Zaidi Final Project

Syed Zaidi

May 13, 2023

1 Note to run the Python code

```
To use the cv2.recoverPose() use the command line:
python FinalProject.py
To use the custom_recover_pose() function use the command line:
python FinalProject.py --custom_recover_pose
```

2 Estimate Rotations and Translations Between Frames

2.1 Compute Intrinsic Matrix

The first step is to compute the intrinsic matrix K. The intrinsic matrix is a 3×3 matrix containing the camera's focal length (f_x, f_y) and optical center (c_x, c_y). To compute the intrinsic matrix, we use the `ReadCameraModel` function provided in the `ReadCameraModel.py` script. This function reads the camera parameters from the dataset and returns the focal length (f_x, f_y) and optical center (c_x, c_y) values.

After obtaining these values, we create the intrinsic matrix K as follows:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

We define a function called `compute_intrinsic_matrix`, which takes the focal length and optical center values as input and returns the intrinsic matrix K.

2.2 Load and Demosaic Images

The second step is to load and demosaic the input images. The input images are in Bayer format, which is a single-channel representation of color images. To recover the color images, we use the OpenCV function `cv2.cvtColor` with the `cv2.COLOR_BayerGR2BGR` conversion code.

First, we load the Bayer pattern encoded image using `cv2.imread` with the `flags=-1` option. Then, we convert the Bayer image into a color image using `cv2.cvtColor`.

Optionally, we can undistort the images using the provided `UndistortImage.py` script. This script takes the color image and a lookup table (LUT) as input, and returns the undistorted image.

We define a function called `load_and_demosaic_image` that takes the filename and LUT as input and returns the undistorted color image.

2.3 Keypoint Correspondences

The third step is to find keypoint correspondences between successive frames. To do this, we use the ORB (Oriented FAST and Rotated BRIEF) feature detector and descriptor, which is a fast and efficient method for detecting and describing features in images. We first detect keypoints and compute their descriptors in both images using the ORB detector.

Next, we use the Brute-Force matcher to match the descriptors from the two images. This matcher calculates the Hamming distance between each pair of descriptors and returns the best matches. After matching the descriptors, we sort the matches by their distance, which is a measure of the similarity between the keypoints.

To keep only the good matches, we define a ratio parameter called `good_match_ratio` and keep only the top fraction of matches according to this ratio. By default, we set this ratio to 0.75, meaning that we keep the top 75% of matches.

Finally, we extract the locations of the good matches from both images, which are the point correspondences between the successive frames.

2.4 Estimate Fundamental Matrix

The next step is to estimate the fundamental matrix between the two frames using the matched keypoints. The fundamental matrix relates the corresponding points in two images taken from different camera positions. It can be used to compute the epipolar lines and epipoles in the images, which are useful for stereo vision and 3D reconstruction.

We use the OpenCV function `cv2.findFundamentalMat` to estimate the fundamental matrix. This function takes the matched keypoints from the previous step as input and returns the estimated fundamental matrix. We use the RANSAC method for robust estimation, which is less sensitive to outliers. The RANSAC method requires a reprojection threshold, which we set to 3.0 pixels by default. This threshold determines the maximum distance between the reprojected points and the epipolar lines for a match to be considered as an inlier.

2.5 Recover Essential Matrix

The next step is to recover the essential matrix E from the fundamental matrix F. The essential matrix is related to the fundamental matrix and the camera's intrinsic parameters. It can be used to recover the relative pose between two cameras, which is useful for 3D reconstruction.

To recover the essential matrix, we use the following formula:

$$E = K^T \cdot F \cdot K$$

where K is the camera's intrinsic matrix, and F is the fundamental matrix estimated in the previous step. In Python, we define a function called `recover_essential_matrix` that takes the fundamental matrix F and the intrinsic matrix K as input and returns the essential matrix E.

2.6 Reconstruct Rotation and Translation Parameters from E

The final step is to decompose the essential matrix E into a physically realizable translation vector T and rotation matrix R. This step is necessary to obtain the relative pose between the two cameras, which is useful for 3D reconstruction and visual odometry.

We use the OpenCV function `cv2.recoverPose` to decompose the essential matrix. This function takes the essential matrix E, the matched keypoints from the previous steps, and the camera's intrinsic matrix K as input, and returns the rotation matrix R and the translation vector T. The function chooses the decomposition that satisfies the depth positivity constraint, which ensures that the scene points are in front of both cameras.

In Python, we define a function called `decompose_essential_matrix` that takes the essential matrix E, the matched keypoints, and the intrinsic matrix K as input, and returns the rotation matrix R and the translation vector T.

3 Reconstruct the Trajectory

In this section, we compute and plot the positions of the camera centers for each frame based on the rotation and translation parameters between successive frames. We assume that the first video frame started at the origin.

3.1 Accumulating Transformations

We accumulate the transformations by iteratively computing the camera centers as follows:

$$X_{i+1} = R_i X_i + t_i \tag{1}$$

where X_i represents the position of the camera center in the coordinate system of the first camera at frame i, R_i is the rotation matrix, and t_i is the translation vector between frame i and frame $i + 1$.

We start from the origin and compute the positions of the camera centers recursively.

3.2 Plotting the Trajectory

Once the camera centers are computed, we plot the trajectory in both 2D and 3D. The 2D plot projects the trajectory onto the X-Z plane, while the 3D plot visualizes the full 3D motion of the camera.

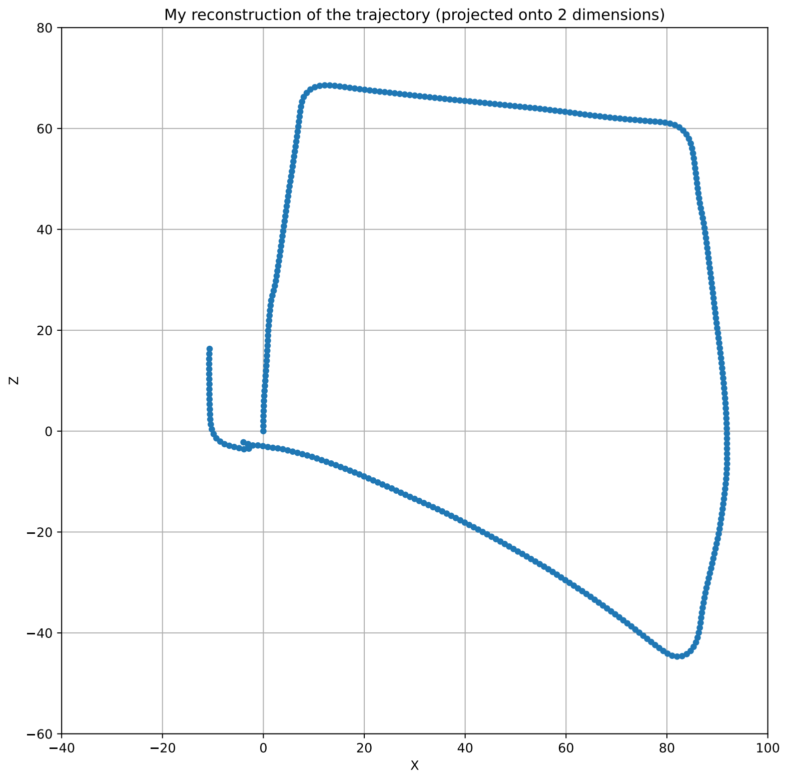

Figure 1: Reconstructed camera trajectory projected onto 2 dimensions (X-Z plane)

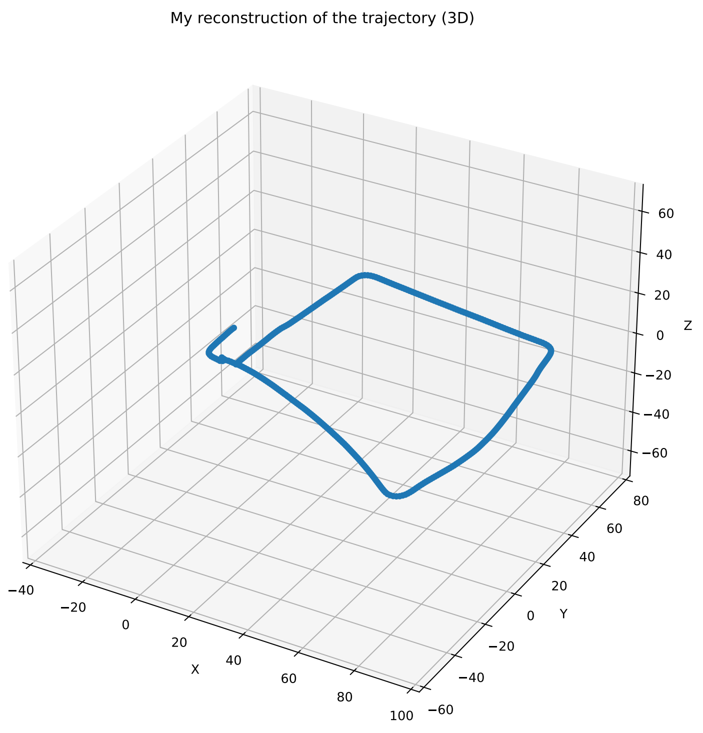

Figure 2: Reconstructed camera trajectory in 3D

Our solution successfully reconstructs the camera trajectory based on the rotation and translation parameters between successive frames. The resulting 2D and 3D plots provide a clear visualization of the camera motion throughout the video sequence.

4 Reconstruct Rotation and Translation Parameters Yourself

The provided code effectively implements a function to recover camera pose from the essential matrix. The procedure begins by decomposing the essential matrix into four potential combinations of rotation and translation matrices. This decomposition employs a mathematical technique known as singular value decomposition (SVD).

Following this initial decomposition, each of the four possible poses is scrutinized for its validity. This involves confirming the determinant of the rotation matrix is positive. A positive determinant is critical as it assures the rotation matrix does not induce any mirroring transformations, which are not physically meaningful in the context of camera pose.

Upon validation of each pose, the function proceeds to triangulate 3D points using the current pose and the camera parameters. The number of these triangulated points located in front of both cameras is then counted for each pose. This is determined by verifying that the depth (Z-coordinate) of each point is positive in the coordinate systems of both cameras.

Finally, the function returns the pose associated with the maximum number of points located in front of both cameras. This implies that the chosen pose allows the most number of 3D points to be visible from both camera positions. An additional optimization to this function could include an early stopping condition when all points are situated in front of both cameras, thus circumventing unnecessary computations.

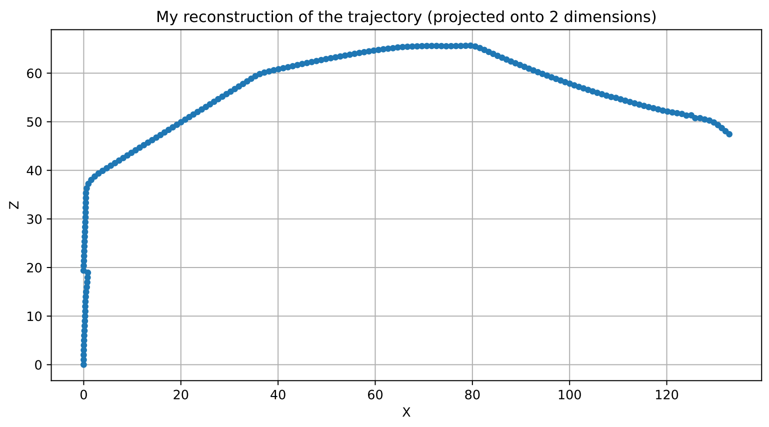

Figure 3: Reconstructed camera trajectory projected onto 2 dimensions (X-Z plane)

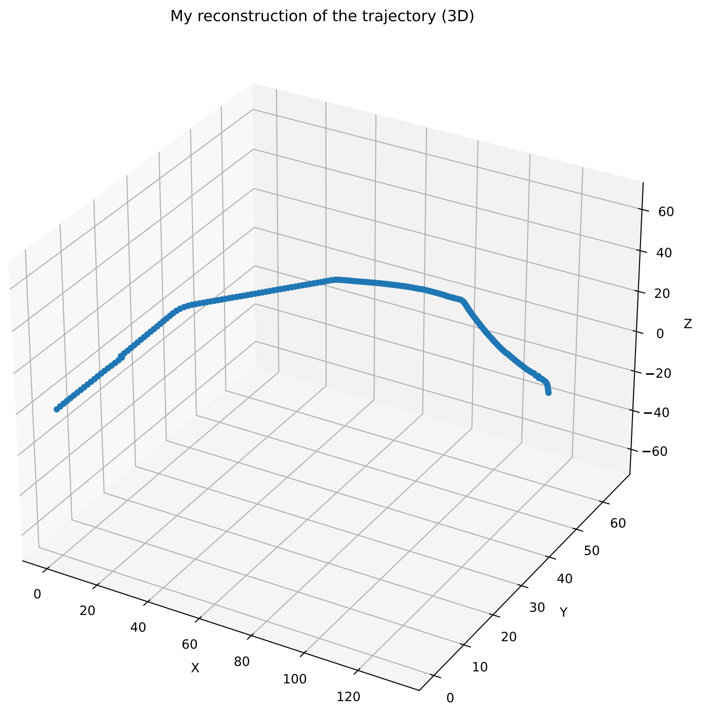

Figure 4: Reconstructed camera trajectory in 3D

But it seems, although the code is able of capturing the trajectory of the straight road, it fails to capture the sharp turns of the camera leading to this soft turn and overall wrong full trajectory.