ENPM 673: Perception for Autonomous Robots
Project 5

Shreya Gummadi
Sneha Ganesh Nayak
Ghanem Jamal Eddine

6 May 2019

1. Visual Odometry

Visual odometry is the process of determining the position and orientation of a robot by analyzing the associated camera images. In this we compute the position of camera between two successive frames and plot it. We then compute the intrinsic camera parameters using ReadCameraModel. Next, we extract features for each frame and match them, followed by computation of the best Fundamental Matrix with the implementation of the 8-point Normalization with the aid of RANSAC. The Essential Matrix is then computed which then helps in obtaining the camera poses. The most correct camera pose is obtained using the Linear Triangulation and Chirality condition.

In order to compare and contrast the plots, we have also implemented the inbuilt version of this code in OpenCV. In addition, we have implemented the code for Non-Linear Triangulation, which would give better results.

2. Data Preparation

In order to proceed with the pipeline, we first need to do some image processing. This is done in the DataPrep.py.

- The dataset is in Bayer format, cv2.cvtColor(img, cv2.COLOR BayerGR2BGR) is used to convert the images into color images.
- After that, UndistortImage is used to undistort the images. We save these new images and work with them.

Next we extract the intrinsic camera parameter using ReadCameraModel.

3. The Basic Pipeline

3.1. **Feature extraction**

The features are extracted and matched using ORB and Brute force matcher. We take the 50 best matches.

3.2. **Fundamental Matrix with RANSAC**

8 random points are selected and checked to see if they are inliers or outliers. We define a threshold to define what points are inliers. Once we get the inliers points, it is used to compute the Fundamental Matrix.

3.3. **Essential Matrix**

The Fundamental matrix is used to estimate the Essential Matrix.

3.4. **Pose set**

The essential matrix is decomposed to get 4 sets of rotations and translations.

3.5. **Linear Triangulation**

In order to compute the 3D points from the 2D points, we created and used linear_triangulation function.

3.6. **Chirality Condition**

The chirality condition is checked to obtain the correct rotation and translation i.e. the pose of the camera.

3.7. **Update the global camera pose**

Once we obtain the unique camera, we use that to get the global rotation and translation using the following equations:

$$R = R*R\_prev;\ t = t+R*t\_prev$$

3.8. **Compute Camera Trajectory**

The orientation and translation are calculated in the previous step with respect to the world coordinate frames.The computed trajectory is then plotted.

All these steps are defined in the Functions.py which are then called in the main script.

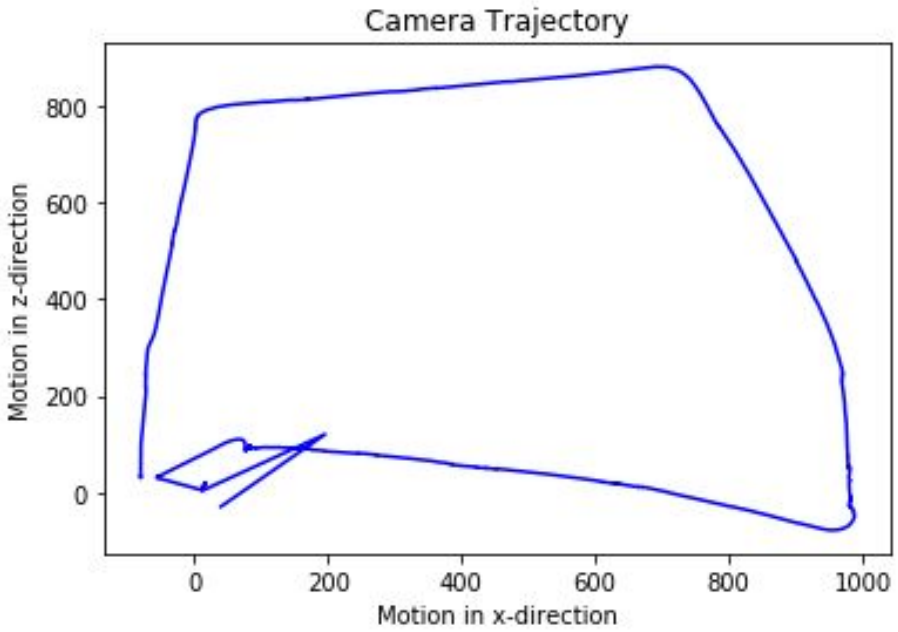

Fig 1. Camera Trajectory generated using our function.

4. Extra Credit

We used the OpenCV inbuilt functions cv2.findEssentialMat() to get the essential matrix and cv2.recoverPose() to get the camera pose. We compared this to our result. This is done in the inbuilt.py.

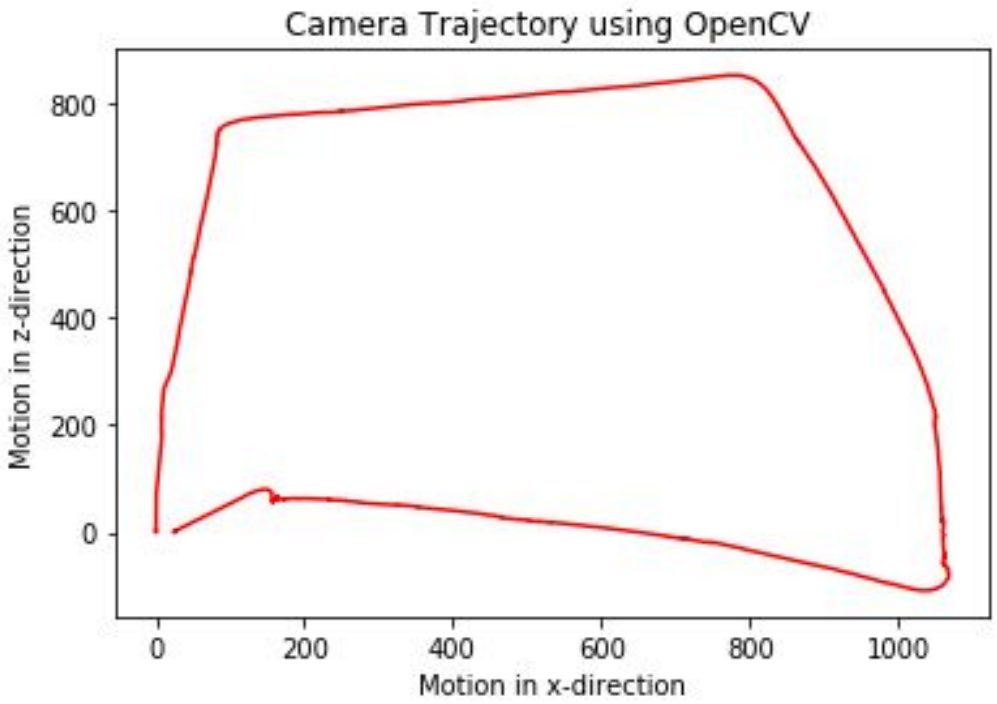

Fig 2. Camera Trajectory using OpenCV.

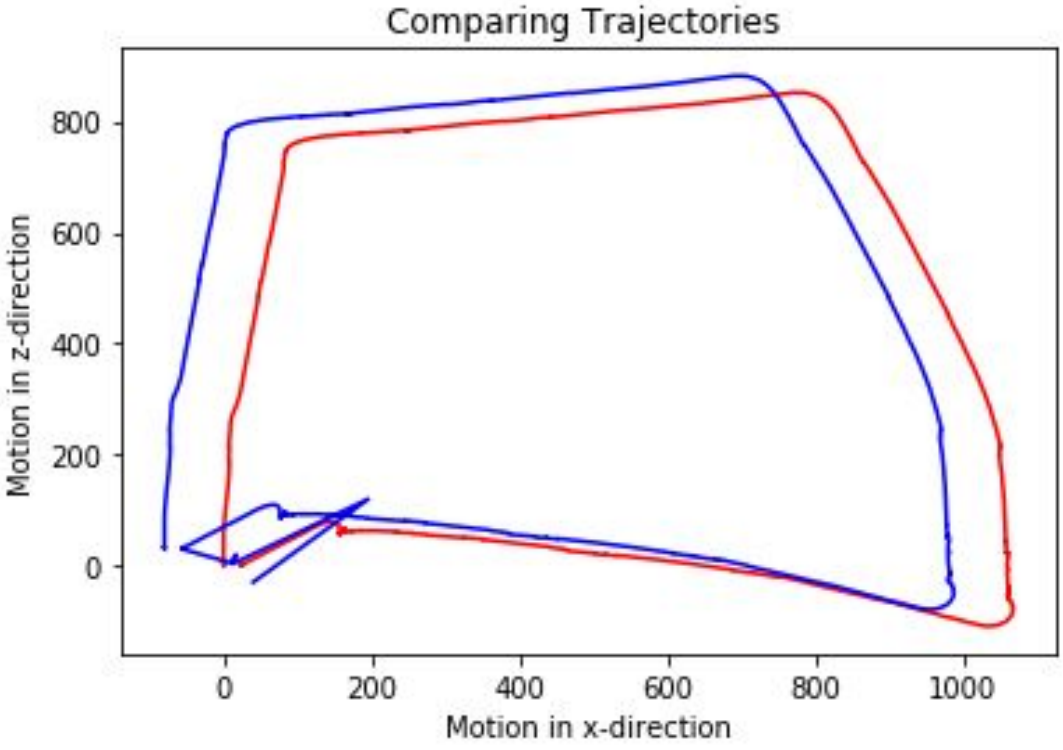

Fig 3. Trajectories generated by OpenCV functions vs our functions (blue:ours,red:OpenCV)

We also wrote the code for the Non-linear triangulation which takes the camera pose and linearly triangulated points and calculates the error. It non linearly optimizes the error. This can be found in NonlinearTriangulation.py.

```
def fromHomogenous(X):
    print('Homo',X)
    x = X[:-1, :]
    x /= X[-1, :]
    return x

#convert from normal to homogenous
def toHomogenous(x):
    X = np.vstack([x, np.ones((1, x.shape[1]))])
    return X

#perform non-linear triangulation
def nonlinearTriangulation(pose1, pose2, pt1, pt2):
    cmatrix = [[964.828979,0, 643.788025],[0, 964.828979, 484.40799],[0,0,1]]
    _,X,_ = linear_triangulation(pose1, pose2, pt1, pt2)
    print('XX',X)
    X = np.transpose(X)
    mat, success = leastsq(triangulationError, X, args=(pose1, pose2, pt1, pt2), maxfev=10000)

    #non-linear triangulated matrix
    mat = np.matrix(mat)
    mat = np.transpose(mat)
    return mat

#compute the error while doing triangulation
def triangulationError(x, Rt1, Rt2, x1, x2):
    cmatrix = [[964.828979,0, 643.788025],[0, 964.828979, 484.40799],[0,0,1]]
    X = np.matrix(x).T
    print(Rt1,R)
    print('X',X)
    px1 = fromHomogenous(np.matmul(cmatrix, np.matmul(Rt1, toHomogenous(fromHomogenous(X)))))
    px2 = fromHomogenous(np.matmul(cmatrix, np.matmul(Rt2, toHomogenous(fromHomogenous(X)))))
    diff1 = px1 - x1[:2]
    diff2 = px2 - x2[:2]

    return np.asarray(np.vstack([diff1, diff2]).T)[0, :]
```

Fig 3. Non Linear Triangulation Code.