**Darius Rückert**
**AmirAbbas Davari**
**7.05.2018**

**The deadline for this exercise is on Sunday 20.05.2018, 23:59**

# Panorama Stitching

## 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 should produce the result below.

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

## 2 Feature Matching [5 Points]

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 and are stored row-wise in the matrix `ImageData::descriptors`.

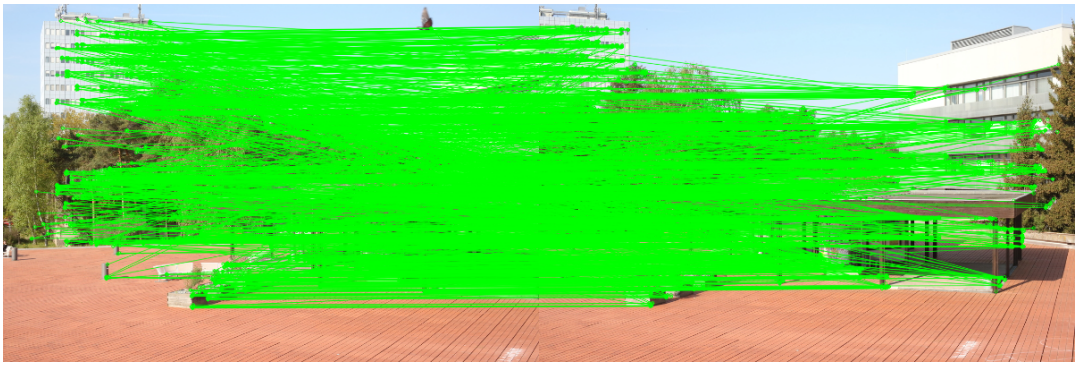

Figure 2: Feature matches before the ratio test.

### 2.1 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*.

Implement the function `matchknn2` in `matching.cpp`, which finds the two nearest neighbors of an image 1 descriptor in image 2. The output is a vector of vector of `DMatch`, with dimensions $n \times 2$, where n is the number of descriptors in image 1. The distance between two descriptors can be computed with:

```
double distance = norm(descriptors1.row(i), descriptors2.row(j),
    NORM_HAMMING);
```

The `DMatch` objects are used to identify specific matches and are created like this:

```
DMatch dm(i,j,distance);
```

### 2.2 Outlier Removal

Given the two nearest neighbors for a descriptor in image 1, 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 an outlier.
Implement the function `ratioTest` in `matching.cpp`. Add the match, belonging to the nearest neighbor to the `matches` array, only if the distance ratio is smaller than `ratio`.

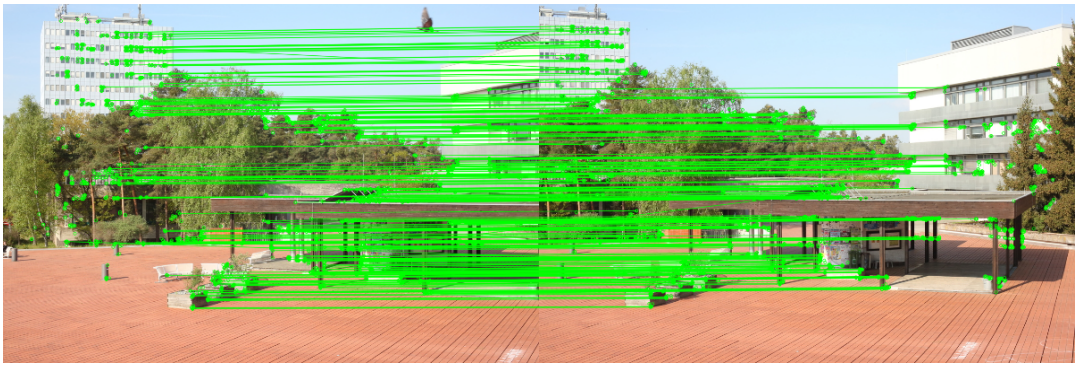

Figure 3: Feature matches after the ratio test.

Console output:

```
(0,1) found 557 matches.
(0,1) found 528 RANSAC inliers.
(1,2) found 577 matches.
(1,2) found 534 RANSAC inliers.
(2,3) found 581 matches.
(2,3) found 543 RANSAC inliers.
(3,4) found 536 matches.
(3,4) found 506 RANSAC inliers.
(4,5) found 589 matches.
(4,5) found 559 RANSAC inliers.
(5,6) found 630 matches.
(5,6) found 581 RANSAC inliers.
(6,7) found 629 matches.
(6,7) found 590 RANSAC inliers.
(7,8) found 625 matches.
(7,8) found 593 RANSAC inliers.
```

## 3 Homography [5 Points]

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

The homography can be computed from four 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:

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

$$Ah = 0$$

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

Implement the function `computeHomography` in `homography.cpp`. 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 $1/h_8$

At the beginning of `main` a small test is executed to validate your implementation. Make sure this test succeeds before you start with the next task!

## 4 RANSAC [5 Points]

Even after applying the ratio test in task 2.2 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. We will use the RANSAC algorithm to achieve a robust result:

1. Select a random minimal (4 matches) subset of all matches
2. Compute $H$ with this subset
3. Count the number of inliers that satisfy $|p - Hq| < t_r$
4. Remember the best $H$ and goto 1.

### 4.1 RANSAC Inliers

Implement the function `numInliers` in `ransac.cpp` that computes the number of inlier matches given a 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.

### 4.2 Random Sampling

Implement the rest of `computeHomographyRansac` in `ransac.cpp`.

- Compute a random subsets of the input points. You can obtain random numbers in the correct range by calling:

```
int randomNumber = dis(gen);
```

- Compute the homography for the subsets and the number of inliers
- Store the best homography in the variable `bestH`