len(template1\_sizeArray)>=numberOfPositive1Images):

img\_template = resize(img\_template , (template1\_sizeArray[i-1], template1\_sizeArray[i-1]))

**else**:

img\_template = resize(img\_template , (20, 20))

img\_template = Laplacian(img\_template, CONST\_CV\_8U)

img\_template = GaussianBlur(img\_template, (3,3),0)

imageLocation = imageLocation\_positiveImages1\_prefix+str(i)+imageLocation\_positiveImages1\_suffix

**print**("Image at '"+imageLocation+"'")

img\_test\_original = readImage(imageLocation)

img\_test = img\_test\_original.copy()

img\_test = Laplacian(img\_test, CONST\_CV\_8U)

img\_test = GaussianBlur(img\_test, (3,3), 0)

img\_test = filter2D(img\_test, -1, sharpeningKernel)

result = matchTemplate(img\_test, img\_template, CONST\_TM\_CCOEFF)

min\_val, max\_val, min\_loc, max\_loc = minMaxLoc(result)

**print**(max\_val)

**if** (max\_val>400000):

**print**(" Template Found")

img\_test\_original = np.dstack([img\_test\_original, img\_test\_original, img\_test\_original])

rectangle(img\_test\_original, (max\_loc[0], max\_loc[1]), (max\_loc[0]+img\_template.shape[1], max\_loc[1]+img\_template.shape[0]), (255, 0, 0), 2)

**else**:

**print**(" Template Not Found")

writeImage(img\_test\_original, "t1\_"+str(i))

**print**("Template 1 positive images processed")

**print**("Starting Template 2 processing")

sharpeningKernel = getSharpeningKernel()

**for** i **in** range(1, numberOfPositive2Images+1):

img\_template = readImage(imageLocation\_template2)

img\_template = resize(img\_template , (30,30))

img\_template = Laplacian(img\_template, CONST\_CV\_8U)

img\_template = GaussianBlur(img\_template, (3,3),0)

imageLocation = imageLocation\_positiveImages2\_prefix+str(i)+imageLocation\_positiveImages2\_suffix

**print**("Image at '"+imageLocation+"'")

img\_test\_original = readImage(imageLocation)

img\_test = img\_test\_original.copy()

img\_test = Laplacian(img\_test, CONST\_CV\_8U)

img\_test = GaussianBlur(img\_test, (3,3), 0)

img\_test = filter2D(img\_test, -1, sharpeningKernel)

result = matchTemplate(img\_test, img\_template, CONST\_TM\_CCOEFF)

min\_val, max\_val, min\_loc, max\_loc = minMaxLoc(result)

**print**(max\_val)

**if** (max\_val>400000):

**print**(" Template Found")

img\_test\_original = np.dstack([img\_test\_original, img\_test\_original, img\_test\_original])

rectangle(img\_test\_original, (max\_loc[0], max\_loc[1]), (max\_loc[0]+img\_template.shape[1], max\_loc[1]+img\_template.shape[0]), (0, 255, 0), 2)

**else**:

**print**(" Template Not Found")

writeImage(img\_test\_original, "t2\_"+str(i))

**print**("Template 2 positive images processed")

**print**("Starting Template 3 processing")

sharpeningKernel = getSharpeningKernel()

**for** i **in** range(1, numberOfPositive3Images+1):

img\_template = readImage(imageLocation\_template3)

**if** (len(template3\_sizeArray)>=numberOfPositive3Images):

img\_template = resize(img\_template , (template3\_sizeArray[i-1], template3\_sizeArray[i-1]))

**else**:

img\_template = resize(img\_template , (30, 30))

img\_template = Laplacian(img\_template, CONST\_CV\_8U)

img\_template = GaussianBlur(img\_template, (3,3),0)

imageLocation = imageLocation\_positiveImages3\_prefix+str(i)+imageLocation\_positiveImages3\_suffix

**print**("Image at '"+imageLocation+"'")

img\_test\_original = readImage(imageLocation)

img\_test = img\_test\_original.copy()

img\_test = Laplacian(img\_test, CONST\_CV\_8U)

img\_test = GaussianBlur(img\_test, (3,3), 0)

img\_test = filter2D(img\_test, -1, sharpeningKernel)

result = matchTemplate(img\_test, img\_template, CONST\_TM\_CCOEFF)

min\_val, max\_val, min\_loc, max\_loc = minMaxLoc(result)

**print**(max\_val)

**if** (max\_val>400000):

**print**(" Template Found")

img\_test\_original = np.dstack([img\_test\_original, img\_test\_original, img\_test\_original])

rectangle(img\_test\_original, (max\_loc[0], max\_loc[1]), (max\_loc[0]+img\_template.shape[1], max\_loc[1]+img\_template.shape[0]), (0, 0, 255), 2)

**else**:

**print**(" Template Not Found")

writeImage(img\_test\_original, "t3\_"+str(i))

**print**("Template 3 positive images processed")

**print**("Starting Negative Image processing against all templates")

**for** i **in** range(1, numberOfNegativeImages+1):

img\_template1 = readImage(imageLocation\_template1)

img\_template1 = resize(img\_template1 , (20,20))

img\_template1 = Laplacian(img\_template1, CONST\_CV\_8U)

img\_template1 = GaussianBlur(img\_template1, (3,3),0)

img\_template2 = readImage(imageLocation\_template2)

img\_template2 = resize(img\_template2 , (20,20))

img\_template2 = Laplacian(img\_template2, CONST\_CV\_8U)

img\_template2 = GaussianBlur(img\_template2, (3,3),0)

img\_template3 = readImage(imageLocation\_template3)

img\_template3 = resize(img\_template3 , (20,20))

img\_template3 = Laplacian(img\_template3, CONST\_CV\_8U)

img\_template3 = GaussianBlur(img\_template3, (3,3),0)

**if** (i==7):

**continue**

imageLocation = imageLocation\_negativeImages\_prefix+str(i)+imageLocation\_negativeImages\_suffix

**print**("Image at '"+imageLocation+"'")

img\_test\_original = readImage(imageLocation)

img\_test = img\_test\_original.copy()

img\_test = Laplacian(img\_test, CONST\_CV\_8U)

img\_test = GaussianBlur(img\_test, (3,3), 0)

img\_test = filter2D(img\_test, -1, sharpeningKernel)

result1 = matchTemplate(img\_test, img\_template1, CONST\_TM\_CCOEFF)

result2 = matchTemplate(img\_test, img\_template2, CONST\_TM\_CCOEFF)

result3 = matchTemplate(img\_test, img\_template3, CONST\_TM\_CCOEFF)

min\_val1, max\_val1, min\_loc1, max\_loc1 = minMaxLoc(result1)

min\_val2, max\_val2, min\_loc2, max\_loc2 = minMaxLoc(result2)

min\_val3, max\_val3, min\_loc3, max\_loc3 = minMaxLoc(result3)

img\_test\_original = np.dstack([img\_test\_original, img\_test\_original, img\_test\_original])

**print**(max\_val1)

**print**(max\_val2)

**print**(max\_val3)

**if** (max\_val1>400000):

**print**(" Template1 Found")

rectangle(img\_test\_original, (max\_loc1[0], max\_loc1[1]), (max\_loc1[0]+img\_template1.shape[1], max\_loc1[1]+img\_template1.shape[0]), (255, 0, 0), 2)

**else**:

**print**(" Template1 Not Found")

**if** (max\_val2>400000):

**print**(" Template2 Found")

rectangle(img\_test\_original, (max\_loc2[0], max\_loc2[1]), (max\_loc2[0]+img\_template2.shape[1], max\_loc2[1]+img\_template2.shape[0]), (0, 255, 0), 2)

**else**:

**print**(" Template3 Not Found")

**if** (max\_val3>400000):

**print**(" Template3 Found")

rectangle(img\_test\_original, (max\_loc3[0], max\_loc3[1]), (max\_loc3[0]+img\_template3.shape[1], max\_loc3[1]+img\_template3.shape[0]), (0, 0, 255), 2)

**else**:

**print**(" Template3 Not Found")

writeImage(img\_test\_original, "Neg"+str(i))

**print**("Negative images processed")

main()

**Output:**

Starting Template 1 processing

Image at './task3\_bonus/t1\_1.jpg'

864794.8125

Template Found

Image at './task3\_bonus/t1\_2.jpg'

599012.3125

Template Found

Image at './task3\_bonus/t1\_3.jpg'

611379.75

Template Found

Image at './task3\_bonus/t1\_4.jpg'

532922.0

Template Found

Image at './task3\_bonus/t1\_5.jpg'

878601.625

Template Found

Image at './task3\_bonus/t1\_6.jpg'

585157.5625

Template Found

Template 1 positive images processed

Starting Template 2 processing

Image at './task3\_bonus/t2\_1.jpg'

1736049.875

Template Found

Image at './task3\_bonus/t2\_2.jpg'

1005640.6875

Template Found

Image at './task3\_bonus/t2\_3.jpg'

1479634.125

Template Found

Image at './task3\_bonus/t2\_4.jpg'

1119297.75

Template Found

Image at './task3\_bonus/t2\_5.jpg'

1579382.125

Template Found

Image at './task3\_bonus/t2\_6.jpg'

1480881.125

Template Found

Template 2 positive images processed

Starting Template 3 processing

Image at './task3\_bonus/t3\_1.jpg'

1217451.75

Template Found

Image at './task3\_bonus/t3\_2.jpg'

1042473.5

Template Found

Image at './task3\_bonus/t3\_3.jpg'

1281221.75

Template Found

Image at './task3\_bonus/t3\_4.jpg'

1051726.75

Template Found

Image at './task3\_bonus/t3\_5.jpg'

1005968.3125

Template Found

Image at './task3\_bonus/t3\_6.jpg'

917122.25

Template Found

Template 3 positive images processed

Starting Negative Image processing against all templates

Image at './task3\_bonus/neg\_1.jpg'

431679.8125

545770.25

288935.09375

Template1 Found

Template2 Found

Template3 Not Found

Image at './task3\_bonus/neg\_2.jpg'

464935.6875

530749.5625

409781.5

Template1 Found

Template2 Found

Template3 Found

Image at './task3\_bonus/neg\_3.jpg'

381674.03125

465951.53125

343886.5625

Template1 Not Found

Template2 Found

Template3 Not Found

Image at './task3\_bonus/neg\_4.jpg'

620208.75

485574.75

431591.8125

Template1 Found

Template2 Found

Template3 Found

Image at './task3\_bonus/neg\_5.jpg'

526614.5

434617.65625

316558.5

Template1 Found

Template2 Found

Template3 Not Found

Image at './task3\_bonus/neg\_6.jpg'

448916.125

297159.6875

247536.203125

Template1 Found

Template3 Not Found

Template3 Not Found

Image at './task3\_bonus/neg\_8.jpg'

437199.9375

361765.1875

217250.171875

Template1 Found

Template3 Not Found

Template3 Not Found

Image at './task3\_bonus/neg\_9.jpg'

472234.875

483697.53125

321448.5625

Template1 Found

Template2 Found

Template3 Not Found

Image at './task3\_bonus/neg\_10.jpg'

472966.03125

533719.4375

315643.375

Template1 Found

Template2 Found

Template3 Not Found

Image at './task3\_bonus/neg\_11.jpg'

429694.375

514381.15625

355578.9375

Template1 Found

Template2 Found

Template3 Not Found

Image at './task3\_bonus/neg\_12.jpg'

458366.71875

334345.4375

249977.625

Template1 Found

Template3 Not Found

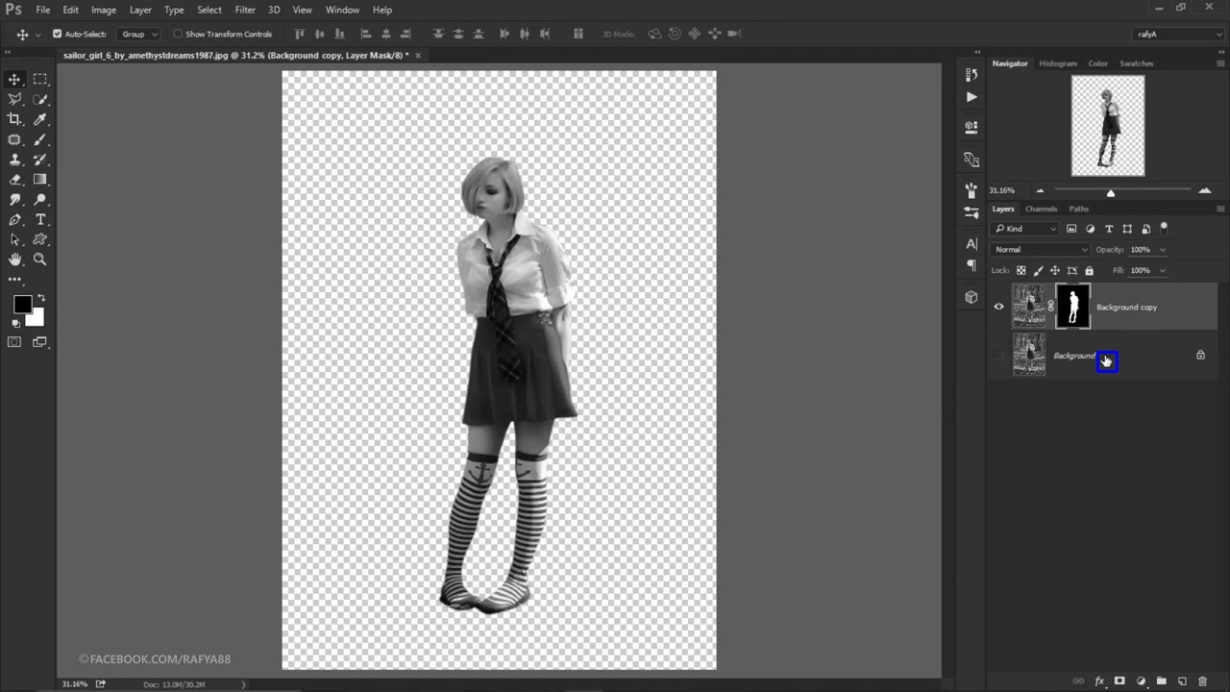
Template3 Not Found

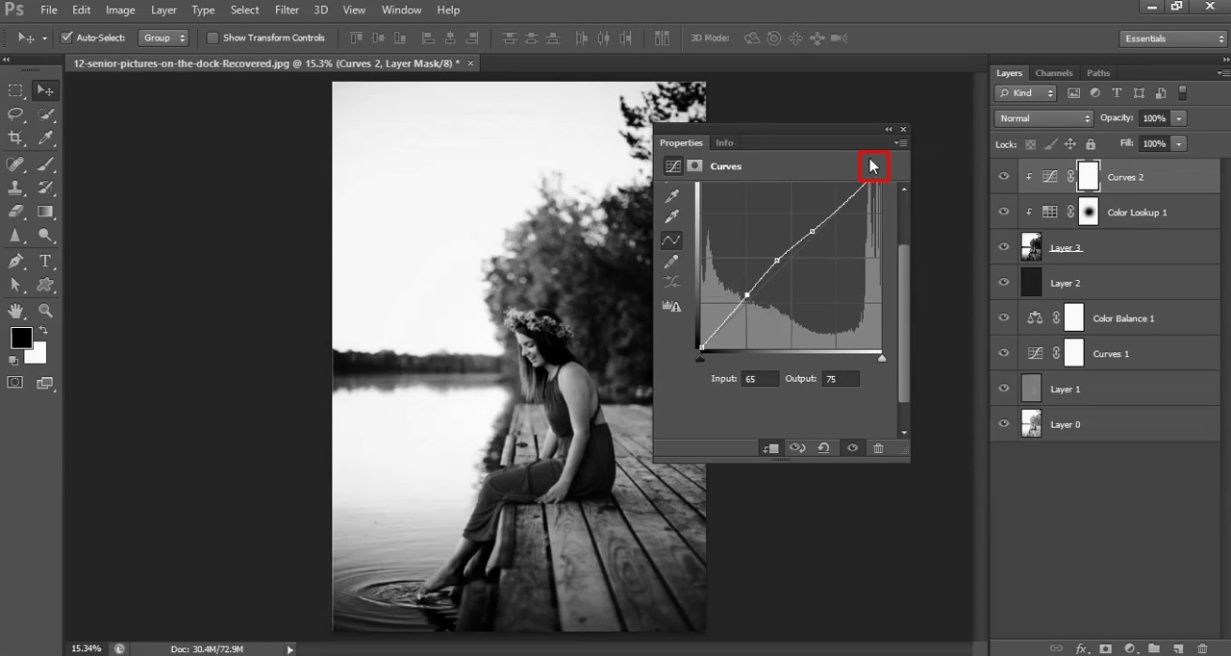
Negative images processed

[Finished in 4.4s]

**Accuracy:** My code has an overall accuracy of 83% for template 1, 83% for template 2, 100% for template 3 and 0% for negatives.

**Some Output Images:**

****

****

****