Laboratory Session 3: Robust Homography and Fundamental Matrix Estimation

In this laboratory session, we will implement a robust data fitting for the automatic computation of the homography (H) and the Fundamental matrix (F) between a pair of projective images observing a 3D scene.

Goals of the assignment:

1. Feature matching based on descriptor similarity.
2. RANSAC robust approach for model fitting of the homography and the Fundamental Matrix between two views from point matches.

Evaluation of the assignment:

The resulting work will be shown by presenting the obtained results in the following Laboratory session, and the code will be submitted through the ADD (Moodle).

1. Feature matching using Nearest Neighbor distance between descriptors

We provide the file `SIFTmatching.py` where a basic SIFT extraction is performed. Subsequently the matching using the nearest neighbor distance between descriptors is computed.

Modify the threshold on maximum distance between descriptors and visualize the results. Identify the matching errors.

1. False positive matches
2. False negative matches

Select a threshold to have a low false positive rate. Focus on the chessboard where *image aliasing is evident, and discuss why it is difficult to remove the false positive* matches.

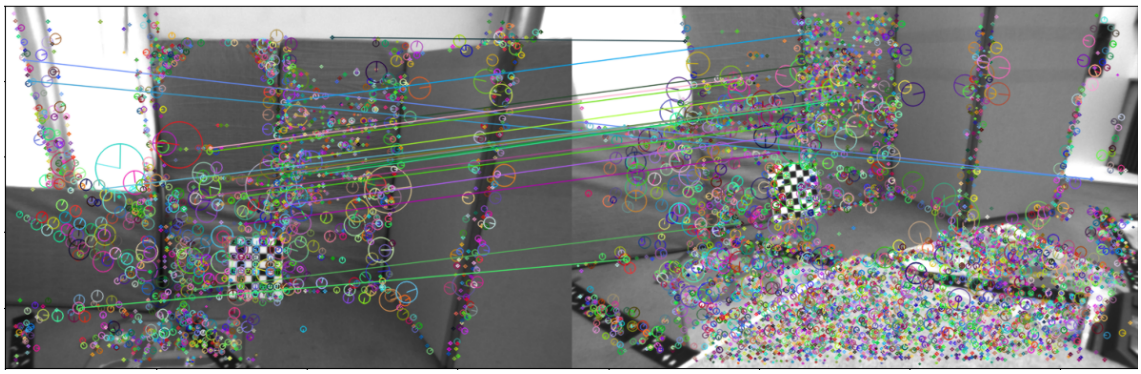

Fig.1 Example of NN matches based on SIFT features

2. Features Matching using Nearest Neighbors Distance Ratio (NNDR)

Modify the function `matchWith2NNDR` in order to match features using Nearest Neighbors Distance Ratio. For each feature of the image 1 you have to find the two features of image 2 with closer descriptor using a descriptor distance (for example L2 norm of the difference between two 128 size vectors). Consider that the two obtained minimum distances are d1 (for the minimum, the nearest neighbor) and d2 (for the second nearest neighbor). By enforcing `d1>d2*distRatio` you can reject false positive matches corresponding to image aliasing, for example in the chessboard.

```
def matchWith2NNDR(desc1, desc2, distRatio, minDist):
    """
    Nearest Neighbours Matching algorithm checking the Distance Ratio.
    A match is accepted only if its distance is less than distRatio times
    the distance to the second match.

    -input:
        desc1: descriptors from image 1 nDesc x 128
        desc2: descriptors from image 2 nDesc x 128
        distRatio:

    -output:
       matches: nMatches x 3 --> [[indexDesc1,indexDesc2,descriptorDistance],...]]
    """
```

3. Feature matching using a learning based approach (Superglue)

As alternative for SIFT features and nearest neighbors matching approaches we propose using Superglue matching. For that purpose you can install SuperGlue (see Appendix A) in the subject virtual environment CVISEnv and launch the matching demo between the two images (see Appendix A.2).

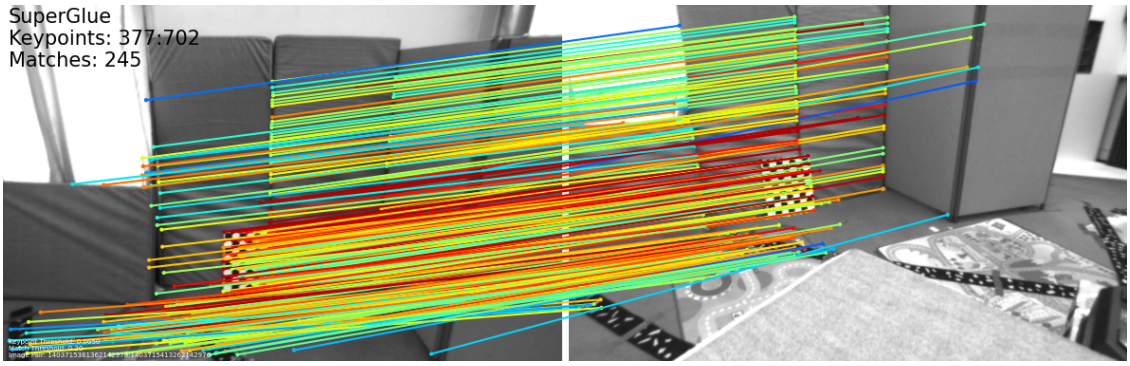

Fig.2 Example of matches based on SuperGlue

The output of the demo is a .npz file that can be loaded from your python script (see Appendix A.3).

4. RANSAC homography estimation

Implement a robust approach using a minimal solution for estimating a homography that can relate the two images and a scene plane.

We provide a file `lineRansac.py` with an implementation of a RANSAC algorithm for robustly fitting 2D lines. Use this implementation as inspiration for writing your own RANSAC algorithm for fitting the homography.

Random Sample Consensus is an algorithm for model fitting when having spurious data that could corrupt your fitting.

Instead of considering the whole set of matches for fitting the homography, the goal of RANSAC algorithm is identifying a sub-set of matches (known as inliers) that are consistent with a homography. The matches non-consistent with a homography are considered spurious matches, they are called outliers.

In RANSAC algorithm sub-sets of points defining the minimal solution are randomly chosen, i.e. 4 random points for homography. Each minimal subset produces a H hypothesis. Each hypothesis is evaluated with all the matches in the matches set. If the transfer error according to the H is below a threshold the match is inlier, otherwise outlier. We consider the number of inliers fitting each hypothesis as votes. The most voted hypothesis is the model which is more consistent with the provided data.

For the homography we recommend using as transfer error the Euclidian distance L2 between the matched point and the transformed point from the other image.

Recommendation, use `drawMatches` to display:

1. The set of matches
2. For some of the RANSAC iterations: the 4 matches producing the hypothesis, the set of matching inliers with the hypothesis.
3. The matches of the most voted hypothesis.

Identify the scene plane that defines the homography by transferring points from one image to the other.

Compare the differences between using NNDR SIFT matches or SuperGlue matches as input set.

5. RANSAC fundamental matrix estimation

Create a RANSAC algorithm to compute the fundamental matrix and the set of inliers matches. As transfer error we recommend using the distance between a point and the corresponding epipolar line in pixels.
If F is correctly estimated, the epipolar lines should intersect at the epipole, i.e. the projection of the optical center of the other image. You can project this epipole using the provided ground truth data in order to check if your estimation is correct. Compare the differences between using NNDR SIFT matches or SuperGlue matches as input set.

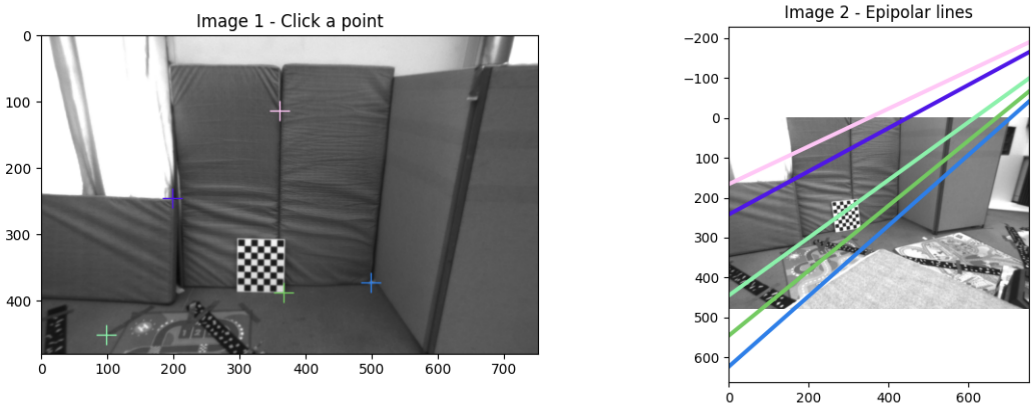

Fig.3 Example of epipolar lines

6. Guided matching using epipolar geometry (optional)

Since a match must satisfy the epipolar constraint, once you have the epipolar geometry, it can help you to reduce the rate of false negative matches by finding new matches between the two views.

Implement a guided matching that exploits the epipolar constraint to include more matches

Calibration parameters: α_x= 458.654 px, α_y= 457.296 px, x_0= 367.215 px, y_0= 248.375 px

Pose of the two cameras:

```
T_w_c1 =[[-0.0744, -0.3079,  0.9485,  0.8801],
         [-0.997 ,  0.0428, -0.0643,  0.8695],
         [-0.0208, -0.9505, -0.3102,  1.73  ],
         [ 0.    ,  0.    ,  0.    ,  1.    ]]
```

```
T_w_c2 = [[-0.512 , -0.2779,  0.8128,  0.5236],
          [-0.8586,  0.194 , -0.4745,  1.9537],
          [-0.0258, -0.9408, -0.338 ,  1.2876],
          [ 0.    ,  0.    ,  0.    ,  1.    ]]
```

Appendix A: Installation and use of SuperGlue feature matching

A.1. SuperGlue installation

You can download the SuperGlue demo from Github in a .zip file from the url https://github.com/magicleap/SuperGluePretrainedNetwork or directly through git tools (`https://gitforwindows.org/`) by using the command,

```
git clone https://github.com/magicleap/SuperGluePretrainedNetwork.git
```

A.2. Command-line SuperGlue interface

Superglue provides different demos for testing its implementation. We suggest using the "indoor pairs" example. You need to store the images you want to match in a directory and write a .txt file containing name of each pair of images in each row (see `assets/phototourism_sample_pairs.txt` as example).
Then, you can call `match_pairs.py` for launching the demo using the following command,

```
python ./match_pairs.py --resize 752 --superglue indoor --max_keypoints 2048
--nms_radius 3 --resize_float --input_dir <images_path> --input_pairs
<pairs_file_path>/euroc_sample_pairs.txt --output_dir <output_path> --viz
```

where `<images_path>` is the path where the images are stored and `<pairs_file_path>` is a path for the .txt file describing the pairs.
The demo save the output data in a `.npz` file which is treated as a dictionary by python. When activating the `–viz` option a plotting of the matches is also stored.

A.3. Loading the SuperGlue output file

The `.npz` file is a python standard that allows you accessing the superGlue matches and keypoints. You can list the keys with npz files and access them as a dict.

```
import numpy as np
if   name___== '  main  ':
    path = './pilar01_pilar02_matches.npz'
    npz = np.load(path) # Load dictionary with super point
    # Create a boolean mask with True for keypoints with a good
    match, and False for the rest
    mask = npz['matches'] > -1
    # Using the boolean mask, select the indexes of matched
    keypoints from image 2
    idxs = npz['matches'][mask]
    # Using the boolean mask, select the keypoints from image 1
    with a good match
    x1_sp = npz['keypoints0'][mask]
    # Using the indexes, select matched keypoints from image 2
    x2_sp = npz['keypoints1'][idxs]
```

A.4. Modifying the demo to also save the descriptors

The file is saved at line 283 in `match_pairs.py`. With a simple modification you can also save the Superpoints descriptors.

```
dsc0 = pred['descriptors0']
dsc1 = pred['descriptors1']
# Write the matches to disk.
out_matches = {'keypoints0': kpts0, 'keypoints1': kpts1,
               'matches': matches, 'match_confidence':
conf, 'descriptors0': dsc0, 'descriptors1': dsc1 }
np.savez(str(matches_path), **out_matches)
```