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

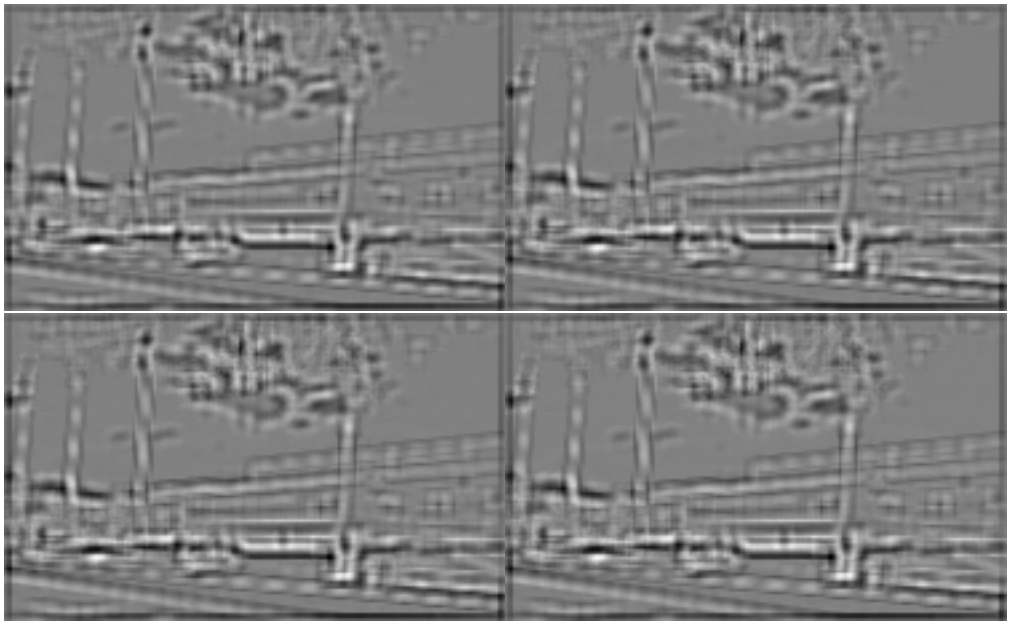
1. include images of the second and third octave and specify their resolution (width height, unit pixel);

Fig_2.1.1 octave_2_img (resolution: (229, 375))

Fig_2.1.2 octave_3_img (resolution: (115, 188))

2. Include DoG images obtained using the second and third octave.
   Second octave:

Third octave:



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

4. 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  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