ENPM673 - Perception for Autonomous Robots

# Project 5

**Visual Odometry**

**Authors:**
Trevian Jenkins (116781381)
Karan Sutradhar (117037272)
Markose Jacob (117000269)

Date: May 4th, 2020

## Objective:

This project will estimate the 3D motion of the camera and the result will be a plot of the trajectory of the camera. This project involves the implementation of concepts of Visual Odometry.

## Theory:

Odometry basically means counting of steps,  and visual odometry is the process of incrementally estimating the position and orientation with respect to initial reference frames by tracking only the visual features.

Given an estimate of $\boldsymbol{R_k}$, $\boldsymbol{T_k}$ of the current camera pose as well as 3D points $\mathbf{X_P} = (X_p, Y_p, Z_p)$ and correspondences to calibrated point projections in frame $\mathbf{(k+1)(x_p^{k+1}, y_p^{k+1})}$

Updates to the pose $\boldsymbol{R_{k+1}}$, $\boldsymbol{T_{k+1}}$

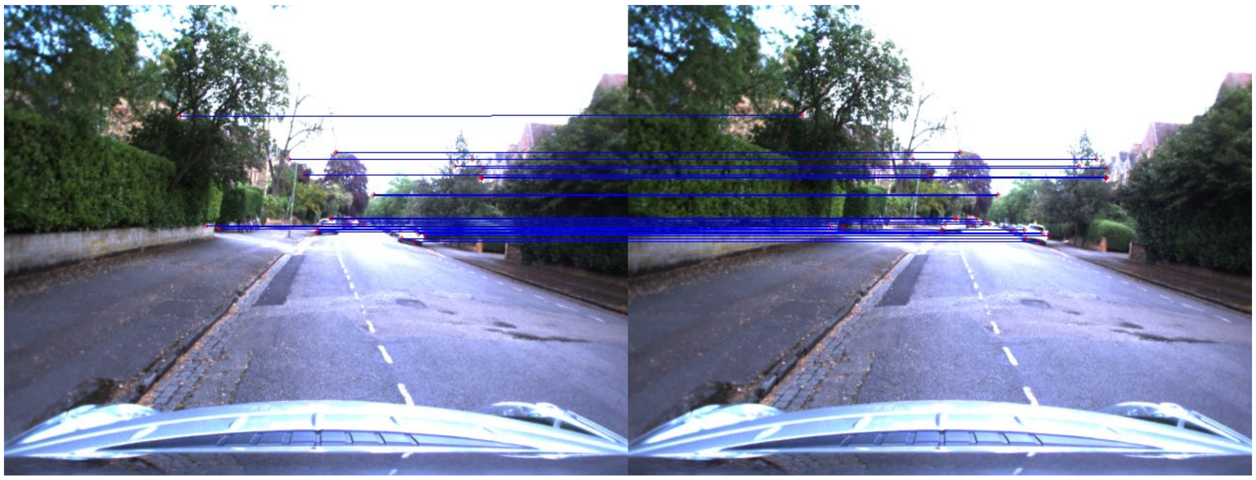
## Data Preparation:

Since the data is provided in Bayer format we had to convert the data set to color. We used the function cv2.COLOR_BayerGR2BGR for this. Next we undistorted the image, in order to undistort the image we used the python files provided to us (ReadCameraModel and UndistortImage). This new color image which is undistorted is saved and used in our final code. convertToColor.py file does all this for us. We are converting the images to color and saving these images to speed up our code.

Undistorted color image

## Oriented FAST and Rotated BRIEF (ORB):

We used ORB to detect the key points between two images. We decided to go with ORB as it is not patented and also faster than SIFT and SURF. ORB detects key points in an image and stores them, these keypoints are identified in another image by individually comparing each feature in both the images.



Feature matching using ORB. The left frame precedes the right frame in time.

## Methodology:

The K matrix containing the intrinsic camera parameters was provided from ReadCameraModel.py. We undistort the original Bayer-formatted images to get rid of the intrinsic warping. We then establish a base pose, which is a 4 x 4 identity matrix.

For the next section, we iterate through each frame, comparing <f0, f1>, then <f1, f2>, then <f2, f3>, and so on. Using OpenCV's ORB, we find the 30 best matches between the first and second image. We then use the matching keypoints to calculate the fundamental matrix. To calculate the fundamental matrix, we form a matrix *A*, indicated below, using 8 random keypoint correspondences:

$$\begin{bmatrix} x_1x'_1 & x_1y'_1 & x_1 & y_1x'_1 & y_1y'_1 & y_1 & x'_1 & y'_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_mx'_m & x_my'_m & x_m & y_mx'_m & y_my'_m & y_m & x'_m & y'_m & 1 \end{bmatrix}$$

Taking the SVD of A gives us the matrices U, S, and $V^T$. The 3 x 3 fundamental matrix *F* was gathered from the 9 elements of the last column of $V^T$. We take the SVD of this F matrix, replace the matrix of singular values with the diagonal matrix:

[1 0 0]
[0 1 0]
[0 0 0]

Using the U, S, and $V^T$ matrices, we recalculate F. We then normalize F using the root-sum-squared values of the difference between the points predicted using the F-matrix and the actual corresponding points. We recalculate the F matrix using several different sets of 8 random points, selecting the F matrix that has the most points within a given threshold.

After calculating the best F matrix, we use F and K to calculate the essential matrix E. We first calculate E with the equation $E = K^TFK$. We take the SVD of the matrix $K^TFK$, replacing the S matrix as done previously with the fundamental matrix, and multiply the matrices once again E. We take the SVD of the essential matrix, replacing S with the matrix W below:

$$\begin{matrix}[0 & -1 & 0] \\ [1 & 0 & 0] \\ [0 & 0 & 1]\end{matrix}$$

We get the rotations of the camera pose by multiplying the matrices from the previous SVD calculation together, using W instead of S. This calculation gives four camera rotations and four translations. We find the number of inliers using the distance equation:

$$d = [X\ Y\ 1]\ F\ [X'\ Y'\ 1]^T$$

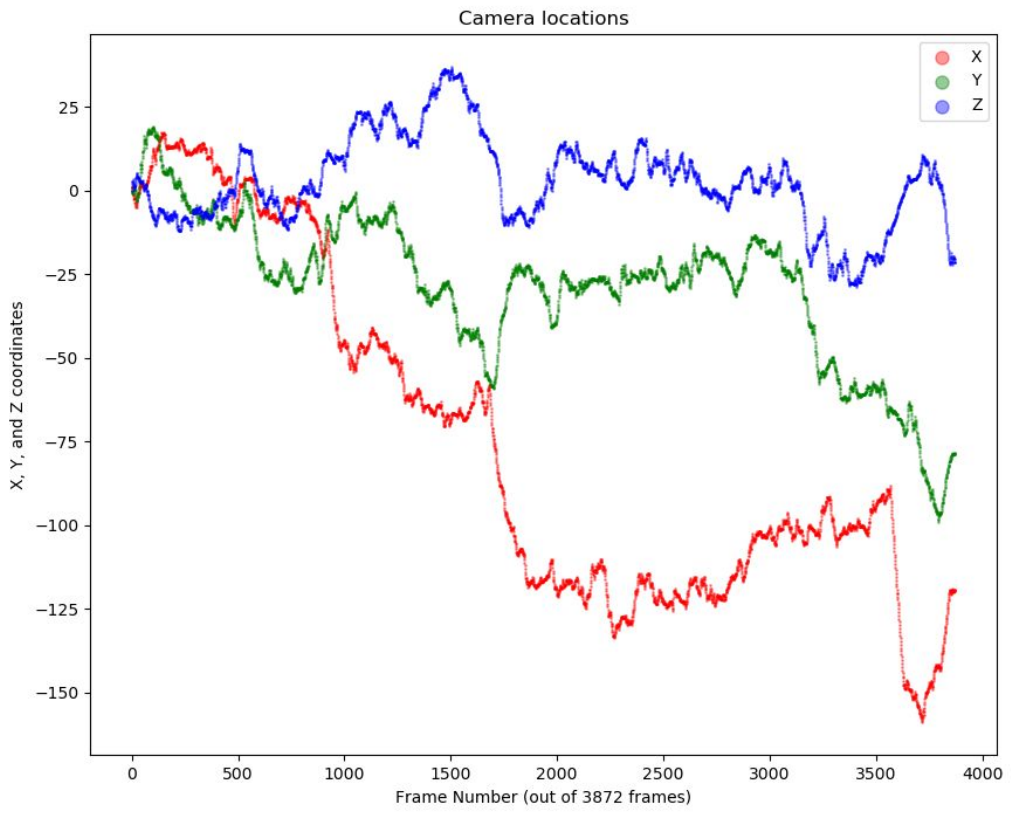
In the above equation, the correspondences are indicated using prime notation. All the point correspondences with a distance below a certain threshold are considered inliers, which are used to calculate the best camera pose. The base pose B is updated using the camera pose C:

$$B' = BC$$

Iteratively updating the base pose provides us with a trajectory of the camera to plot.

## Result:

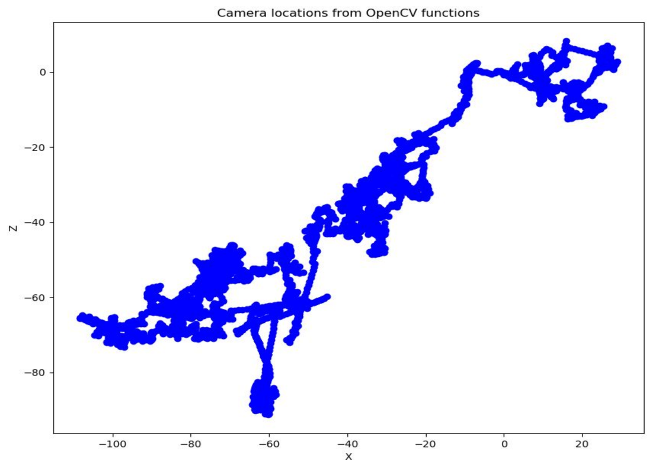
The location of the camera in each frame is plotted in the following figure.



## Extra Credits:

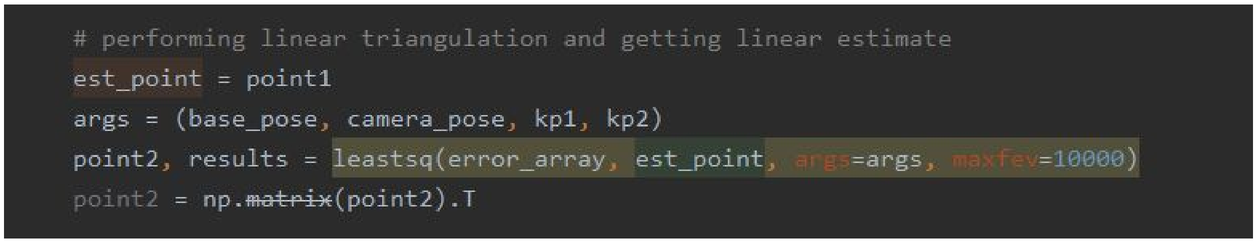
**Comparison:**

The inbuilt OpenCV functions as (cv2.findEssentialMat and cv2.recoverPose) used to find the rotation/translation parameters i.e., essential matrix and best camera pose. The camera trajectory is plotted and the result is shown below. The result drawn from both i.e., one from openCV and the other from our implementation, thus the plot results in accumulated drift in trajectory.



**Additional Steps in the Pipeline:**
Solved non-linearly for the depth and 3D motion using algebraic error for better estimate as mentioned below.
Estimation of depth using using non-linear triangulation
The linear triangulation method minimizes the algebraic error by finding the 3D points from the 2D image points. As the reprojection error is geometrically meaningful and can be computed by measuring error between measurement and projected 3D point.
This minimization is highly nonlinear due to the divisions, this minimization was solved using nonlinear optimization functions such as scipy.optimize.leastsq in Scipy library.

Estimation of  the rotation and translation non-linear triangulation using the Nonlinear PnP
The linear PnP minimizes algebraic error. Reprojection error that is geometrically meaningful is computed by measuring error between measurement and projected 3D point.
This minimization is highly nonlinear because of the divisions and quaternion parameterization.
This minimization was solved in the code using a nonlinear optimization function such as scipy.optimize.leastsq in Scipy library.
The code snippet is shown below as in our program.



```
# performing linear triangulation and getting linear estimate
est_point = point1
args = (base_pose, camera_pose, kp1, kp2)
point2, results = leastsq(error_array, est_point, args=args, maxfev=10000)
point2 = np.~~matrix~~(point2).T
```

References:
https://cmsc733.github.io/2019/proj/p3/#tri