Final: Report and Code Output

Alejandro J. Rigau

May 19, 2021

Abstract:

In this Final, I present my Visual Odometry results. I will use images obtained from a video to estimate Rotations and Translations Between Frames with hopes that I will be able to reconstruct the trajectory of the video. In the end, two graphs are generated, one 2D and another 3D, both showing the trajectory of the car in the video.

3 Estimate Rotations and Translations Between Frames

In this part I estimate the 3D motion (translation and rotation) between successive frames in the sequence. I will do this with all the images in the video starting from the first image and ending in the second to last image.

3.1 Compute Intrinsic Matrix

In this section, I used the provided ***ReadCameraModel()*** function to extract the camera parameters which I used to construct the intrinsic matrix:

$$\begin{matrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{matrix}$$

Which in code is K = np.array([[fx,0,cx],[0,fy,cy],[0,0,1]])

3.2 Load and Demosaic Images

Since the images that we have in our dataset are in Bayer format, I have to recover the color of each image. Using the **os** library, I obtained a list of all the names of the images in the folder. With this list, I can now iterate over the list and load two images at a time, i and i+1. To load the images, I use the **imread()** function provided in the project instructions and the output would then be passed to the **cvtColor()** function to restore the color. I also completed the optional step that would undistort the current frame by calling the **UndistortImage()** function provided in one of the starter files.

3.3 Keypoint Correspondences

Now that I have loaded two images at a time, I can find point correspondences between them. I used the code from https://docs.opencv.org/master/da/de9/tutorial_py_epipolar_geometry.html because it had all the steps that I needed to find the correct point correspondences. This method would first initiate a SIFT detector and use it to find keypoints and descriptors for each of the two images. The descriptors would then be passed to a FLANN based matcher that would use K Nearest Neighbors to find features that matched. Now that we have the matches, the code performs a ratio test to ensure that the points match to a given amount of error. These points that pass these tests are then stored in arrays to be used in the next part.

3.4 Estimate Fundamental Matrix

Now that I have the important points, I can use them to calculate the Fundamental matrix. This matrix will then be used to calculate our Essential Matrix which is needed to determine translation and rotation. To find the Fundamental Matrix I used the **cv2.findFundamentalMat()** where I used the RANSAC method. A point in one image is translated into a line in the other using the Fundamental Matrix. This is calculated using points from both images that match. To find the fundamental matrix, a minimum of 8 such points are required, and the RANSAC algorithm provides a very reliable result.

3.5 Recover Essential Matrix

The translation and rotation information in the Essential Matrix describe the location of the second camera in relation to the first. This can now be easily calculated now that we have the Fundamental Matrix and the Intrinsic Matrix. I used the formula:

$$E = K^T F K$$

Or in code: E = np.matmul(np.matmul(K.T, F), K)

3.6 Reconstruct Rotation and Translation Parameters from E

To Decompose E into a physically realizable translation T and rotation R, I use the function **recoverPose()**. This function takes in E, the key points, and finally the Intrinsic Matrix. By passing in K, the function is able to properly compute the trajectory of the camera. This function decomposes an essential matrix and then performs a cheirality check to verify possible pose hypotheses. The depth of the triangulated 3D points should be positive, according to the cheirality check. The function would return a rotation and translation, which I would use with the following formula to update our current position:

$$R_{pos} = RR_{pos}$$

$$t_{pos} = t_{pos} + tR_{pos}$$

Where R_{pos} and t_{pos} are the new current rotation and translation.

For every image pair being processed, I would save the t_{pos} to an array so that I could later plot them.

4 Reconstruct the Trajectory

The previous section had to be computed for every image in the dataset for a total of 376 distinct rotations and translations. The results are as follow:

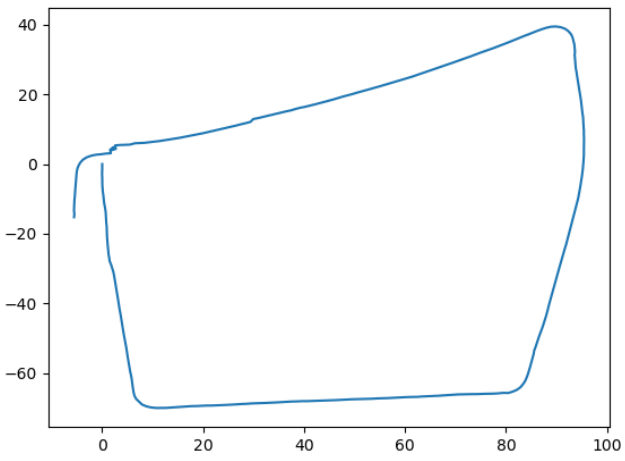

Figure 1: 2D Reconstruction

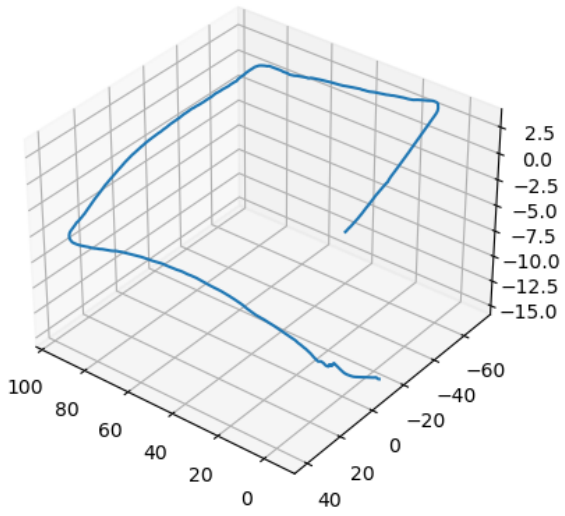

Figure 2: 3D Reconstruction

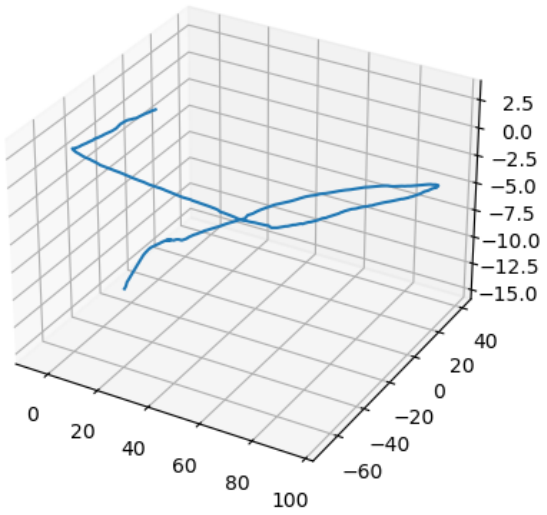

Figure 3: 3D Reconstruction from another angle

From the obtained images we can see that I was able to reconstruct the trajectory pretty accurately in the 2D space but the car seems to fly slightly in the 3D space. This is due to many

details in the images like speed bumps or other elevations in the road. The model could be tuned better to reduce this effect in the Z axis since I essentially used most of the default parameters in the OpenCV functions.