Structure from Motion

If you are working on your own computer, you will need to install the library tqdm for this exercise:

```
pip install tqdm
```

1 Introduction

In this exercise, we will compute the relative transformation of two cameras, i.e., rotation and translation, and the 3D position of a set of feature matches. Consider Fig. 1 that shows a *memorial shield* captured from two different angles. Given a point x in image 1 and the corresponding match x' in image 2, the following constraint can be derived from epipolar geometry:

$$\mathbf{x'}^T \mathbf{F} \mathbf{x} = 0, \tag{1}$$

The 3×3 matrix $\mathbf{F}$ is the fundamental matrix, mapping points in one image to epipolar lines in the other image.

Figure 1: A *memorial shield* captured from two different viewpoints at the Germanic National Museum.

2 Computation of F

Similar to the homography computation of the last exercise sheet, the fundamental matrix can be computed from 8 matching feature points. Also like previous times, this procedure is wrapped in a RANSAC algorithm (this time the function is provided to you) to account for incorrect matches.

2.1 Identification of Inliers: Distance from the Epipolar Line

As we will be using a RANSAC variant to choose the optimal fundamental matrix, it is crucial to have an estimate on the quality of a fundamental matrix hypothesis. In the following, p describes a point in the source image, and q its matching point in the destination image.

Implement the function `inliers_epipolar_constraint` in `utils/functions.py` that checks whether a given match satisfies the epipolar constraint. This is the case if the distance of q to the epipolar line of p is smaller than a threshold t_e.

- Convert points p and q to homogeneous coordinates.
- Compute the epipolar line $l = (l_x, l_y, l_w) = Fp$.
- Normalize l to get the scale-independent line $l' = l/\sqrt{l_x^2 + l_y^2}$.
- Compute the point-line distance $d = q^T l'$.
- A match is an inlier if $|d| < t_e$.

2.2 8-Point Algorithm

Given 8 correct feature matches, the fundamental matrix can be computed with the 8-point algorithm. Implement the function `compute_fundamental_matrix` in `utils/functions.py` with the following steps:

1. Construct the matrix A. (Note: 1 row per match $\rightarrow$ 8 rows in total)

$$A = \begin{bmatrix} p_{x1}q_{x1} & p_{x1}q_{y1} & p_{x1} & p_{y1}q_{x1} & p_{y1}q_{y1} & p_{y1} & q_{x1} & q_{y1} & 1 \\ p_{x2}q_{x2} & p_{x2}q_{y2} & p_{x2} & p_{y2}q_{x2} & p_{y2}q_{y2} & p_{y2} & q_{x2} & q_{y2} & 1 \\ \dots & & & & & & & & \\ \dots & & & & & & & & \\ p_{x8}q_{x8} & p_{x8}q_{y8} & p_{x8} & p_{y8}q_{x8} & p_{y8}q_{y8} & p_{y8} & q_{x8} & q_{y8} & 1 \end{bmatrix}$$

2. Solve $Af = 0$ and extract F

$$f = \begin{bmatrix} f_0 & f_1 & f_2 & f_3 & f_4 & f_5 & f_6 & f_7 & f_8 \end{bmatrix}^T$$

$$F = \begin{bmatrix} f_0 & f_3 & f_6 \\ f_1 & f_4 & f_7 \\ f_2 & f_5 & f_8 \end{bmatrix}$$

3. Make sure that $Rank(F) = 2$.
   - Compute the SVD $F = U\Sigma V^T$ with

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix}$$

   - Set $\sigma_3 = 0$
   - Recompute F with the updated Σ.
4. Normalize F.

There's a test-case in *test/test.py* to validate your 8-point implementation. This test should succeed before continuing with the next tasks.

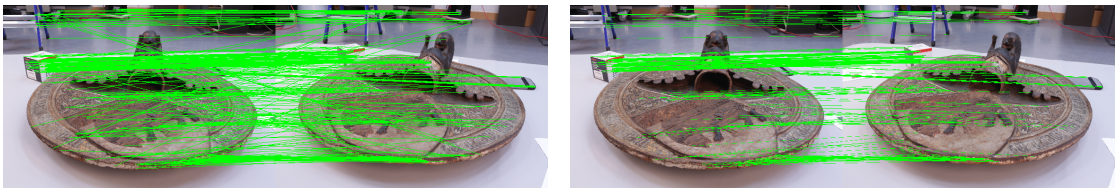

Figure 2: The *geometric consistent* matches. (left) All matches **before** passing the *ratio test* **and** satisfying the epipolar constraint; (right) All matches **after** passing the *ratio test* **and** satisfying the epipolar constraint

3 Triangulation

Triangulation describes the method of finding the position of a point in 3D space given its image in two views and the camera parameters of both views (see Figure 3).

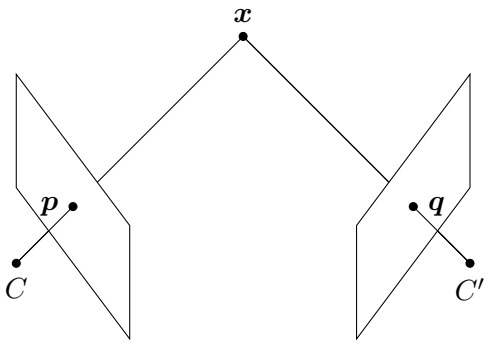

Figure 3: A point $\boldsymbol{x}$ is seen by two cameras. The rays starting from each camera center passing through the imaged points $\boldsymbol{p}$ and $\boldsymbol{q}$ intersect at the world point $\boldsymbol{x}$.

Given the pinhole camera model with the projection matrices $M_i = K_i V_i$ the projection equations

$$\boldsymbol{p} = \boldsymbol{M_1 x}$$

$$\boldsymbol{q} = \boldsymbol{M_2 x},$$

can be transformed to the homogeneous linear system of equations $Ax = 0$ with

$$\boldsymbol{A} = \begin{bmatrix} p_x \boldsymbol{m}_1^3 - \boldsymbol{m}_1^1 \\ p_y \boldsymbol{m}_1^3 - \boldsymbol{m}_1^2 \\ q_x \boldsymbol{m}_2^3 - \boldsymbol{m}_2^1 \\ q_y \boldsymbol{m}_2^3 - \boldsymbol{m}_2^2 \end{bmatrix},$$

where $\boldsymbol{m}_i^j$ is the j-th row-vector of M_i.

Implement the function `triangulate` in `utils/functions.py` with the description from above. Check if the test-case, which is executed in `test/test.py`, succeeds before you continue with the exercise.

4 Relative Transformation

To compute the relative transformation (rotation and translation) between the input images, first, the essential matrix E is computed from the fundamental matrix F and the cameras' intrinsics.

4.1 Essential Matrix

The camera positions are mathematically related via the essential matrix:

$$E = {K'}^T F K$$

where K and K' are the intrinsic camera parameter matrices of camera 1 and 2 respectively. In our case, because we are using only one camera and move it to generate a second image, $K = K'$.

Implement the function `compute_essential_matrix` in `utils/functions.py`. There is a test-case to check if the essential matrix is correct in `test/test.py`.

4.2 Decomposing Rotation and Translation Matrices

The essential matrix is then decomposed into two rotation matrices and translation vectors, giving a total number of 4 possible transformations per essential matrix. For each relative transformation, the feature matches are triangulated (Task 3) and projected to each image. Only one solution over all pose configurations is geometrically valid, which means that all 3D points lie in front of both cameras. The other configurations have at least one point that is behind one or both cameras.

Implement the function `decompose` in `utils/functions.py` with the following steps:

1. Compute the SVD $E = U\Sigma V^T$
2. The two possible rotations are

   $$R_1 = UWV^T$$

   $$R_2 = UW^TV^T$$

   with

   $$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3. The two possible translations are

   $$t_1 = s$$

   $$t_2 = -s$$

   with

   $$s = u_3 / ||u_3||$$

   and u_3 being the third column vector of U.

There is a test-case to check that the `decompose` function generates correct rotation and translation matrices in `test/test.py`. The view matrix of the second camera should (roughly) look like this:

```
View2:
[[ 0.93126 -0.15450  0.32997 -0.99363]
 [ 0.17217  0.98475 -0.02482 -0.11094]
 [-0.32111  0.07993  0.94366  0.01979]]
```

The template code visualizes the computed point cloud and the camera views as depicted in Fig. 3. Your 3D visualizations should look similar to Fig. 4:

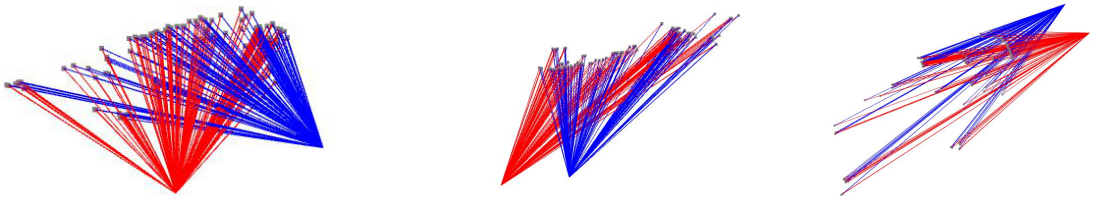

Figure 4: The two-view reconstruction after task 4. We have computed the relative transformation between both views and a set of 3D points that are seen by both cameras. The blue and red tips represent the camera positions and the points in gray represents the 3d point cloud. These 3 images represent the same scene from 3 different angles. Open3d lets you rotate the scene interactively.