UE - Computer Vision
TP-5, **Camera-motion**

Panagiotis PAPADAKIS

The objective of this TP is to estimate **camera motion**, also known as **camera reconstruction** or **visual odometry**, using only 2 views of the observed scene. This amounts to estimating an Euclidean transformation ${}^{\mathcal{C}}\mathrm{T}_{\mathcal{C}'} = [\mathrm{R}^T | -\mathrm{R}^T\mathbf{t}]$ between two cameras $\mathcal{C}$ and $\mathcal{C}'$ (see Figure 1), assuming that the $\mathcal{C}$ frame coincides with the world frame.

In the absence of prior information with respect to the scale of the observed scene, the estimated translation vector $\mathbf{t}$ can only indicate the direction of translation and will be a vector of unit norm.

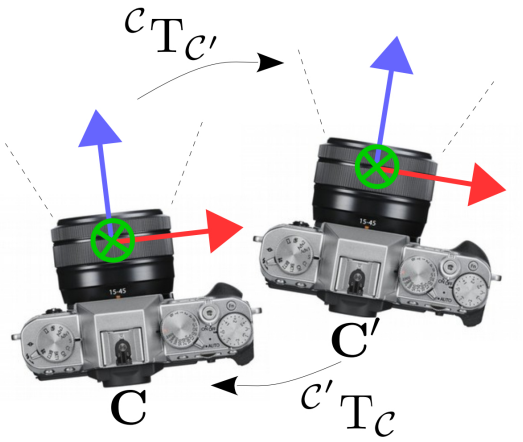

Figure 1: Schema of the two-camera set-up that we seek to estimate (XYZ axes are denoted by RED, GREEN and BLUE AXES).

1 Exercises

Use a previously given pair of images and corresponding calibration matrix (available in the folder `data/real-pair-of-images`) to estimate camera motion. Make sure, however, that you obtain and save your own pair of images (e.g. using `cheese` software) and corresponding calibration matrix so that you can use them later to test your code (spend 10 minutes max for data acquisition).

1.1 Extract 2D keypoints, correspondences and visualize

Remember to use `python3` and for every new terminal session type: `source /opt/campux/virtualenv/computervision/bin/activate`. For a pair of images for which you have removed lens distortion, detect and compute SIFT features together with the corresponding keypoints. Establish the correspondence between the detected keypoints, keep and visualize the good matches. An example of what is expected to be obtained is shown in Figure 2.

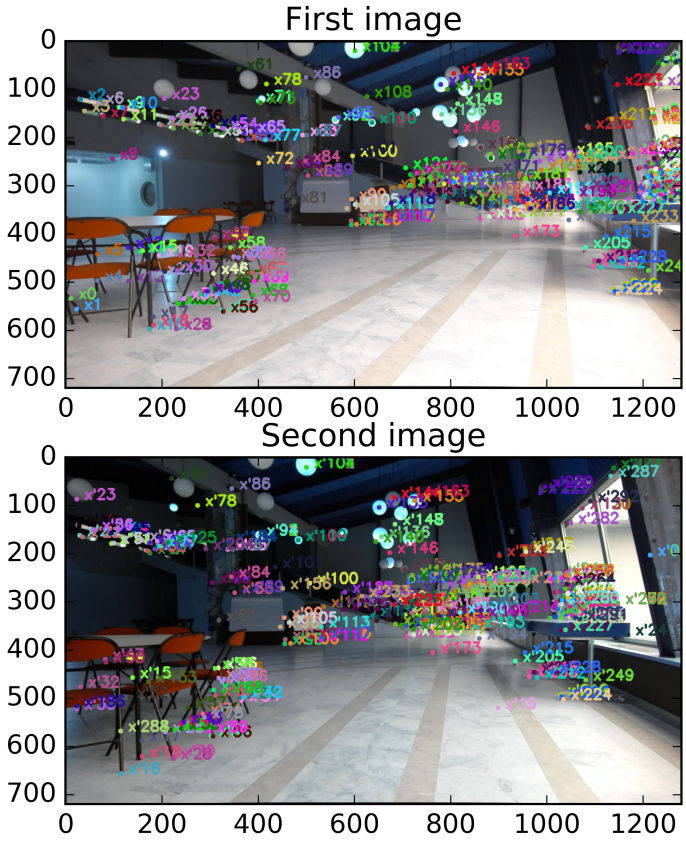

Figure 2: Pair of images with detected SIFT keypoints corresponding to *good* matches. Corresponding keypoints are indexed with the same number/color.

Due to matching errors, it is expected that some of the correspondences $\mathbf{x}_i \leftrightarrow \mathbf{x}'_i$ do not correspond to the same part of the scene. On the other hand, the remaining big majority of correspondences should be valid.

For this exercise, you will need to use OpenCV function `cv2.xfeatures2d` (cf. `https://bit.ly/2TLT6jF`) and class `cv2.BFMatcher` (cf. `https://bit.ly/2IoA9ye`).

1.2 Calculation of epipolar geometry between views

Using the previously established (good) SIFT keypoint matches between the pair of images, calculate the underlying epipolar geometry, namely:

The *Fundamental* matrix F This matrix **does not** require knowledge of the calibration matrices of the two cameras views. Then, use F to draw in the first image the epipolar lines $\mathbf{l}$ corresponding to each keypoint $\mathbf{x}'$, as well as the epipolar lines $\mathbf{l}'$ in the second image corresponding to each keypoint $\mathbf{x}$ (see example in Figure 3). You will need to use OpenCV function `cv2.findFundamentalMat` (cf. `https://bit.ly/2IqGp92`).

The *Essential* matrix E This matrix **does** require knowledge of the calibration matrices of the two cameras views.

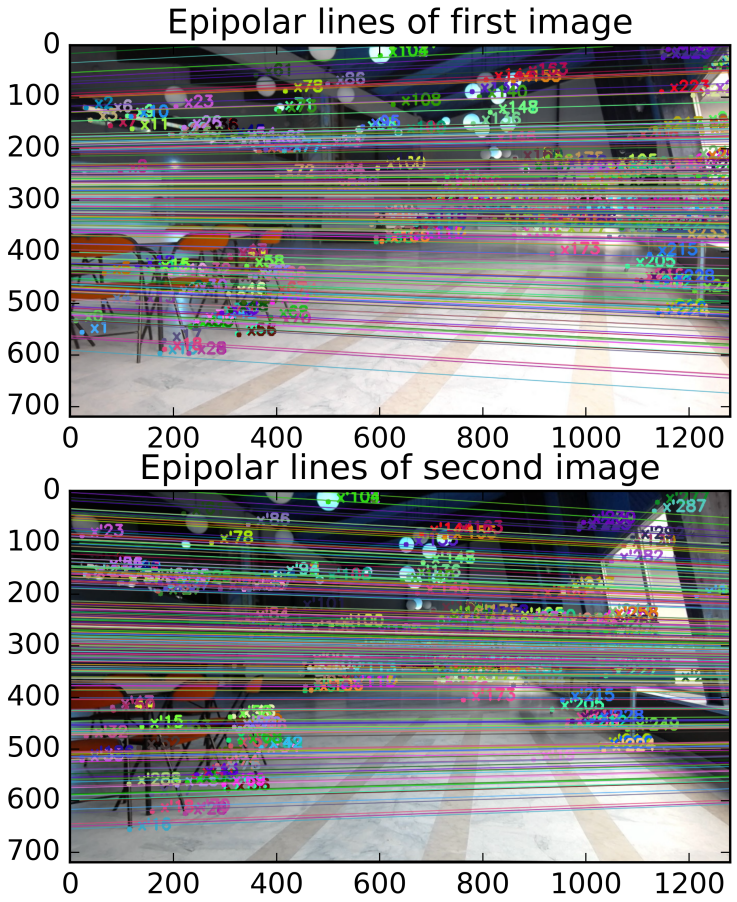

Figure 3: Epipolar lines for the correspondences between images.

1.3 3D pose recovery through triangulation

Knowing that $\mathrm{P}' = \mathrm{K}'[\mathrm{R}|\mathbf{t}]$ and $\mathbf{x}' = \mathrm{P}'\mathbf{X}$ and that the first camera coincides with the world coordinate frame, use OpenCV's function `cv2.recoverPose` (cf. `https://bit.ly/2QOZ2UY`), to obtain the motion of the second camera with respect to the first camera, namely, ${}^{\mathcal{C}}\mathrm{T}_{\mathcal{C}'} = [\mathrm{R}^T|-\mathrm{R}^T\mathbf{t}]$. This OpenCV function uses **triangulation** to decide which one of the 4 possible expressions for E (cf. slide 23 of the lecture) is realistically valid.

Finally, using the python module `Rotation` of `scipy.spatial.transform`, express the rotation matrix block of ${}^{\mathcal{C}}\mathrm{T}_{\mathcal{C}'}$ using Euler angles.

For the pair of real images that are given to you (see Figure 1), a robust estimate of the camera motion should be similar to:

$$ {}^{\mathcal{C}}\mathrm{T}_{\mathcal{C}'} = \begin{bmatrix} 0.97343359 & -0.00752095 & 0.22884597 & -0.99622637 \\ 0.01343752 & 0.99961424 & -0.02430667 & 0.00744974 \\ -0.22857488 & 0.02673605 & 0.97315914 & -0.08647267 \\ 0. & 0. & 0. & 1. \end{bmatrix} \quad (1) $$

which corresponds to Euler angles $[1.57371744^{\mathrm{o}}, 13.21318368^{\mathrm{o}}, 0.79087495^{\mathrm{o}}]$, around the $\mathbf{x}, \mathbf{y}$ and $\mathbf{z}$ axes respectively and a direction of translation parallel to $[-0.99622637, 0.00744974, -0.08647267]$. Looking at the first and second image in Figure 2, this solution corresponds intuitively to our expectation for a camera that was moved horizontally while slightly turning around the vertical axis, in the camera coordinate frame.

A pair of simulated images of a scene, taken by a simulated idealistic pinhole camera is further given to you (look in `data/simulated-pair-of-images`) together with the calibration parameters, as shown below:

Figure 4: Simulated pair of camera images.

The second view was taken at ${}^{\mathcal{C}}\mathrm{T}_{\mathcal{C}'} = [\mathrm{I}|[0.1, 0., 0.2]^T]$, namely, by only moving the camera forward by $0.2m$ and to the right by $0.1m$. Test the code that you developed earlier for recovering the camera motion, using the simulated pair view and provided intrinsic camera parameters.

1.4 Bonus - Extra

Following the instructions given in slides $24 - 26$, implement yourself the above function and verify that it gives the same results as `cv2.recoverPose`.