CSCI 677 Homework 3

Jingyun Yang

Student ID: 8357011225

1 Introduction

In this assignment, we write and execute a program to locate desired objects in given images by using SIFT features to find feature matches and using RANSAC to fit homography matrices. We show the results of our program by presenting intermediate and final outputs of the algorithm and performing analysis on these outputs. In section 2, we provide a summary of the program we write, including the directory structure and all the sources used in our program; in section 3, we show the outputs produced by our program on the given input image data; in section 4, we provide insights to our results by qualitatively analyzing them.

2 Summary of Program

2.1 Directory Structure

Under the program directory, `hw3.py` is the main python program that reads the input images, performs feature mapping and homography matrix computation, and outputs statistics and results to directory named `out`; `utils.py` is a python file containing useful helper functions for the main program. `HW3_Data` is the provided folder for input image data. The directory tree, with `hw3` as the base directory name, is shown below.

```
hw3/
|-- hw3.py
|-- utils.py
|-- out/
|-- HW3_Data/
```

2.2 Executing the Program

To execute the program, run `python3 hw3.py --imgdir=HW3_Data --outdir=out` in the base directory. Outputs will be saved to directory named `out`.

2.3 Source Listing

Now we list the source files for our program.

```
''' hw3.py '''
import numpy as np
import cv2
import os
import glob
import argparse
from utils import mkdir, show_img, get_corner_coords, dist_kp

''' Compute sift features and then compute homography transformation using RANSAC.

Args:
    query_img: the image to query in a scene (src_img)
    train_img: the scene that contains the query img (dst_img)
```

```
    f: log file to write program results

Returns:
    img_query_features: query image with sift features
    img_train_features: train image with sift features
    img_matches_before: image showing matches before homography matrix is computed
    img_matches_after: image showing matches after homography matrix is computed
    h: computed homography matrix
'''
def compute_homo_transformation(query_img, train_img, f=None):
    # initiate SIFT detector
    sift = cv2.xfeatures2d.SIFT_create()

    # find the keypoints and descriptors with SIFT
    kp_query, des_query = sift.detectAndCompute(query_img, None)
    kp_train, des_train = sift.detectAndCompute(train_img, None)

    n_features_query = len(kp_query)
    n_features_train = len(kp_train)

    if f:
        f.write('Number of features for query image: {}\n'.format(n_features_query))
        f.write('Number of features for query image: {}\n'.format(n_features_train))

    # convert image colors to grey so that sift features can be drawn on top
    gray_query_img = cv2.cvtColor(query_img, cv2.COLOR_BGR2GRAY)
    gray_train_img = cv2.cvtColor(train_img, cv2.COLOR_BGR2GRAY)

    # draw sift features on gray image
    draw_kp_params = dict(outImage=np.array([]), flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
                                                       )
    img_query_features = cv2.drawKeypoints(gray_query_img, kp_query, **draw_kp_params)
    img_train_features = cv2.drawKeypoints(gray_train_img, kp_train, **draw_kp_params)

    # BFMatcher with variable k parameter
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des_query, des_train, k=2)

    # get number of matches
    n_matches = len(matches)

    # filter only good matches using ratio test
    matches = filter(lambda f: f[0].distance < 0.7 * f[1].distance, matches)
    matches = map(lambda f: [f[0]], matches)

    # sort matches by distance
    matches = sorted(matches, key=lambda f: f[0].distance)

    # get number of matches and number of good matches
    n_good_matches = len(matches)

    if f:
        f.write("Number of matches: {}\n".format(n_matches))
        f.write("Number of good matches: {}\n".format(n_good_matches))

    # draw first 20 matches
    draw_params = dict(matchColor=(0, 255, 0), singlePointColor=None, outImg=np.array([]), flags=
                                                         2)
    img_matches_before = cv2.drawMatchesKnn(query_img, kp_query, train_img, kp_train,
                    matches[:20], **draw_params)

    # extract points in the matches
    matched_points_query = np.vstack([kp_query[match[0].queryIdx].pt for match in matches])
    matched_points_train = np.vstack([kp_train[match[0].trainIdx].pt for match in matches])

    # find homography between query and train image using RANSAC algorithm
    h, status = cv2.findHomography(matched_points_query, matched_points_train, cv2.RANSAC)
```

```
    # write
    if f:
        f.write('h Matrix:\n')
        f.write(str(h))
        f.write('\n')

    # calculate boundaries of query image after homography transformation
    corners = get_corner_coords(query_img.shape)
    transformed_corners = cv2.perspectiveTransform(np.float32([corners]), h).astype(np.int32)
    train_img_with_transformed_corners = cv2.polylines(train_img, transformed_corners, True, (246
                                                    , 148, 59), 4)

    # caculate key points after homography transformation
    kp_warp = cv2.perspectiveTransform(np.array([[[kp.pt[0], kp.pt[1]] for kp in kp_query]]), h)[
                                                    0]
    kp_warp = np.array([cv2.KeyPoint(pt[0], pt[1], 1) for pt in kp_warp.tolist()])

    # filter matched key points
    matches_after_homo = list(filter(
        lambda match: dist_kp(kp_warp[match[0].queryIdx], kp_train[match[0].trainIdx]) < 1,
        matches
    ))

    # get total number of matches consitent with the computed homography
    n_consistent_matches = len(matches_after_homo)

    if f:
        f.write("Number of matches consistent with homography: {}\n".format(n_consistent_matches)
                                                        )

    # show top 10 matches after homography transformation
    img_matches_after = cv2.drawMatchesKnn(query_img, kp_query,
                    train_img_with_transformed_corners, kp_train,
                    matches_after_homo[:10], **draw_params)

    return img_query_features, img_train_features, img_matches_before, img_matches_after, h

def main():
    # setup argparser
    parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument('--imgdir', type=str, default=None, help='directory to source images.')
    parser.add_argument('--outdir', type=str, default=None, help='directory to save output data.'
                                                    )
    parser.add_argument('--show_img', help='show image results', action='store_true')
    args = parser.parse_args()

    # make out directory if not already done
    mkdir(args.outdir)

    # open log file
    f = open(os.path.join(args.outdir, 'logs.txt'), 'w')

    # get all src and dst files
    src_files = glob.glob(os.path.join(args.imgdir, 'src_*.jpg'))
    dst_files = glob.glob(os.path.join(args.imgdir, 'dst_*.jpg'))

    # loop through all combinations of source and destination files
    for i, src_file in enumerate(src_files):
        for j, dst_file in enumerate(dst_files):
            # read image files
            src_img = cv2.imread(src_file)
            dst_img = cv2.imread(dst_file)

            # get image file names without extension
            src_filename = src_file.split('/')[-1][:-4]
            dst_filename = src_file.split('/')[-1][:-4]

            f.write('Experiment: src image {} and dst image {}.'.format(i + 1, j + 1))
```

```
            # run SIFT and RANSAC on given images
            q_feature, t_feature, match_b, match_a, h = compute_homo_transformation(src_img,
                                                                dst_img, f=f)

            # show annotated images if needed
            if args.show_img:
                show_img('SIFT Features for Query Image', q_feature)
                show_img('SIFT Features for Train Image', t_feature)
                show_img('Top 20 Matches before Homography', match_b)
                show_img('Top 10 Matches after Homography', match_a)

            # save images to file
            cv2.imwrite(os.path.join(args.outdir, 'img{}{}_qf.jpg'.format(i, j)), q_feature)
            cv2.imwrite(os.path.join(args.outdir, 'img{}{}_tf.jpg'.format(i, j)), t_feature)
            cv2.imwrite(os.path.join(args.outdir, 'img{}{}_mb.jpg'.format(i, j)), match_b)
            cv2.imwrite(os.path.join(args.outdir, 'img{}{}_ma.jpg'.format(i, j)), match_a)

            f.write('\n')

    # close log file
    f.close()

if __name__ == '__main__':
    main()
```

```
''' utils.py '''
import os
import cv2
import numpy as np

''' Safe mkdir that checks directory before creation. '''
def mkdir(dir):
    if not os.path.exists(dir):
        os.makedirs(dir)

''' Display an image in a window with given name. '''
def show_img(window_name, img):
    cv2.imshow(window_name, img)
    cv2.waitKey(0)
    cv2.destroyWindow(window_name)

''' Compute a 4x2 numpy matrix of corner coords of an image. '''
def get_corner_coords(shape):
    h, w, _ = shape
    return np.array([
        [0, 0],
        [w, 0],
        [w, h],
        [0, h]
    ])

''' Compute the distance between two key points. '''
def dist_kp(kp1, kp2):
    return np.linalg.norm(np.array(kp1.pt) - np.array(kp2.pt))
```

3 Results

3.1 SIFT Features

The detected SIFT features for each destination and source image is shown in Figure 1 and fig:sift2, respectively. For destination images, **1726**, **1626**, and **1654** SIFT features are found for `dst_1.jpg`, `dst_2.jpg`, and `dst_3.jpg`, respectively. For source images, **3248** and **2807** SIFT features are found for `src_1.jpg` and `dst_2.jpg`, respectively.

Figure 1: SIFT Features for Destination Images

Figure 2: SIFT Features for Source Images

3.2 Feature Matches before RANSAC

We used "brute force" matcher to calculate the feature matches between source images and destination images. The top 20 matches found by the matcher for each pair of source and destination images are shown in Figure 3.

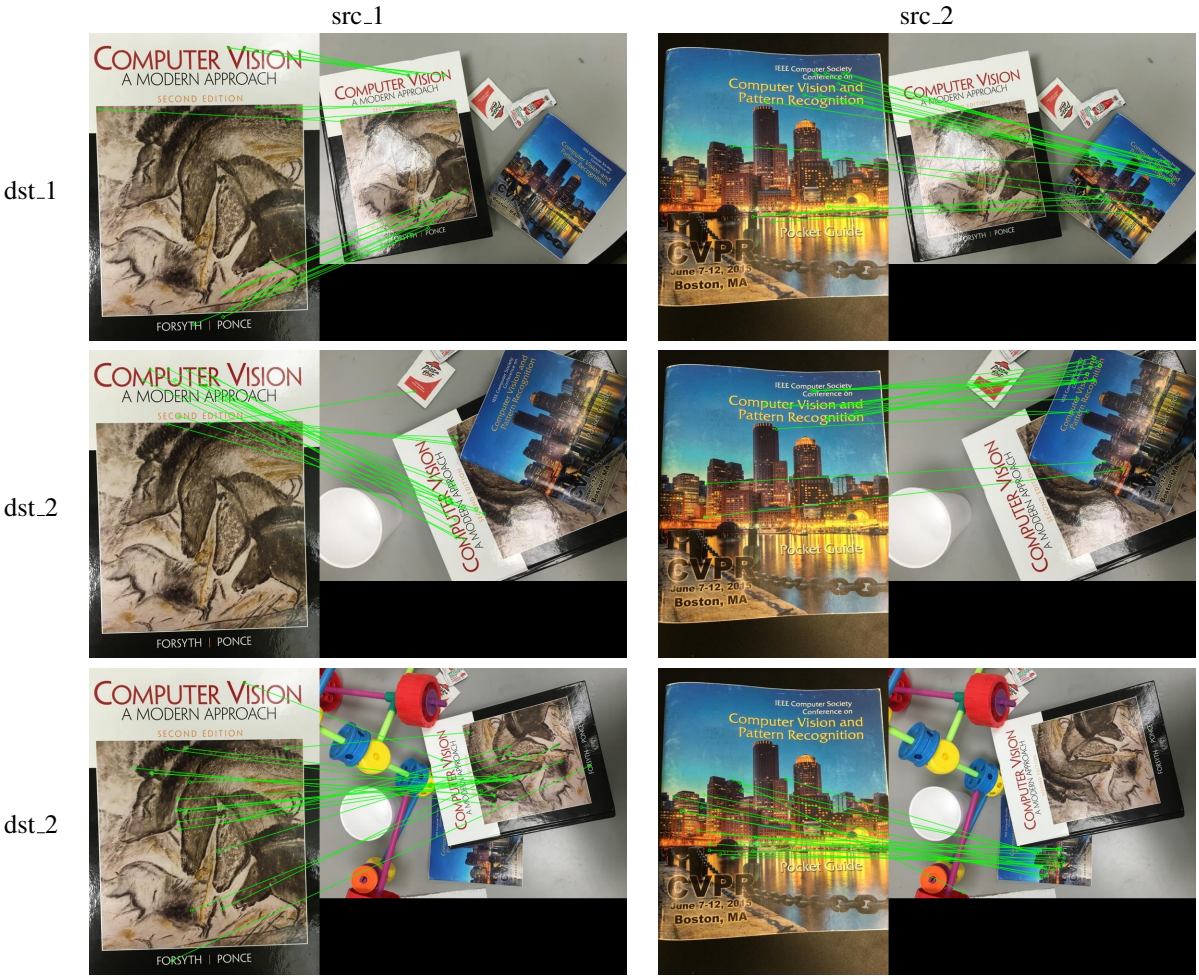

Figure 3: Top 20 Scoring Matches Found by the Matcher before RANSAC

3.3 Feature Matches after Homography Computation

After homography matrix is computed, we show the top 10 matches for each pair of images. Also, we draw light blue boxes in each train image representing the border of the query image after homography transformation. We show our results in Figure 4.

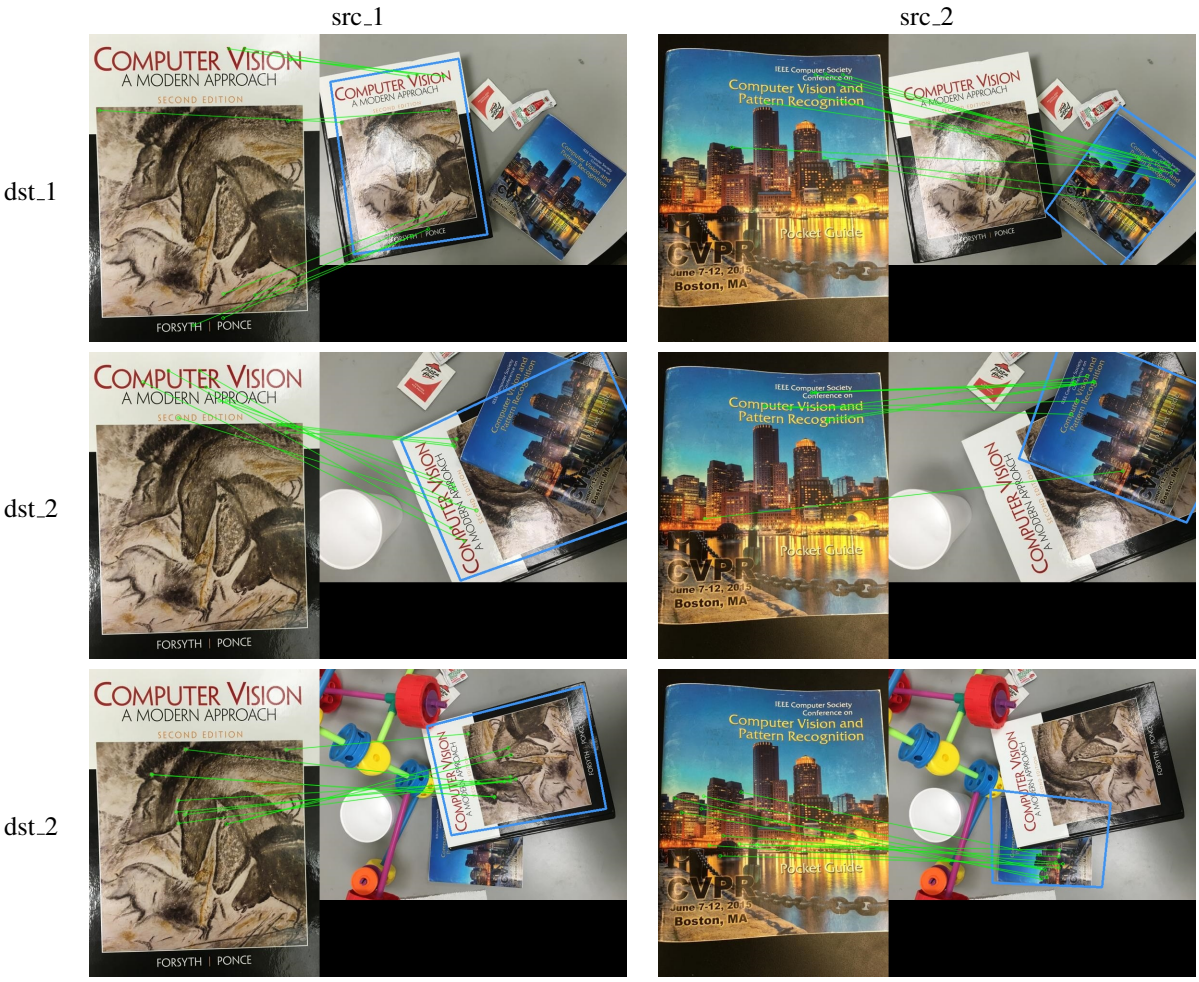

Figure 4: Top 10 Scoring Matches Found by the Matcher after RANSAC

3.4 Quantitative Results for Feature Matching

We consider a matching to be consistent with the homography matrix if transformed coordinates of the destination image in the matching is within 1 pixel Euclidean distance away from the coordinates of the source image in the matching. Denote m as the total number of matches found by the matcher, g as the number of good matches after performing ratio test with ratio 0.7, and c as the number of matches consistent with the computed homography matrix. We report all m, g, and c values for our experiments in Table 1.

Source Image Index	Destination Image Index	m	g	c
1	1	3248	417	354
1	2	3248	235	171
1	3	3248	564	511
2	1	2807	197	152
2	2	2807	210	152
2	3	2807	47	34

Table 1: Quantitative Results for Feature Matching

3.5 Homography Matrices

The homography matrices found by our program are shown in Table 2.

	dst_1	dst_2	dst_3
src_1	$\begin{pmatrix} 0.5414 & 0.0730 & 22.2714 \\ -0.1020 & 0.5335 & 99.6409 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} -0.2541 & 0.5320 & 293.1227 \\ -0.6056 & -0.3089 & 476.6988 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} -0.1176 & 0.4076 & 274.4636 \\ -0.4768 & -0.1330 & 345.3537 \\ 0 & 0 & 1 \end{pmatrix}$
src_2	$\begin{pmatrix} 0.2559 & -0.1638 & 482.7577 \\ 0.2114 & 0.4310 & 150.2802 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0.1324 & 0.6016 & 271.7112 \\ -0.5611 & 0.2396 & 222.3588 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} -0.0506 & 0.4215 & 216.7896 \\ -0.4547 & 0.0817 & 444.5762 \\ 0 & 0 & 1 \end{pmatrix}$

Table 2: Computed Homography Matrices

4 Qualitative Analysis

Overall, the program works decently in finding matches between images and fitting homography matrices. Through the outputs of our program, we can see that our program is robust enough to cope with the situation in which there is presence of other objects in the target images. It can be also noticed that our program can also deal with images in which the matching query image has various sizes and orientations. The problem we noticed in the output of our program is that our algorithm is not very robust against scenes with occlusions of the objects of interest. In particular, in the cases of `(src_1, dst_2)` and `(src_2, dst_3)`, the ratio between the number of consistent matches and the total number of matches found is much lower than those in other cases in which no occlusion of the object of interest is present. This might be due to the fact that when occlusions exist, it is harder to find matching SIFT features because part of the object of interest is hidden from view.