Angelos Nikolaou
Prathmesh Madhu
24.05.2021

The deadline for this exercise is on Monday 7.06.2021, 23:59

Panorama Stitching

- You will need to install the following libraries for this exercise

  `'pip install scipy fargv'`

  When you are finished with this exercise,

- Compress the complete directory into a ZIP and upload it to StudOn. Make sure folder *results/ex2* exists in the project root. On Unix (and powershell) systems, the following command generates the submission archive

```
cd ComputerVision2021 (or CV2021);
# as a single line:
zip ../submission_ex2.zip ./data/ex2/*.jpg ./bin/run_ex2.py
./ex2/*py test/test_ex2.py ./results/ex2/*png
```

- For groups, one member uploads the submissions and adds their partner to the exercise. Don't forget to add all members of the team in *ex2/__init__.py*

- We are not grading based on vectorization in this exercise. However, your code should not take more than 1 minute to run for your own convenience.

1 Introduction

In this exercise you are asked to implement an application that stiches multiple images into a single panorama picture. We will use image features and local descriptors to find corresponding points in image pairs and compute a perspective transformations between the 3D *image planes*. In a last step, all images are warped to the same reference *image plane* and displayed on the screen.

After you have implemented all tasks, feel free to run the application with your own panorama images. If you are using smartphone images, make sure to scale down the input to an appropriate size. The skeleton code already loads a default dataset that produces the result below after you have completed this exercise.

Figure 1: Panorama of the 'Red Square' from nine input images.

2 Feature Extraction [10%]

Implement the function `extract_features` in `ex2/functions.py`.

To compute a homographic transformation between a pair of images, it is required to have at least four correct point to point correspondences. These correspondences can be obtained by matching the feature descriptors of the computed keypoints. In this exercise, we use `ORB` keypoints and descriptors. The ORB feature descriptors are uchar vectors with 32 elements.

3 Feature Matching [25%]

Implement the function `filter_and_align_descriptors` in `ex2/functions.py`, which finds the significant matches between descriptors in the source and destination image and returns their locations.

3.1 k-Nearest Neighbor Search

Matching feature descriptors breaks down to a nearest neighbor search on the feature vectors. In the next task we will also implement an outlier detection technique that uses the ratio of nearest, to the second nearest neighbor. The problem of finding more than one nearest neighbor is called *kNN search*.

The output is a tuple of numpy arrays both sized [N x 2] representing the similar point locations. In order to realize the *kNN* search, you should compute a distance matrix between every keypoint descriptor in source and the destination image. The most suitable distance would be Hamming distance.

3.2 Descriptor-Outlier Removal

Given the two nearest neighbors for a descriptor src image, a good way of identifying outliers is to compute the ratio between the distances d_0 and d_1. If this ratio is above a threshold t_r, this match is considered ambiguous and classified as a descriptor-outlier.

4 Homography [25%]

Implement `compute_homography` in `ex2/functions.py`.

Note: While working on homography, we always use homogenous coordinates. A homogenous coordinates system has an extra dimension that contains the length of the unit vector. This makes 2D points lie in a 3D space. Any transformation is expressed as 3x3 matrix on these coordinates.

A homography is a perspective transformation that maps planes to planes in a three dimensional space. We will compute the homography that transform the *image plane* of one camera to the *image plane* of an other camera. These transformations allow us to project all image to the same *image plane* and to create the final panorama image.

Figure 2: Computing the pairwise perspective transformations lets us warp every image into every other image.

The homography can be computed from four or more feature matches $\{(p_1, q_1), (p_2, q_2)(p_3, q_3), (p_4, q_4)\}$ by solving the following linear system of equations:

$$A = \begin{bmatrix} -p_{x1} & -p_{y1} & -1 & 0 & 0 & 0 & p_{x1}q_{x1} & p_{y1}q_{x1} & q_{x1} \\ 0 & 0 & 0 & -p_{x1} & -p_{y1} & -1 & p_{x1}q_{y1} & p_{y1}q_{y1} & q_{y1} \\ \cdots & & & & & & & & \\ \cdots & & & & & & & & \\ \cdots & & & & & & & & \\ \cdots & & & & & & & & \\ -p_{x4} & -p_{y4} & -1 & 0 & 0 & 0 & p_{x4}q_{x4} & p_{y4}q_{x4} & q_{x4} \\ 0 & 0 & 0 & -p_{x4} & -p_{y4} & -1 & p_{x4}q_{y4} & p_{y4}q_{y4} & q_{y4} \end{bmatrix}$$

$$Ah = 0$$

$$H = \begin{bmatrix} h_0 & h_1 & h_2 \\ h_3 & h_4 & h_5 \\ h_6 & h_7 & h_8 \end{bmatrix}$$

Note that, we can always use more than 4 points to over define a homography.

Compute the homography H following these steps:

1. Fill the $(>= 8) \times 9$ matrix A.
2. Compute the SVD of $A = U\Sigma V^T$
3. Compute h as the 9-th column vector of V.
4. Create the matrix H from h.
5. Normalize H by multiplying with $1/h_8$

5 RANSAC algorithm [35%]

Implement `_get_inlier_count` [10%] and `ransac` [25%] in `ex2/functions.py`.

Even after aligning and filtering in 3.2, some of the feature matches are outliers. If one of the four matches used to compute the homogprahy is an outlier, the resulting transformation matrix is incorrect.

5.1 RANSAC Inliers

The function `_get_inlier_count` computes the number of inlier matches given a set of matching point hypotheses (two numpy arrays), homography matrix and the threshold in pixels. Note that, when applying a perspective transformation a 2D image point must be temporary lifted into homogeneous space.

5.2 RANSAC

We will use the RANSAC Inliers and the RANSAC algorithm to achieve a robust result as follows:

1. filter and align descriptors
2. initialize the optimization loop - already done!
3. optimization loop - already done!
   a) select a random subset of at least 4 points
   b) compute homography for these selected random points
   c) See if that's better than anything you found and store it.
4. return the best homography

Since RANSAC is not a deterministic algorithm, small variations are to be expected. However, we provide an example console output as follows. Note that +- 10% of these numbers are a sign for a proper solution.

```
Step:   999   165 RANSAC points match!
Step:   999   118 RANSAC points match!
Step:   999   160 RANSAC points match!
Step:   999   132 RANSAC points match!
Step:   999   160 RANSAC points match!
Step:   999   184 RANSAC points match!
Step:   999   159 RANSAC points match!
Step:   999   141 RANSAC points match!
```

6 Stitching [5%]

Implement `translate_homographies` in `ex2/functions.py`.

Overall panorama stitching can be seen as having three stages:

1. **Compute homographies between consecutive images.** All previous parts of the exercise were implementing this stage.

2. **Compute homographies from any image to a common reference.** This stage is realized by function `probagate_homographies`. The reason for not computing directly the homographies between any image and the reference image, is that the correct estimate of a homography depends on the number of matching points between the images. Homographies can be correctly estimated with RANSAC only between images with a high overlap, images that are near. Because homographies are linear operations (multiplication with a 3x3 matrix), we use the commutative property to compute a homography between any two images as long as there exists a chain of homographies linking them.

3. **Project all images in the common reference and select the appropriate pixel.** This stage is realized by many functions but you only have to implement `translate_homographies`.

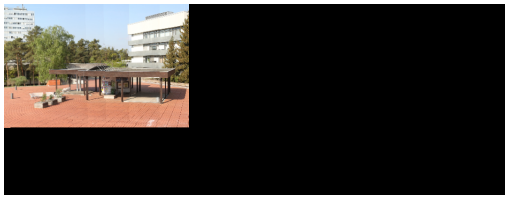
In `translate_homographies` function we want you to modify (deep copy) a dictionary where values are homographies $H : (x, y, 1) \mapsto (x\prime, y\prime, 1)$ to a dictionary were those values are $H : (x, y, 1) \mapsto (x\prime + dx, y\prime + dy, 1)$.

This function is used to move the homographies of the panorama before it is rendered so that it is all rendered inside the destination image.

In Fig. 6, we can see how panorama look when `translate_homographies` does nothing.

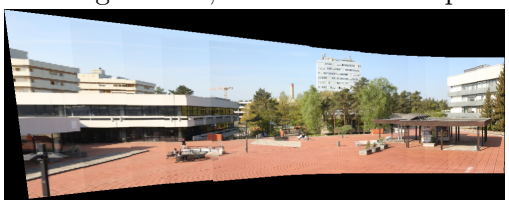
Hint: We can easily see that `translate_homographies` is not as important when the reference image is on the left, use the left image for a reference and return the input homographies until your implementation works correctly to render nice panoramas.

Reference: left-most, Translation unimplemented



Reference: right-most, Translation unimplemented

Reference: left-most, Translation implemented



Reference: right-most, Translation implemented

Figure 3: Effect of reference image selection with `translate_homographies` doing nothing and working correctly.