CS 484, Spring 2019
Homework Assignment 2:
Local Features

Full name: Fuad Aghazada
ID: 21503691
Used Programming language: Python3
Used Library: OpenCV 3.4.2.16

Step 1: Preparing data

To load the images, as said in the assignment, a text file with the all image file names is read into a list as in a sorted way. Then for each file name, the corresponding image has been read as a grayscale image object (2D matrix) and inserted into the list of all images. The image list is returned from the function. In addition, for a large number of images, to fit the panorama into the screen, a *scale* parameter has been added to the function so that the size of the images can be reduced by having a large *scale* number (for *scale* = 2, image dimensions are halved):

```
def load_images(filename, scale = 2):
  dir = ROOT + '/data/'
  images = []
  files = []

  with open(ROOT + '/txt/' + filename, 'r') as file:
      for line in file:
          files.append(line.replace('\n', ''))

  # Sorting the file according to their names
  files = sorted(files)

  # Loading the images into a list
  for file in files:
      img = cv2.imread(dir + file)
      img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
      img = cv2.resize(img, (img.shape[1] // scale, img.shape[0] // scale))
      images.append(img)

  return images
```

Step 2: Detecting local features

To obtain the features of an image the Scale-Invariant Feature Transform (SIFT) detector of OpenCV has been used:

```
# Initializing SIFT detector
sift = cv2.xfeatures2d.SIFT_create()
```

In order to detect the interest points (key points), the *detect* function of SIFT detector has been used:

```
def detect_local_features(image):
    print("Detecting keypoints...")
    interest_points = sift.detect(image, None)
    print("Keypoints detected!\n--")
    return interest_points
```

Finally, for accessing the attributes (x and y locations, scale, and orientation) of a key point, the respective properties of *KeyPoint* object have been used:

.pt → coordinates of the key point [1]

.size → diameter of the meaningful key point neighborhood [1]

.angle → computed orientation of the key point in [0, 360] degrees and measured relative to image coordinate system in clockwise [1]

Sample output for properties of a key point (coordinate, scale, orientation):
(105.16278076171875, 353.1541442871094) 2.275583505630493 155.365966796875

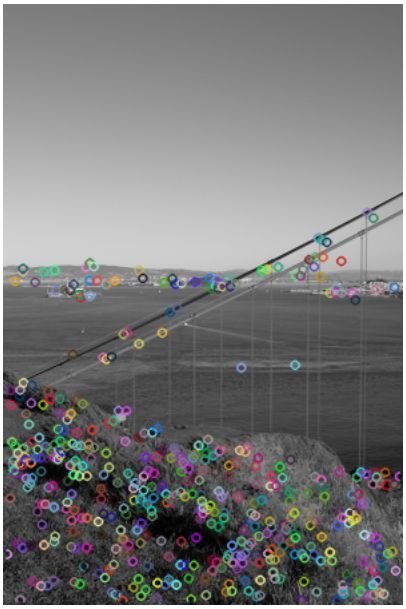

Figure 1. Key points of an image

Step 3: Describing local features

To describe the detected key points 2 different methodologies has been suggested to be used.

- **Gradient-based descriptor:**
    - The descriptor of an image has been calculated by using *compute* function of SIFT detector:

```
def detect_describe_local_features(image):

    print("Computing SIFT keypoints & descriptors...")

    (keypoints, descriptor) =
sift.detectAndCompute(image.astype('uint8'), None)
    features = {"keypoints": keypoints, "descriptors": descriptor}

    print("SIFT Keypoints & Descriptors computed!\n--")

    return features
```

    The descriptor is a 2D matrix for which each row corresponds to a key point and contains a description of length 128 in it:

- **Raw pixel-based descriptor:** *(did not work)*
    - The descriptor of the key point is defined by calculating greyscale histogram of the region of a square by applying some transformations around the key point using its attributes mentioned in Step 2.

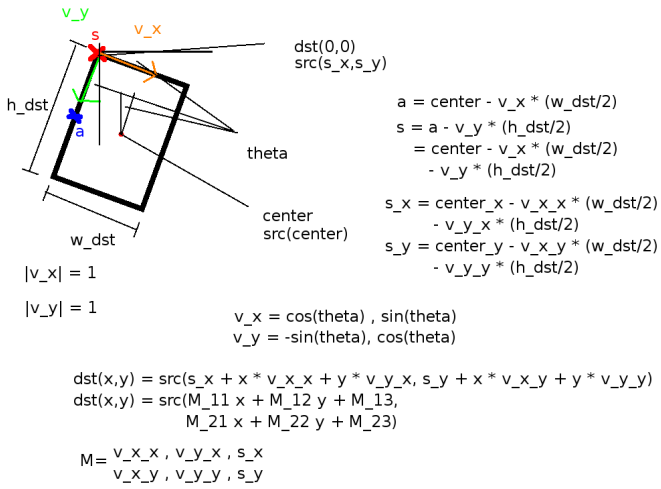

Figure 2. Extracting the square of the region of a key point [2]

The function named *subimage* converts the math above into a Python code:

```
def subimage(image, center, theta, width, height):

    theta *= math.pi / 180 # convert to rad

    v_x = (math.cos(theta), math.sin(theta))
    v_y = (-math.sin(theta), math.cos(theta))
    s_x = center[0] - v_x[0] * ((width-1) / 2) - v_y[0] * ((height-1) / 2)
    s_y = center[1] - v_x[1] * ((width-1) / 2) - v_y[1] * ((height-1) / 2)

    mapping = np.array([[v_x[0],v_y[0], s_x],
                        [v_x[1],v_y[1], s_y]])


    return cv2.warpAffine(image, mapping, (width, height), flags =
cv2.WARP_INVERSE_MAP, borderMode = cv2.BORDER_REPLICATE)
```

- After being able to get the sub-image from the given image with the given parameters of center coordinates, orientation, and scale, for every key point the sub-image (square around the key point) is constructed.
- After the construction of the square image part, the 256 bin greyscale histogram has been calculated for each square (key point). Therefore, all these histograms generate the 2D descriptor matrix for the given image:

```
def describe_raw_pixel_based(image):

    # Detecting keypoints
    keypoints = detect_local_features(image)
```

```
        descriptor = []
        for keypoint in keypoints:
            # Properties of keypoint
            coordinates = keypoint.pt
            size = keypoint.size
            orientation = keypoint.angle

            # Square around keypoint with given coordinate, orientation and scale
            img = np.copy(image)
            square = subimage(img, coordinates, orientation, int(size) ,
    int(size))

            # Calculating the histogram
            histogram = cv2.calcHist([square], [0], None, [256], [0, 256])
            histogram = [x[0] for x in histogram]
            descriptor.append(histogram)

        return {"keypoints": keypoints, "descriptors": np.asarray(descriptor,
    dtype = 'float32')}
```

Note: Applying all the mentioned steps I could not get the desired result.

Step 4: Feature matching

After having descriptors for each key point, the next step is to determine the matching key points. In order to implement this step, the Euclidean distance has been calculated for each key point using their descriptors:

```
def calc_eculidean_distance(desc1, desc2):
      # Calculating the distance
      distance = np.sqrt(np.sum((desc1[:, np.newaxis, :] - desc2[np.newaxis, :, :])
      ** 2, axis = -1))
```

The results *distance* is 2D matrix which contains the Euclidean distances between the given descriptors of two images. To exemplify, distance[1, 3] corresponds to the Euclidean distance between key point 1 and key point 3.

Having a function which calculates the Euclidean distance between two descriptors of the two images, it is needed to apply a threshold to get matching key points and minimize the number of key points between the images.

```
    def match(desc1, desc2, threshold):

        print("Computing matching keypoints...")

        # For keeping the mathcing points
        matches = []
```

```
        # Number of keypoints for each image
        num_keypoints1 = desc1.shape[0]
        num_keypoints2 = desc2.shape[0]

        # Calculating the distances between keypoints of two images
        distances = calc_eculidean_distance(desc1, desc2)

        for i in range(num_keypoints1):
                for j in range(num_keypoints2):
                        if distances[i][j] < threshold:
                                matches.append((i, j))

        print("Matching points computed!\n--")

        return matches
```

After several experiments threshold value has been chosen as 100. Having a high threshold value increases the number of matching key points by also increasing the computational cost. Therefore, 100 as a threshold value seems like a valid choice.

And in order to verify the result of matching images, I wrote the following function to display the matching points by connecting them with 2D lines:

```
def draw_matches(image1, image2, keypoints1, keypoints2, matches):
        # Dimensions of the images
        h1, w1 = image1.shape
        h2, w2 = image2.shape

        # 2 images horizontally together
        img = np.zeros((max(h1, h2), w1 + w2), dtype = "uint8")
        img[0:h1, 0:w1] = image1
        img[0:h2, w1: ] = image2

        # Drawing the lines
        for i in range(len(matches)):
                # Matching points
                pt1 = (int(keypoints1[matches[i][0]].pt[0]),
    int(keypoints1[matches[i][0]].pt[1]))
                pt2 = (int(keypoints2[matches[i][1]].pt[0]) + w1,
    int(keypoints2[matches[i][1]].pt[1]))

                # Drawing mathcing points
                cv2.circle(img, pt1, 2, (0, 255, 0), 1)
                cv2.circle(img, pt2, 2, (0, 0, 255), 1)

                # Drawing line between the points
                cv2.line(img, pt1, pt2, (255, 0, 0), 1)

return img
```

Here is the resulting image after determining matching points between the two images:

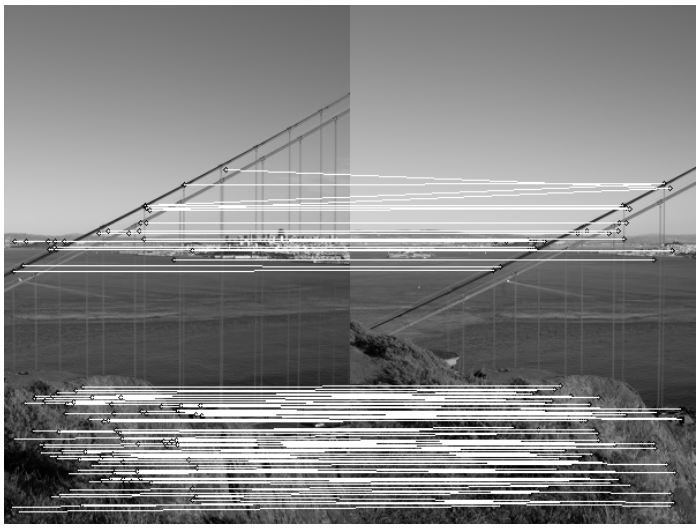

Figure 3. Matching lines between two images

Step 5: Image registration

After having matching points of two images, according to the RANdom SAmple Consense (RANSAC) algorithm, the proper affine transform for an image can be computed so that the image can be aligned/registered accordingly.

Having matching key points of two images, firstly, it is needed to find the correct homographic level between the image, which has been computed using *findHomography* function with the parameters of matching points and RANSAC methodology:

```
# Matching points of two images
 m_kps0, m_kps1 = get_matching_points(keypoints1, keypoints2, matches)

# Finding Homography using RANSAC
homography, status = cv2.findHomography(m_kps0, m_kps1, cv2.RANSAC)
```

Now, having the *homography,* the proper affine transformation can be applied to one of the images, and the other image can be simply concatenated to the resulting image:
Here the steps can be seen clearly:

Figure 4. The first image with applied affine transformation

Figure 5. Adding the second image to the beginning of the resulingt image

However, registering the images in such a way creates an issue: what if the images for the panoramic view align in a way that we would need to add the second image to the right of the resulting image (not to left as in implemented above).

To solve this problem, a function called *direction* has been implemented by taking two images as parameters:

```
def direction(image1, image2):

    # Features (Keypoints and Descriptors) & matches
    features = get_features_of_two_images(image1, image2)

    # Keypoints
    keypoints1 = features['image1'][0]
    keypoints2 = features['image2'][0]

    # Descriptors
    descriptor1 = features['image1'][1]
    descriptor2 = features['image2'][1]

    # Matching indices between two images
    matches = features['matches']

    # Matching points of two images
    m_kps0, m_kps1 = get_matching_points(keypoints1, keypoints2, matches)

    # Finding Homography using RANSAC
    homography, status = cv2.findHomography(m_kps0, m_kps1, cv2.RANSAC)

    # Creating result image
    img = cv2.warpPerspective(image1, homography, (image1.shape[1] +
image2.shape[1], image1.shape[0]))

    # Calculating the percentage of black area in the affined image
    all_0_count = list(img.flatten()).count(0)
    black_region_0_count = list(img[0:image2.shape[0],
image2.shape[1]:].flatten()).count(0)
    percentage = (black_region_0_count / all_0_count) * 100

    return percentage
```

The logic behind the function is as follows: the function checks the percentage of the black region after aligning the images in different orders. Having more black region means that adding the second image to the left at the same time having the first image on the left too, which means that we are in a wrong direction. As a result, we need to choose the proper order in which the percentage coming from the function is smaller than the other:

```
dir1, dir2 = direction(images[0], images[1]), direction(images[1], images[0])
if dir1 > dir2: return 1
elif dir1 < dir2: return 0
```

If the *dir* returned as 1, we need to reverse our list of images so that we can still align the second image passed to the function of *register_images* on the left of the resulting image.

And finally after registering the images in the list, the result image is shown as following:

Figure 6. Image alignment final version (Golden Gate)

Figure 7. Image alignment final version (Fishbowl)

Note: SIFT descriptor has been used in determining the direction.

Step 6: Blending

In order to apply blending, it was necessary to define the overlapped area between the images. To implement this, after applying affine transformations to the first image (result like in Figure 4), the image has been iterated to ignore the black area around the image.

```
def find_overlap(img, img2):
    L = len(img)
    x1_up = 0
    for i in range(0, L):
        if img[20, i] != 0:
            x1_up = i
            break
        img[0, i] = 255

    x1_down = 0
    for i in range(0, L):
        if img[img.shape[0] - 20, i] != 0:
            x1_down = i
            break
        img[img.shape[0] - 1, i] = 255

    cv2.imwrite('test.png', img)

    x1 = min(x1_up, x1_down)
    x2 = img2.shape[1]
    y1 = 0
    y2 = img.shape[0]
```

```
return x1, y1, x2, y2
```

Iterating through a reasonable point from above and below (20 pixel from above, 20 pixel from below), two different x points has been got; `x1_up , x1_down.` To reduce the effect of the black region in the average the minimum of these two points has been selected as the x point for the first vertical edge of the overlapped region:

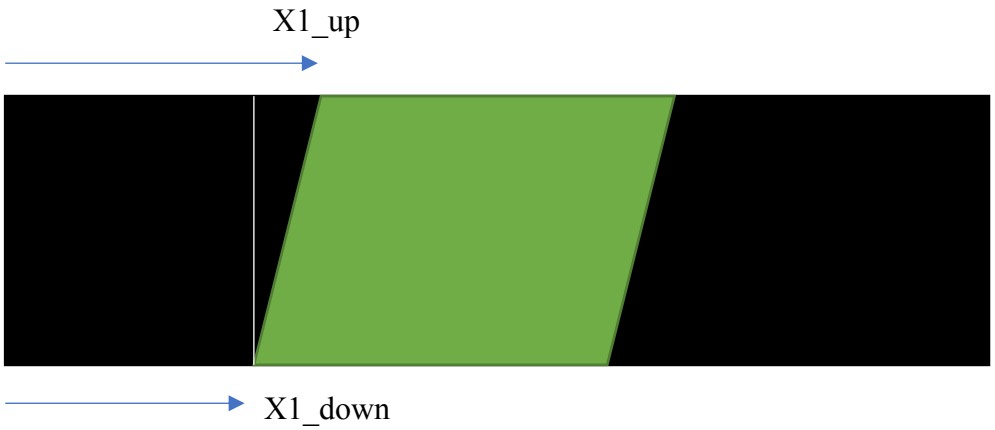

Figure 8. Determining overlapped region 1

After finding the first x value, then it is easy to determine the second x value for the other edge of the overlapped region. It is basically equal to the width of the second image:

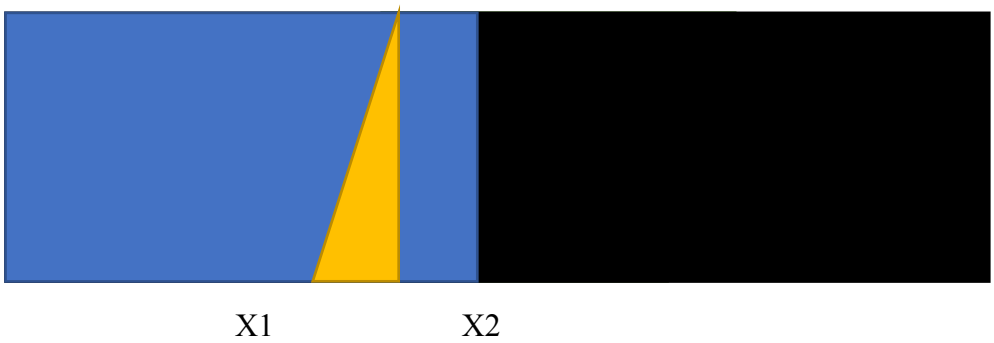

Figure 9. Determining overlapped region 2

Having obtained the x values, we got the overlapped region for both images, so we can apply the blending algorithms. For the first techniques, we just add both overlapped parts and take an average:

```
avg = im1 * 0.5 + im2 * 0.5
```

For the second technique, we iterate through the width of the overlapped region (d), and apply weighted average starting from the 1 and 0, and updating by the value of 1 / d; as a result, we get a soft transition between the images:

```
i = 0
inc = 1
avg = np.zeros((im1.shape))
while i < d:
    avg[:, i] = im1[:, i] * (1 - inc) + im2[:, i] * inc
    inc -= (1 / d)
    i += 1
```

Here are the blended versions of stitching:

Figure 10. Blending Equal Weighted Average

Figure 11. Blending Linear Weighted Average

Works Cited

[1] OpenCV, "cv::KeyPoint Class Reference," Open Source Computer Vision, [Online]. Available: https://docs.opencv.org/3.4/d2/d29/classcv_1_1KeyPoint.html#aee152750aa98ea54a48196a937197095. [Accessed 19 April 2019].

[2] xaedes, "StackOverFlow," [Online]. Available: https://stackoverflow.com/questions/11627362/how-to-straighten-a-rotated-rectangle-area-of-an-image-using-opencv-in-python. [Accessed 18 April 2019].