Panorama Stitching

If you are working on your own computer, you will need to install the libraries scipy and fargv for this exercise:

```
pip install scipy fargv
```

The CIP pool computers have all necessary packages installed if you load the CV module.

1 Introduction

In this exercise, you are asked to implement an application that stitches multiple images into a single panorama picture. We will use image features and local descriptors to find corresponding points in image pairs and compute 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

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 key-points. In this exercise, we use `ORB` key-points and descriptors. The ORB feature descriptors are `uchar` vectors with 32 elements.

Implement the function `extract_features` in `utils/functions.py`. Use the OpenCV function `cv.ORB_create` and find other helpful OpenCV functions to detect the key-points.

3 Feature Matching

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

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

3.2 Descriptor-Outlier Removal

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

4 Homography

Note: While working with homographies, we always use homogenous coordinates. A homogenous coordinate 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 a 3×3 matrix on these coordinates.

Implement the function `compute_homography`. A homography is a perspective transformation that maps planes to planes in a three dimensional space. We will compute the homography that transforms the *image plane* of one camera to the *image plane* of another camera. These transformations allow us to project all images 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 is computed from four feature matches $\{(p_1, q_1), (p_2, q_2), (p_3, q_3), (p_4, q_4)\}$ (or more) by solving the following linear system of equations:

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

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

$$Ah = 0$$

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. Create the matrix H from h, where h is the 9-th column vector of V. Normalize H by multiplying with $1/h_8$

5 RANSAC algorithm

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

5.1 RANSAC Inliers

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

5.2 RANSAC

Implement the function `ransac`. We will use the RANSAC inliers and the RANSAC algorithm to achieve a robust result as follows:

1. Filter and align descriptors
2. In an optimization loop:
   a) Select a random subset of at least 4 points
   b) Compute the homography for these selected random points
   c) See if this homography performs better than any prior homography and, if so, store it.
3. 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.

```
After 1000 steps: 164 RANSAC points match!
After 1000 steps: 182 RANSAC points match!
After 1000 steps: 162 RANSAC points match!
After 1000 steps: 137 RANSAC points match!
After 1000 steps: 142 RANSAC points match!
After 1000 steps: 169 RANSAC points match!
After 1000 steps: 142 RANSAC points match!
After 1000 steps: 171 RANSAC points match!
```

6 Stitching

Overall, panorama stitching can be seen as having three stages:

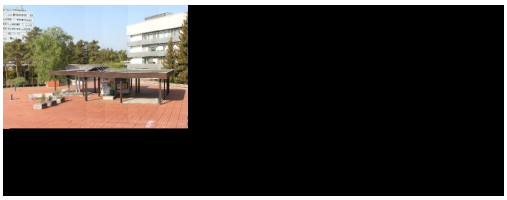
1. **Compute homographies between consecutive images.** All previous parts of the exercise were implementing this stage.

2. **Compute homographies from any image to a common reference.** This stage is already realized by the function `propagate_homographies`. The reason for not directly computing 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 between two images can only be estimated correctly with RANSAC if the images overlap significantly, i.e., images that are close. Because homographies are linear operations (multiplication with a 3×3 matrix), we exploit 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 this function, we want you to change a dictionary where the values are homographies $H : (x, y, 1) \mapsto (x\prime, y\prime, 1)$ to a dictionary where those values are $H : (x, y, 1) \mapsto (x\prime + dx, y\prime + dy, 1)$. Do not modify the original dictionary directly but make a deep copy and change and return the copy.

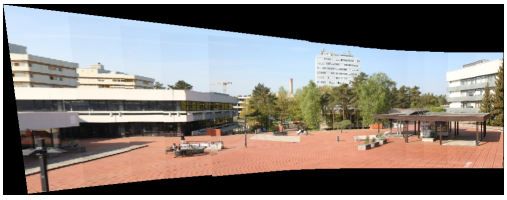
   This function is used to offset the homography matrices so that the final stitched image is centered within the destination image. See the effect of the function in the Figure below.

Reference: 2nd left-most, Translation unimplemented



Reference: right-most, Translation unimplemented

Reference: 2nd left-most, Translation implemented



Reference: right-most, Translation implemented