ENPM 673: Perception for Autonomous Robots **Released: April-21.**

Report

Name: Arpit Aggarwal and Shantam Bajpai

1 Visual Odometry

The aim of the project was estimating the 3D motion of the camera and plotting the trajectory of the camera. The following steps were followed:

1.1 Finding keypoints

For each pair of successive frames, the point correspondences were found. The SIFT Keypoint algorithm, along with cv2.FlannBasedMatcher() was used to find the point correspondences between images. An example is shown in Figure 1.

1.2 Finding Fundamental Matrix using RANSAC

After finding the keypoints between successive frames, the fundamental matrix is found. The fundamental matrix is found using RANSAC method, where at a particular iteration 8 random points are taken and then the F matrix which satisfies the following constraint for most of the points, is considered as the best estimate.

$$x_r^T F x_l < 0.06$$

where, F is the fundamental matrix, and the other two represent the points in the left and right image.

As mentioned, at a particular iteration 8 random points are taken and the fundamental matrix is found by solving the equation:

$$Ax = 0$$

where A matrix is formed by stacking the 8 points as shown below and x represents the elements of the fundamental matrix to be found.

Figure 1: Keypoints for two successive frames

$$A = \begin{bmatrix} x_1 * u_1 & y_1 * u_1 & u_1 & x_1 * v_1 & y_1 * v_1 & v_1 & x_1 & y_1 & 1 \\ x_2 * u_2 & y_2 * u_2 & u_2 & x_2 * v_2 & y_2 * v_2 & v_2 & x_2 & y_2 & 1 \\ x_3 * u_3 & y_3 * u_3 & u_3 & x_3 * v_3 & y_3 * v_3 & v_3 & x_3 & y_3 & 1 \\ x_4 * u_4 & y_4 * u_4 & u_4 & x_4 * v_4 & y_4 * v_4 & v_4 & x_4 & y_4 & 1 \\ x_5 * u_5 & y_5 * u_5 & u_5 & x_5 * v_5 & y_5 * v_5 & v_5 & x_5 & y_5 & 1 \\ x_6 * u_6 & y_6 * u_6 & u_6 & x_6 * v_6 & y_6 * v_6 & v_6 & x_6 & y_6 & 1 \\ x_7 * u_7 & y_7 * u_7 & u_7 & x_7 * v_7 & y_7 * v_7 & v_7 & x_7 & y_7 & 1 \\ x_8 * u_8 & y_8 * u_8 & u_8 & x_8 * v_8 & y_8 * v_8 & v_8 & x_8 & y_8 & 1 \end{bmatrix}$$

The process was repeated for 2000 iterations to obtain a better estimate of the fundamental matrix. After finding the fundamental matrix, the rank 2 constraint on the fundamental matrix is put by finding the SVD of the matrix and setting the smallest eigenvalue to zero. Also, the points were normalized to give a better result.

1.3 Finding Essential Matrix

After finding the fundamental matrix, the essential matrix is found using the fundamental matrix and camera matrix as follows:

$$K^T F K = 0$$

This gives us a noisy essential matrix. For better estimate, we find the SVD of the calculated essential matrix and set the three eigenvalues to 1, 1, 0. Then the essential matrix is re-calculated to obtain a better result. The code for finding the essential matrix is included in "utils.py" file.

1.4 Finding Camera Poses

After finding the essential matrix, the four camera poses are found from the Single Value Decomposition of the essential matrix. The following steps are followed:

1. Find u, d and v matrices from the Single Value Decomposition of the essential matrix.
2. The four translation matrices are:

translationMatrix1 = u[:, 2]
translationMatrix2 = -u[:, 2]
translationMatrix3 = u[:, 2]
translationMatrix4 = -u[:, 2]

3. The four rotation matrices are:

rotationMatrix1 = np.dot(u, np.dot(w, v))
rotationMatrix2 = np.dot(u, np.dot(w, v))
rotationMatrix3 = np.dot(u, np.dot(w.T, v))
rotationMatrix4 = np.dot(u, np.dot(w.T, v))

where,

$$w = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4. If the determinant of the rotation matrices (r) is negative, then the rotation matrix 'r' is updated as "r = -r" and the translation matrix is updated as "t = -t".
5. This gives us the four camera poses.

1.5 Finding Optimal Camera Pose

After finding the four camera poses, the optimal camera pose is to be found. We used linear triangulation to find the 3D point for the corresponding 2D points in the two images. Using the 3D point, we took the camera pose which gave the maximum number of 3D points in the front of the camera. This camera pose was used for further processing. The code for finding the optimal camera pose is included in "utils.py" file.

1.6 Plotting Camera Trajectory

The following plot was obtained for the camera trajectory:

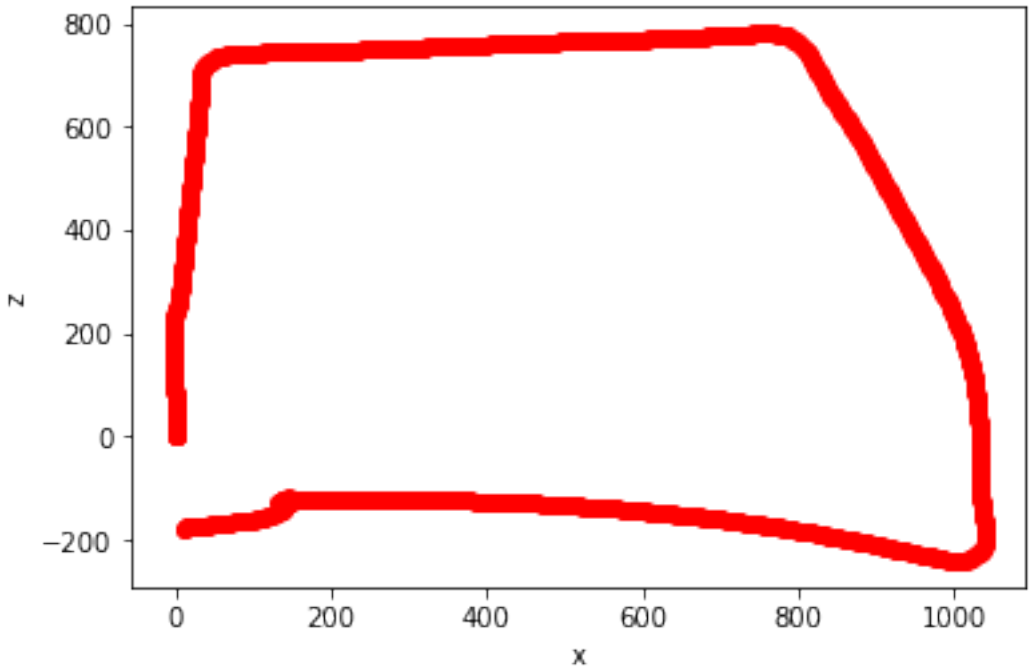

Figure 2: Camera Trajectory

2 Extra Credit

2.1 Non-Linear Triangulation

Linear Triangulation method finds the corresponding 3D points from the 2D image points by minimizing the algebraic error. However, this estimate is not always accurate. To obtain a better result we use Non-Linear Triangulation method, where we minimize the re-projection error by using the "scipy.optimize.leastsq" method. This tends to give a better 3D point estimate for the corresponding 2D image points in the left and right image. The code snippet is shown in Figure 3 and included in the "utils.py" file.

2.2 Non-Linear PnP

After finding the 2D-3D correspondences and the camera matrix, the camera pose was estimated using linear least squares. The 2D points were normalized using the camera matrix. The solution of the linear system that relates the 2D and 3D points gave the rotation and translation matrix.

```python
# performs non-linear triangulation
def get_non_linear_triangulation(camera_pose_1, camera_pose_2, pointLeft, pointRight):
    """
    Inputs:

    camera_pose_1: the base camera pose
    camera_pose_2: the camera pose
    pointLeft: the image point in the left image
    pointRight: the image point in the right image
    k_matrix: the camera matrix

    Output:

    point: the 3D point in camera coordinate system
    """

    # perform linear triangulation and get linear estimate
    estimated_point = get_linear_triangulation(camera_pose_1, camera_pose_2, pointLeft, pointRight)

    # run Levenberg-Marquardt algorithm
    args = (camera_pose_1, camera_pose_2, pointLeft, pointRight)
    point, success = leastsq(get_triangulation_error, estimated_point, args = args, maxfev = 10000)
    point = np.matrix(point).T

    # return point
    return point


# the triangulation error function for non-linear triangulation
def get_triangulation_error(estimated_point, camera_pose_1, camera_pose_2, pointLeft, pointRight):

    # project into each frame
    estimated_point = np.array([estimated_point[0, 0], estimated_point[1, 0], estimated_point[2, 0], [1]])
    estimated_ptLeft = fromHomogenous(np.dot(camera_pose_1, estimated_point))
    estimated_ptRight = fromHomogenous(np.dot(camera_pose_2, estimated_point))
    estimated_ptLeft = np.array([estimated_ptLeft[0, 0] / estimated_ptLeft[2, 0], estimated_ptLeft[0, 0] / estimate
    estimated_ptRight = np.array([estimated_ptRight[0, 0] / estimated_ptRight[2, 0], estimated_ptRight[0, 0] / esti

    # compute the diffs
    diff1 = estimated_ptLeft - pointLeft
    diff2 = estimated_ptRight - pointRight
```

Figure 3: Non-Linear Triangulation method (Code in utils.py file)

The linearly estimated camera pose can be improved by RANSAC to take into account the outliers that may exist.

The next step was implementing Non-Linear PnP. Using the 2D-3D correspondences and the linearly estimated camera pose, a refined camera pose was estimated by minimizing the reprojection error by using the "scipy.optimize.leastsq" method. The code snippet is shown in Figure 4 and included in the "utils.py" file.

2.3 Camera Trajectory using in-built OpenCV functions

The camera trajectory using inbuilt OpenCV functions is shown. Here, the essential matrix and optimal camera pose were found using the OpenCV functions. The code for obtaining the trajectory of the camera through OpenCV functions is included in "main.py" file.

```python
def NonLinearPnpError(X,x,K,C,R):
    """
    Inputs:
    X: This is an Nx4 Matrix whose row gives correspondence with the 2D Image
    x: This is an Nx2 Matrix whose row gives correspondence with the 3D Image
    C, R: The Camera Pose
    """
    quaternion = RotationMatrixToQuaternion(R)
    R = QuaternionToMatrix(quaternion)

    # Estimate the Camera Pose
    P = np.dot(np.dot(K,R), np.concatenate((np.eye(3), -C), axis = 1))

    # The reprojection terms
    u_reprojection = np.matmul(P[0].reshape(1,x.shape[0]), X) / np.matmul(P[2].reshape(1,x.shape[0]), X)
    v_reprojection = np.matmul(P[1].reshape(1,x.shape[0]), X) / np.matmul(P[2].reshape(1,x.shape[0]), X)

    # Calculate the difference
    diff1 = (x.T[0,:].reshape(1,x.shape[0]) - u_reprojection)**2
    diff2 = (x.T[1,:].reshape(1,x.shape[0]) - v_reprojection)**2

    # Reprojection Error
    reprojection_error = diff1 + diff2

    return reprojection_error

def NonLinearPnp(X,x,K,C,R):
    """
    Inputs:
    X: This is an Nx3 Matrix whose row gives correspondence with the 2D Image
    x: This is an Nx2 Matrix whose row gives correspondence with the 3D Image
    C, R: The Camera Pose

    Output:
    C, R: The Camera Pose
    """
    # Arguments for the Error Method
    args = (X,x,K)

    # Optimize the error function
    pose,success = leastsq(NonLinearPnpError,[C,R], args = args)

    return pose[0],pose[1]
```

Figure 4: Non-Linear PnP method (Code in utils.py file)

3 References

1. https://docs.opencv.org/master/da/de9/tutorial_py_epipolar_geometry.html
2. https://cmsc733.github.io/2019/proj/p3/
3. https://answers.opencv.org/question/18125/epilines-not-correct/
4. https://www.youtube.com/watch?v=1X93H_0_W5k

5 https://web.stanford.edu/class/cs231a/course_notes/04-stereo-systems.pdf

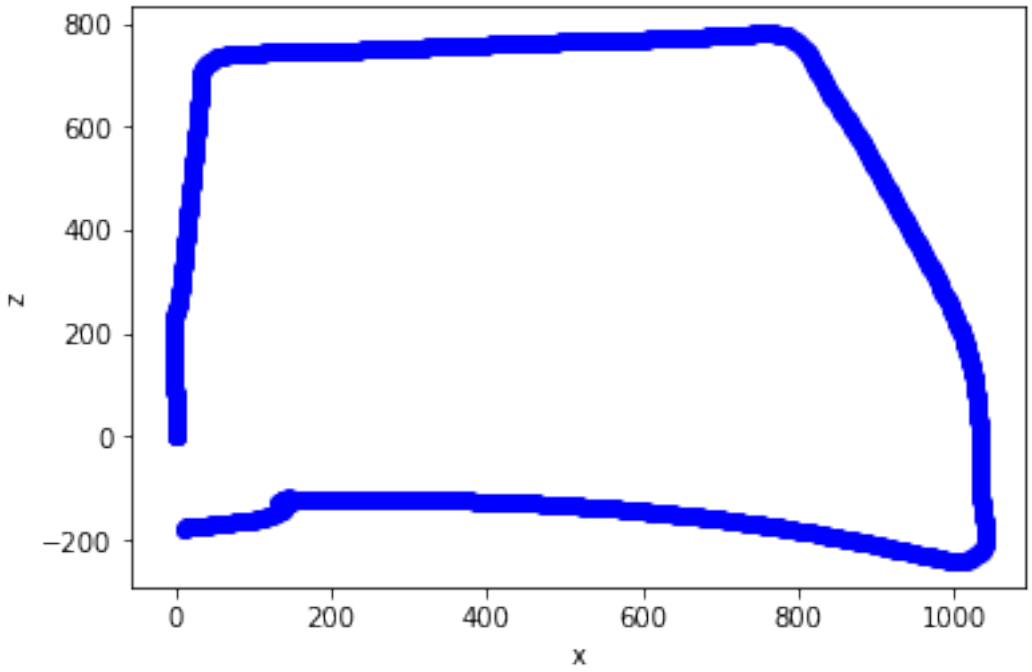

Figure 5: Camera Trajectory using OpenCV functions

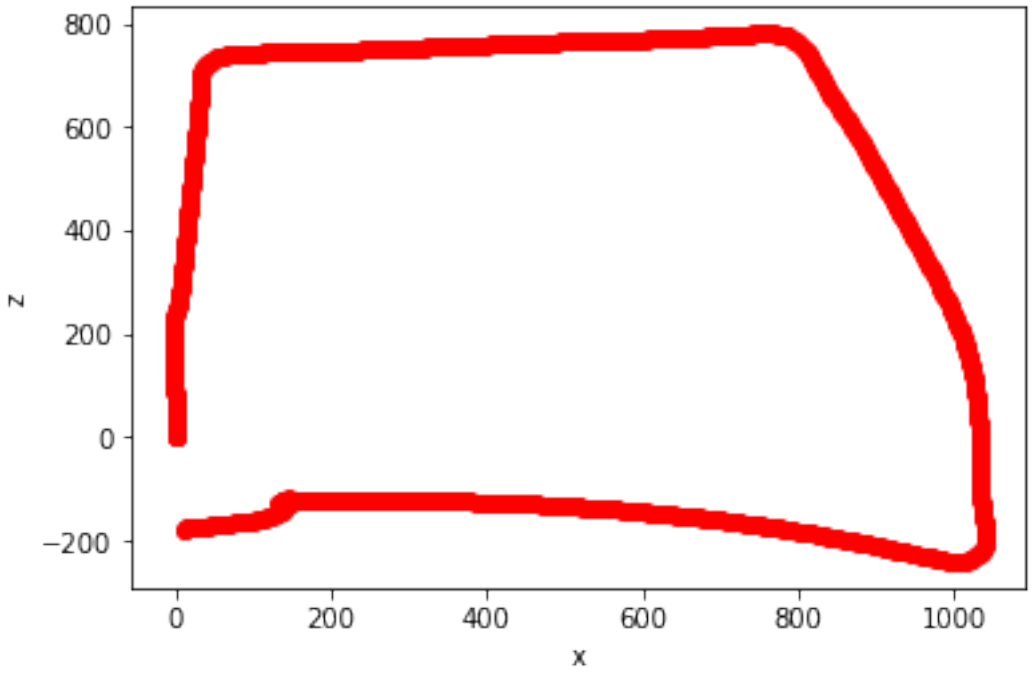

Figure 6: Camera Trajectory without using OpenCV functions