CSE-483: Mobile Robotics **Monsoon 2019**

Assignment-2

Released: August 27th *Deadline: September 7th*

Instructions

- This assignment is designed to get you familiar with epipolar geometry and visual odometry.
- Code can be written in Python or C++ only. OpenCV or any equivalent library can be used for image input & output, and where ever specified. Make sure your code is modular since you might be reusing them for future assignments.
- Submit your code files and a report (or as a single Jupyter notebook if you wish) as a zip file, named as ⟨team_id⟩_assignment2.zip.
- The report should include all outputs and results and a description of your approach. It should also briefly describe what each member worked on. The scoring will be primarily based on the report.
- Refer to the late days policy and plagiarism policy in the course information document.
- Start early! This assignment may take fairly long.

Q1) Epipolar lines and Epipoles

You've been given two images of the same scene, taken from different view-points. The fundamental matrix (F) that encodes their relative geometry and a subset of the corresponding points in both the images that were used to estimate F are provided as well.

Recall that given a point in one image, its corresponding location in the other image can be found to be along a line viz. the epipolar line. a) For the points in the first image, plot their corresponding epipolar lines in the second image as shown. Repeat this for the first image. The convention for F we follow is $x'^T F x = 0$, where x' is the location of the point in the second (right) image.

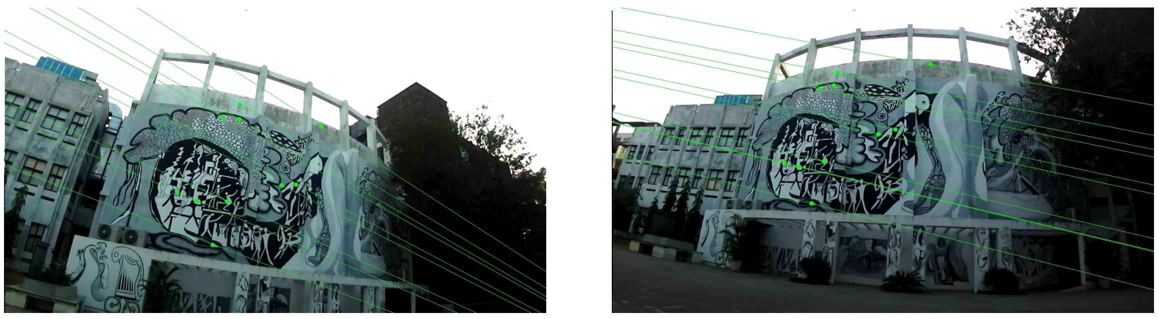

Figure 1: Epipolar lines drawn using the fundamental matrix for both the views.

Recall that the epipolar lines must all converge to their respective epipoles. But the epipoles here seem to lie outside the image. (b) How can you compute the locations of these epipoles without using these lines? Report the locations.

Q2) Feature-based Visual Odometry

Visual odometry (VO) is the process of recovering the egomotion (in other words, the trajectory) of an agent using only the input of a camera or a system of cameras attached to the agent. This is a well-studied problem in robotic vision and is a critical part of many applications such as mars rovers, and self-driving cars for localization. You will be implementing a basic monocular visual odometry algorithm in this part of the assignment.

To begin with, download all the required files from [here]. It contains a sequence of images from the KITTI dataset. The ground truth pose of each frame (in row-major order) and the camera parameters are provided as well.

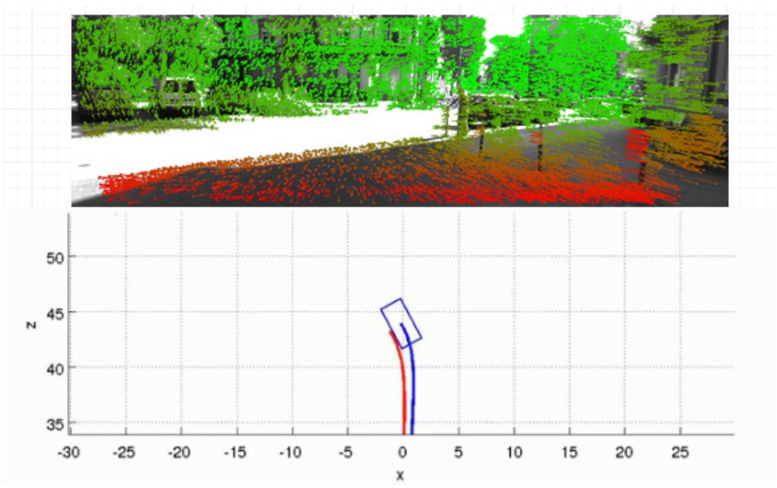

Figure 2: *libviso*, a popular open-source visual odometry library. In red is the ground truth trajectory, in blue is the estimated trajectory.

We will now go through the procedure step-by-step. The following is an overview of the entire algorithm,

1. Find corresponding features between frames I_k, I_{k-1}.
2. Using these feature correspondences, estimate the essential matrix between the two images within a RANSAC scheme.
3. Decompose this essential matrix to obtain the relative rotation R_k and translation t_k, and form the transformation T_k.
4. Scale the translation t_k with the absolute or relative scale.
5. Concatenate the relative transformation by computing $C_k = C_{k-1}T_k$, where C_{k-1} is the previous pose of the camera in the world frame.
6. Repeat steps $1 - 5$ for the remaining pairs of frames.

The main task in VO is to compute the relative transformations T_k from each pair of images I_k and I_{k-1} and then to concatenate these transformations to recover the full trajectory $C_{0:n}$ of the camera, where n is the total number of images. C_0 is taken to be the origin i.e. the world frame. There are two broad approaches to compute the relative motion T_k: appearance-based (or direct) methods, which use the intensity information of all the pixels in the two input images, and feature-based methods, which only use salient and repeatable features extracted and tracked across the images. You will be implementing a feature-based method.

For every new image I_k, the first step consists of detecting and matching 2D features with those from the previous frame. These 2D features (or simply keypoints) are locations in the image which we can reliably find in multiple images and possibly match them. To detect these keypoints use the following OpenCV code.

```
detector = cv2.SIFT()
keypoints = detector.detect(img1)
pts1 = np.array([x.pt for x in keypoints], dtype=np.float32)
```

SIFT (scale invariant feature detector) is one of many feature detectors, which applies a difference-of-Gaussian (DoG) operator on the entire image, followed by a nonmaxima supression on its output to detect the features. It achieves scale invariance by applying the detector on lower-scale and upper-scale versions of the image. You are not expected to know all the details of SIFT here.

Every detected keypoint is then associated with a description of the neighborhood it belongs to, which is called a descriptor. These descriptors are then used for searching for corresponding features in other images based on a similarity measure.

An alternative way to independently finding features in all candidate images and then matching them is to use a detect-then-track approach. Features are detected in the first image, and then tracked over the next set of images. For this, use OpenCV's Lukas-Kanade tracker.

```
pts2, status = cv2.calcOpticalFlowPyrLK(img1,img2,pts1)
pts1 = pts1[status == 1]
pts2 = pts2[status == 1]
```

The function computes the location of the points from the first image in the second image, by computing their 'optical flow', or simply their apparent motion. This optical flow is computed by applying the Lukas-Kande algorithm, an algorithm that uses spatial and temporal image gradients to compute the motion of the points (hence their locations). It also makes the assumption that nearby point have the same motion. Note that some features will eventually move out of the field-of-view, and tracks will be lost, so make sure to detect new features when the number of features goes below a threshold (say, 150).

As mentioned earlier, the main task is motion computation. Using these feature correspondences, implement the 8-point algorithm for fundamental matrix estimation. Implement it inside a RANSAC scheme to get rid of any outliers, as explained in class. Then, compute the essential matrix, and decompose it to the relative R and t using `cv2.recoverPose(E, points1, points2, K, R, t[, mask])`. Note that the function returns the R and t of the first camera with respect to the second, and not the other way around (The joys of working in robotics :')).

Now, you might recall that the absolute scale of the translation cannot be computed from just two images. The above function only returns the direction of t, as a unit vector. Use the ground truth translation to get the absolute scale, and multiply your unit translation with this scale. Then concatenate your transformations, and repeat for the next pair of frames to recover the full absolute trajectory.

Deliverable

- A .txt file containing the estimated poses, provided in the same format as the ground truth file.
- A plot of the estimated trajectory along with the ground truth trajectory. Also report the obtained trajectory error. Use [EVO] for this.

  ```
  pip install evo --upgrade --no-binary evo
  evo_ape kitti ground-truth.txt your-result.txt -va --plot --plot_mode xz
  ```

- Comment on the performance of your algorithm. Where does it work well, where does it fail, and why? If you want to test it on more sequences, ask!
- [Bonus] Describe other ways to compute the absolute scale.

Further reading

- Scaramuzza, D., Fraundorfer, F. (2011). Visual odometry [tutorial]. IEEE robotics & automation magazine, 18(4), 80-92.