SIT789 – Robotics, Computer Vision and Speech Processing

Pass Task 3.1: Image descriptors

Objectives

The objectives of this lab include:

- Practising with different keypoint detectors and image descriptors
- Applying image descriptors to image matching

Tasks

1. Harris corner detector

Download the supplied image empire.jpg (in OnTrack) and copy it into your working directory. In this task, you need to convert the image in empire.jpg to grayscale using the following command.

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt

img = cv.imread('empire.jpg') # load image
img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
plt.imshow(img_gray, 'gray')
```

You should have week 3 handout with you for reference. We first define necessary parameters including:

```
local_region_size = 3 # i.e., W=3x3, see slide 6 in week 3 handout
kernel_size = 3 # Sobel kernel’s size used to calculate horizontal/vertical derivatives
k = 0.04 # parameter k in side 6 in week 3 handout
threshold = 1000.0 #threshold theta introduced in slide 6 in week 3 handout
```

img_gray contains integer values (for pixel intensity) and therefore needs to be converted to float prior to applying the Harris corner detector.

```
img_gray = np.float32(img_gray)
```

We now can compute the Harris response image of img_gray by using the following command

```
Harris_res_img = cv.cornerHarris(img_gray, local_region_size, kernel_size, k)
plt.imshow(Harris_res_img, 'gray')
```

We then select and highlight corners from Harris_res_img using our defined threshold.

```
highlighted_colour = [0, 0, 255] # Blue:0, Green:0, Red: 255
highlighted_img = img.copy()
highlighted_img[Harris_res_img > threshold] = highlighted_colour
plt.imshow(cv.cvtColor(highlighted_img, cv.COLOR_BGR2RGB)) # RGB-> BGR
```

Your task now is to calculate the number of detected corners from Harris_res_img after thresholding.
Hint: you can use `np.sum(Harris_res_img > threshold)` to calculate the number of detected corners.

You can also set the threshold adaptively. For example, you can define the threshold as 1% of the maximum response.

```
ratio = 0.01 # 1%
threshold = ratio * Harris_res_img.max()
print(threshold)
```

Your task is to experiment the Harris corner detector with different ratios in {0.1%, 0.5%, 1%}, observe the corresponding corner detection results and the number of corners detected, and discuss your observation.

2. SIFT

In this section, we will apply SIFT detector and descriptor on the grayscale image img_gray. First, you need to initialise SIFT detector as:

```
sift = cv.SIFT_create()
```

2.1. Keypoint detection

To extract keypoints (aka interest points), you can do as,

```
# Re-initialise img_gray as its content has been converted to float in Section 1
img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
kp = sift.detect(img_gray, None)
```

To visualise SIFT keypoints, you can create an image which is identical to img_gray and draw keypoints on this new image as follows.

```
img_gray_kp = img_gray.copy()
img_gray_kp = cv.drawKeypoints(img_gray, kp, img_gray_kp)
plt.imshow(img_gray_kp)
print("Number of detected keypoints: %d" % (len(kp)))
```

To better observe the keypoints, you can save img_gray_kp to file. To visualise local image regions used for extracting SIFT descriptors, you can perform:

```
img_gray_kp = cv.drawKeypoints(img_gray, kp, img_gray_kp, flags=cv.DRAW_MATCHES_FLAGS_D
RAW_RICH_KEYPOINTS)
plt.imshow(img_gray_kp)
```

2.2. Descriptor computation

Based on the detected keypoints, we extract SIFT descriptors using the following commands.

```
kp, des = sift.compute(img_gray, kp)
```

des is an array of size (2804x128). The rows correspond to keypoints and thus there are 2804 rows. The columns represent descriptors, each column stores a descriptor including 128 elements. You can check the dimension of des by calling:

```
print(des.shape)
```

Your task is to experiment with SIFT on the image files empire_45.jpg, empire_zoomedout.jpg, fisherman.jpg provided in the resources.

3. Image matching using SIFT

In this section, we will investigate how SIFT is robust under rotation and scaling factors. To conduct such an investigation, we will apply SIFT to image matching. Specifically, if an image is deformed due to some transformations, e.g., scaling, rotation, etc., we still can match it with its deformed version by finding corresponding interest points between the original version and its deformed version. In this task, we will examine how to match interest points using SIFT.

We will use the image files empire.jpg, empire_45.jpg, empire_zoomedout.jpg, fisherman.jpg provided in the resources. Instead of detecting the keypoints and then extracting the descriptors at the keypoints in an input image like Section 2, we can do both keypoint detection and descriptor extraction in one command as,

```
kp, des = sift.detectAndCompute(img_gray, None)
```

Using this manner, we can detect the keypoints and the descriptors for the images in empire_45.jpg, empire_zoomedout.jpg, and fisherman.jpg as follows.

```
#load images
img_45 = cv.imread('empire_45.jpg')
img_zoomedout = cv.imread('empire_zoomedout.jpg')
img_another = cv.imread('fisherman.jpg')

#convert the images to grayscale
img_45_gray = cv.cvtColor(img_45, cv.COLOR_BGR2GRAY)
img_zoomedout_gray = cv.cvtColor(img_zoomedout, cv.COLOR_BGR2GRAY)
img_another_gray = cv.cvtColor(img_another, cv.COLOR_BGR2GRAY)

#extract keypoints and descriptors
kp_45, des_45 = sift.detectAndCompute(img_45_gray, None)
kp_zoomedout, des_zoomedout = sift.detectAndCompute(img_zoomedout_gray, None)
kp_another, des_another = sift.detectAndCompute(img_another_gray, None)
```

We then use BFM (Brute-Force Matcher) to match two images. The BFM receives two sets of descriptors, called query and train, as inputs. For each descriptor in the query, the BFM finds its best matching descriptor in the train. The matching score between two descriptors is measured via the distance, e.g., Euclidean distance, between the descriptors. For instance, to find matching descriptors of des on des_45, we perform:

```
# Initialise a brute force matcher with default params
bf = cv.BFMatcher()
train = des_45
query = des
matches_des_des_45 = bf.match(query, train)
```

Note:

1. The number of matches is also the number of keypoints in query. You can check this by comparing len(matches) and len(query)
2. bf.match(des, des_45) is not identical to bf.match(des_45, des).

matches_des_des_45 is a list in which each element is a DMatcher object. For example, the i-th match in matches_des_des_45 include:

- matches_des_des_45[i].trainIdx: the ID of a descriptor in des_45,
- matches_des_des_45[i].queryIdx: the ID of a descriptor in des,
- matches_des_des_45[i].distance: the distance, e.g., Euclidean distance, between the descriptors whose IDs are matches_des_des_45[i].queryIdx (in des) and matches_des_des_45[i].trainIdx (in des_45) respectively. The quality of a match is represented by its associated distance which is also considered as matching score.

We can sort the matches based on their matching scores as,

```
matches_des_des_45 = sorted(matches_des_des_45, key = lambda x:x.distance)
```

To visualise the matches, we draw the best 10 matches.

```
# Draw the best 10 matches.
nBestMatches = 10
matching_des_des_45 = cv.drawMatches(img_gray, kp, img_45_gray, kp_45,
                              matches_des_des_45[:nBestMatches],
                              None,
                              flags = cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
plt.imshow(matching_des_des_45)
```

The following code presents in the detailed information of these best 10 matches in matches_des_des_45.

```
kp_train = kp_45
kp_query = kp
for i in range (0, nBestMatches):
    print("match ", i, " info")
    print("\tdistance:", matches_des_des_45[i].distance)
    print("\tkeypoint in train: ID:", matches_des_des_45[i].trainIdx, " x:",
          kp_train[matches_des_des_45[i].trainIdx].pt[0], " y:",
          kp_train[matches_des_des_45[i].trainIdx].pt[1])
    print("\tkeypoint in query: ID:", matches_des_des_45[i].queryIdx, " x:",
          kp_query[matches_des_des_45[i].queryIdx].pt[0], " y:",
          kp_query[matches_des_des_45[i].queryIdx].pt[1])
```

Note: matches_des_des_45 can be different from matches_des_45_des. To see this, we can swap des and des_45 and redo the matching:

```
matches_des_45_des = bf.match(des_45, des)
matches_des_45_des = sorted(matches_des_45_des, key = lambda x:x.distance)
matching_des_45_des = cv.drawMatches(img_45_gray, kp_45, img_gray, kp,
                              matches_des_45_des[:nBestMatches],
                              None,
                              flags = cv.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
plt.imshow(matching_des_45_des)
```

To compare empire.jpg and empire_45.jpg, we match their descriptors and achieve a list of matches. From this list, we find the best N matches (i.e., `nBestMatches` defined above), and calculate the dissimilarity between the two images as,

$$\frac{1}{2}\left(\sum_{i=1}^{N} \mathrm{matches\_des\_des\_45[i].distance} + \sum_{i=1}^{N} \mathrm{matches\_des\_45\_des[i].distance}\right)$$

Using the above formula, you are to calculate the dissimilarity between:

(i) empire.jpg and empire_45.jpg
(ii) empire.jpg and empire_zoomedout.jpg
(iii) empire.jpg and fisherman.jpg

You are then to perform:

- Experiments (i)-(iii) with `nBestMatches` in the range {10, 50, 100}
- Comparison of the pairs of images in (i)-(iii) using their dissimilarities

Submission instructions

1. Perform tasks required in Sections 1-3
2. Complete the supplied answer sheet with required results and discussions.
3. Submit the answer sheet (in pdf format) to OnTrack.
4. Submit your code (in .py format) to OnTrack