Project 1 of CSE 473/573

Shaoming Xu UB# 50247057

# Task 1



Fig1.1 pos\_edge\_x\_2.png



Fig1.2 pos\_edge\_y\_2.png

The Fig1.1 shows edges along x direction. The Fig1.2 shows edges along y direction. Both images are got from eliminated zero values with method 2 then times 255. If you are interested see all images include magnitude one, please run python3 task1.py.

## Task1 Source code (Only include key funcions):

Notice: here only include key functions. You can see all code in my project folder.

VERTICAL\_SOBEL\_3BY3 = np.array([[1,0,-1],

[2,0,-2],

[1,0,-1]])

HORIZONTAL\_SOBEL\_3BY3 = np.array([[1,2,1],

[0,0,0],

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

def texture\_filtering(img\_gray, kernel):

"""

Purpose:

use to filter the gray image given the kernel

Input:

img\_gray:

an two dimension ndarray matrix, dtype:usually is uint8 representint the gray image.

kernel:

a two dimension ndarray matrix

Output:

The filtered image without padding around.

"""

row\_pad = math.floor(kernel.shape[0] / 2)

col\_pad = math.floor(kernel.shape[1] / 2)

img\_gray = np.ndarray.tolist(img\_gray)

img\_gray = np.asarray(mnp.pad(img\_gray, row\_pad, row\_pad, col\_pad, col\_pad, 0))

img\_res = np.asarray(mnp.zeros(img\_gray.shape[0], img\_gray.shape[1]))

flipped\_kernel = np.asarray((mnp.flip(np.ndarray.tolist(kernel))))

for i in range(row\_pad, img\_gray.shape[0] - row\_pad):

for j in range(col\_pad, img\_gray.shape[1] - col\_pad):

patch = mnp.inner\_product(img\_gray[i-row\_pad:i+row\_pad+1, j-col\_pad:j+col\_pad+1], flipped\_kernel)

img\_res[i,j] = mnp.sum\_all(patch)

return img\_res[row\_pad: img\_res.shape[0] - row\_pad, col\_pad:img\_res.shape[1] - col\_pad]

# Task2

1. include images of the second and third octave and specify their resolution (width height, unit pixel);



Fig\_2.1.1 octave\_2\_img (resolution: (229, 375))



Fig\_2.1.2 octave\_3\_img (resolution: (115, 188))

1. Include DoG images obtained using the second and third octave.

Second octave:



Third octave:



1. Clearly show all the detected keypoints using white dots on the original image.





task2_img/combined_keypoints_imgs/octave_4_keypoints_img.jpg

1. provide coordinates of the five left-most detected keypoints

Consider points on edge: (114, 0), (272, 0), (310, 0), (327, 0), (348, 0)

Not consider points on edge: [(1, 1), (231, 1), (266, 1), (324, 1), (351, 1)]

## Task2 Source code (Only include key funcions):

Notice: here only include key functions. You can see all code in my project folder.

SIGMAS = np.array([[1/np.sqrt(2), 1, np.sqrt(2), 2, 2\*np.sqrt(2)],

[np.sqrt(2), 2, 2\*np.sqrt(2), 4, 4\*np.sqrt(2)],

[2\*np.sqrt(2), 4, 4\*np.sqrt(2), 8, 8\*np.sqrt(2)],

[4\*np.sqrt(2), 8, 8\*np.sqrt(2), 16, 16\*np.sqrt(2)]])

def gaussin\_val(x, y, sigma):

"""

Purpose:

Compute the gaussin val

x:

a real number

y:

a real number

sigma:

a real number

"""

a = 1 / (2 \* np.pi \* mnp.power(sigma,2))

b = np.exp(-(mnp.power(x,2) + mnp.power(y,2)) / (2 \* mnp.power(sigma,2)))

return a \* b

def gaussin\_kernel\_gen(sigma, size=7):

"""

Purpose:

compute the gaussin kernel given the sigma and kernel size

Input:

sigma:

a real number

size:

int, the size of kernel

Output:

a gaussin kernel

"""

if(size % 2 == 0):

raise Exception("kernel size should be odd number")

mat = np.asarray(mnp.zeros(size,size))

pad = int(size/2)

dividend = 0

for i in range(size):

for j in range(size):

mat[i,j] = gaussin\_val(j-pad, pad-i, sigma)

dividend += mat[i,j]

return mat / dividend

def kernels\_db\_gen(sigmas = SIGMAS):

"""

Purpose:

Generate a series of gaussin kernles given a array of sigmas

Input:

sigmas:

a two dimension array which contains sigmas

Output:

a two dimension lists, each element is a kernel.

"""

kernels = []

for row in sigmas:

mats = []

for sigma in row:

mats.append(gaussin\_kernel\_gen(sigma, 7))

kernels.append(mats)

return kernels

def resized\_imgs\_bank\_gen(img\_gray, layer):

resized\_imgs\_bank = []

for i in range(layer):

img\_resized = np.asarray(resize\_shrink(img\_gray, mnp.power(1/2,i), mnp.power(1/2,i)))

resized\_imgs\_bank.append(img\_resized)

return resized\_imgs\_bank

def img\_bank\_gen(img\_gray, kernels\_db, resized\_imgs\_bank):

"""

Purpose:

Generate a series filtered image given the kernels database

Input:

img\_gray:

a two dimension matrix representing the gray image, usually the dtype is uint8

kernels\_db:

a two dimension list, each elements is a kernel.

resized\_imgs\_bank:

a list contains resized\_imgs

Output:

the img\_bank, a two dimension list, each elements is a filterd image.

"""

res = []

print("in img\_bank\_gen")

for i, row in enumerate(kernels\_db):

res\_row = []

img\_resized = resized\_imgs\_bank[i]

for kernel in row:

res\_row.append(texture\_filtering(img\_resized, kernel))

print("fininsh a filterd img")

print("row",i,"fininshed")

res.append(res\_row)

return res

def dog\_bank\_gen(img\_bank):

"""

Purpose:

Generate the Dog image for the images in img\_bank

Input:

img\_bank:

a two dimension list, each elemetns is a filterd image.

Output:

res: a dog\_bank, a two dimension list, each elements is a Dog image

"""

res = []

for row in img\_bank:

res\_row = []

for i in range(len(row[:-1])):

res\_row.append(row[i+1] - row[i])

res.append(res\_row)

return res

def check\_min\_max(upper\_patch, patch, lower\_patch):

"""

Purpose:

check if the middle pixel of patch is the maximum or the minimum pixel in the three patchs

Input:

Upper\_patch:

patch:

lower\_patch:

each patch is a 3 by 3 two dimension matrix.

Output: boolean

"""

if ( (patch[1,1], 1) == mnp.min\_all\_count(patch) and patch[1,1] < mnp.min\_all(upper\_patch)

and patch[1,1] < mnp.min\_all(lower\_patch)

or (patch[1,1],1) == mnp.max\_all\_count(patch) and patch[1,1] > mnp.max\_all(upper\_patch)

and patch[1,1] > mnp.max\_all(lower\_patch)):

return True

else:

return False

def key\_points\_gen(img\_upper, img, img\_lower):

"""

Purpose:

Generate keypoints image

Input:

img\_upper:

img:

img\_lower:

three gray images

Output:

res:

a keypoints image in where the white pixels(255) are keypoints.

"""

res = []

img\_upper = np.ndarray.tolist(img\_upper)

img\_upper = np.asarray(mnp.pad(img\_upper,1,1,1,1))

img = np.ndarray.tolist(img)

img = np.asarray(mnp.pad(img,1,1,1,1))

img\_lower = np.ndarray.tolist(img\_lower)

img\_lower = np.asarray(mnp.pad(img\_lower,1,1,1,1))

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

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

upper\_patch = img\_upper[i-1:i+2, j-1:j+2]

patch = img[i-1:i+2, j-1:j+2]

lower\_patch = img\_lower[i-1:i+2, j-1:j+2]

if check\_min\_max(upper\_patch, patch, lower\_patch):

res.append((i-1,j-1))

return res

def key\_points\_bank\_gen(dog\_bank):

"""

Purpose:

Generate the keypoints imgs bank by the dog\_bank

input:

dog\_bank:

a two dimension list, each elements is a Dog image

Output:

key\_points\_imgs\_bank:

a two dimensions list, each element in the list is a keypoints image.

"""

key\_points\_bank = []

for i in range(len(dog\_bank)):

print("start new row")

key\_points\_bank\_row = []

for j in range(1, len(dog\_bank[i]) - 1):

img\_lower = dog\_bank[i][j-1]

img = dog\_bank[i][j]

img\_upper = dog\_bank[i][j+1]

key\_points\_bank\_row.append(key\_points\_gen(img\_upper, img, img\_lower))

print("finish a key\_points\_list")

key\_points\_bank.append(key\_points\_bank\_row)

return key\_points\_bank

# main function

if \_\_name\_\_ == "\_\_main\_\_":

img = cv2.imread("../task2\_img/task2.jpg", 0)

kernels\_db = kernels\_db\_gen()

resized\_imgs\_bank = resized\_imgs\_bank\_gen(img, len(kernels\_db))

img\_bank = img\_bank\_gen(img, kernels\_db, resized\_imgs\_bank)

dog\_bank = dog\_bank\_gen(img\_bank)

key\_points\_bank = key\_points\_bank\_gen(dog\_bank)

merged\_key\_points = merge\_key\_points\_bank(key\_points\_bank)

print("five left most points:",

[(b,a) for (a,b) in heapq.nsmallest(5,[(b,a) for (a,b) in merged\_key\_points])])

print("five left most points:(Consider the edge case)",

[(b,a) for (a,b) in heapq.nsmallest(5,[(b,a) for (a,b) in merged\_key\_points])])

five\_left = []

for val in merged\_key\_points:

if val[1] == 1:

five\_left.append(val)

five\_left.sort()

print("five left most points:(Not consider the edge case)", five\_left[:5])

save\_resized\_imgs(resized\_imgs\_bank, True)

save\_blured\_imgs(img\_bank, True)

save\_dog\_imgs(dog\_bank, True)

save\_combined\_key\_points\_imgs(key\_points\_bank, resized\_imgs\_bank, True)

# Task3

1. Proposed method for task 3

In task3, I use the my own template task3_img/template_1.jpeg which is got from the image. And I found using it, my function work much better than using the original one.

My function can use all six methods provided in opencv2 library. The mathematic equation behind the methods can be found here: <https://docs.opencv.org/3.4.2/de/da9/tutorial_template_matching.html>.

Beside the six methods, my function enables users to use mask on templates. The mask helps to ignore the useless pixels in template. It is a binary mask, which has a threshold 80, any value greater than it will be 1, otherwise will be 0. In the experiments, some arrows in the images with white background can be detected after using mask. however, I found the mask not works well even in some good thresholds like 80, 100, and 120 etc… besides that, the mask only works on 'cv2.TM\_CCORR\_NORMED' and 'cv2.TM\_SQDIFF' methods

Actually, I also try to preprocess the templates so as to eliminate useless rows and column. But it does not work well in given images. I think the reason is most cursors in the given images are on dark background which is perfect for my template. So I commented out it in code. But in the future, if you want to detect the cursor with varying background, I recommend you use mask and preprocess the template beforehand.

Finally, I also adopt Laplacian transformation method to preprocess image. The steps are as followed:

1. blur the image with a 3x3 Gaussian kernel, sigma could be automatically computed according to width of the kernel by OpenCV.
2. apply Laplacian transformation to the template and the blurred image.
3. use template matching to match the transformed template and the transformed image.

This Laplacian method matches 10 images out of 15 images, without using it it only can matches 9 images.

In my experiments, I satisfy with the results generated from TM\_CCOEFF\_NORMED, TM\_CCORR\_NORMED, and TM\_SQDIFF\_NORMED methods. You can see all results in folders ../task3\_image/(Method\_Name) folders.

For the bonus part, we only need to get the customized templates from the images, then use the same methods as mentioned above. Then we can get the results.

In my demo, for bonus part there are total 18 images, and I get right results for 16 images which is already very precise.

Ps: The result images store in corresponding folders, you can go and see them. For example, /task3\_bonus/TM\_CCORR\_NORMED saves the results images generated by TM\_COOR\_NORMED method for the bonus part.

## Task3 Source code (Only include key funcions):

Notice: here only include key functions. You can see all code in my project folder.

METHODS = ['cv2.TM\_CCOEFF', 'cv2.TM\_CCOEFF\_NORMED', 'cv2.TM\_CCORR',

'cv2.TM\_CCORR\_NORMED', 'cv2.TM\_SQDIFF', 'cv2.TM\_SQDIFF\_NORMED']

def preproces\_laplacian(img):

blur\_img = cv2.GaussianBlur(img,(3,3),0)

return cv2.Laplacian(blur\_img,cv2.CV\_8U)

def template\_match(loc="../task3\_bonus/", temp\_name="template\_1.jpg", img\_prefix="t1\_",num=6,meth = 'cv2.TM\_CCORR\_NORMED', has\_mask = False):

"""

Input:

meth: String

The names of template matching method

has\_mask: boolean

True: use mask on template. Only support TM\_CCORR\_NORMED and TM\_SQDIFF

False: not use

"""

m = re.search(r'cv2.(\w+)', meth)

save\_loc = loc + m.group(1)

template = cv2.imread(loc + temp\_name,0)

mask\_ = None

if has\_mask:

mask\_ = np.ones(template.shape,dtype=np.uint8)

mask\_[template < 80] = 0

save\_loc = save\_loc + "/mask"

method = eval(meth)

#preproces\_laplacian

template = preproces\_laplacian(template)

try:

os.makedirs(save\_loc)

except FileExistsError:

print("use existing folder:", save\_loc)

for i in range(1,num+1):

name = img\_prefix + str(i) + ".jpg"

img = cv2.imread(loc + name)

img\_gray = cv2.imread(loc + name, 0)

img\_gray = preproces\_laplacian(img\_gray)

h, w = template.shape

# Apply template Matching

res = cv2.matchTemplate(img\_gray,template,method, mask = mask\_)

min\_val, max\_val, min\_loc, max\_loc = cv2.minMaxLoc(res)

# If the method is TM\_SQDIFF or TM\_SQDIFF\_NORMED, take minimum

if method in [cv2.TM\_SQDIFF, cv2.TM\_SQDIFF\_NORMED]:

top\_left = min\_loc

else:

top\_left = max\_loc

bottom\_right = (top\_left[0] + w, top\_left[1] + h)

cv2.rectangle(img,top\_left, bottom\_right, (0,0,255), 2)

cv2.imwrite(save\_loc + "/" + img\_prefix + str(i) + "\_res" + ".jpg", img)

def preprocess\_template(template):

## elimate the not useful information

index\_top = 0

index\_bottom = template.shape[0] - 1

index\_left = 0

index\_right = template.shape[1] - 1

for row in template:

if max(row) > 100:

break

index\_top += 1

for row in reversed(template):

if max(row) > 100:

break

index\_bottom -= 1

for col in template.T:

if max(col) > 100:

break

index\_left += 1

for col in reversed(template.T):

if max(col) > 100:

break

index\_right -= 1

res = template[index\_top : index\_bottom + 1, index\_left : index\_right + 1]

return res

template\_match("../task3\_img/",'template\_1.jpeg',"pos\_", 15,'cv2.TM\_CCORR\_NORMED', False)

template\_match("../task3\_img/",'template\_1.jpeg',"pos\_", 15, 'cv2.TM\_CCOEFF\_NORMED', False)

template\_match("../task3\_img/",'template\_1.jpeg',"pos\_", 15, 'cv2.TM\_SQDIFF\_NORMED', False)

##For bonus part:

template\_match("../task3\_bonus/", "template\_1.jpg", "t1\_", 6)

template\_match("../task3\_bonus/", "template\_2.jpg", "t2\_", 6)

template\_match("../task3\_bonus/", "template\_3.jpg", "t3\_", 6)

## Appendix, my library: mycv.py and mynumpy.py

In above modules, I import these two libraries as:

*from mycv import resize\_shrink*

*import mynumpy as mnp*