University of Maryland College Park

Professional Master of Engineering

Robotics Engineering

ENPM673 - Perception for Autonomous Robots - Project 5

Students:
Akash Agarwal (116904636)
Nupur Nimbekar (116894876)
Anubhava Paras (116905909)

Lecturer:
Dr. Mohammed Samer Charifa

May 4, 2020

Link to the Problem Dataset

Output folder

In robotics and computer vision, visual odometry is the way towards deciding the position and orientation of a robot by analyzing the associated camera images.

In this project, we were given frames of a driving sequence taken by a camera in a car, and the scripts to extract the intrinsic parameters. And according to the instructions given, we have implemented all the steps to estimate the 3D motion of the camera, and provide as output a plot of the trajectory of the camera.

Preprocessing of Data:

The preprocessing of dataset involves three steps:

- Convert input image from Bayer encoded image to Colour image using:
  image = cv2.cvtColor(image, cv2.COLOR_BayerGR2BGR)
- Extract the camera parameters using ReadCameraModel.py as follows:
  fx, fy, cx, cy, G camera image, LUT = ReadCameraModel (' . / model ')
- Undistort the current frame and next frame using UndistortImage.py as follows:
  undistorted image = UndistortImage (original image ,LUT)

We have followed the same steps and accomplished the preprocessing of dataset by using imagepreprocessor.py which, imports parameters to efficiently follow every step for each frame and return a proper preprocessed dataset in general.

Part 1: The Basic Pipeline

Step 1:

To find the correspondence between successive frames we have use SIFT (Scale-Invariant feature Transform) as our keypoint algorithm. It is a feature detection algorithm used to detect and describe basic features in the image. We have implemented cropping considering that the camera is fixed on the bonnet of the car.

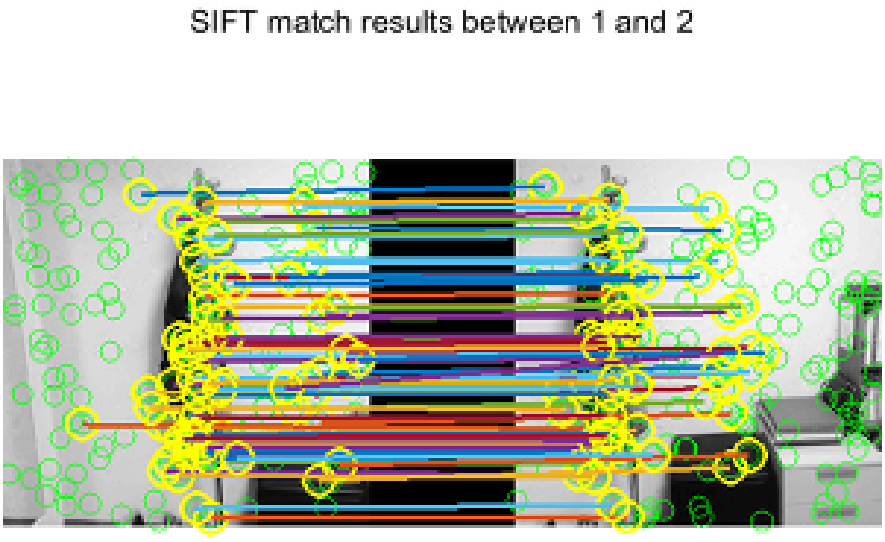


Figure 1: Representation SIFT keypoint matching

Here, keypoint descripters are created and used for keypoint matching. This way we can easily establish relation between successive frames.

Step 2 :

After finding the relation between the successive frames, our goal was to estimate the fundamental Matrix using Eight point algorithm within RANSAC.

Epipolar geaometry is a fundamental geometric relationship between two perspective cameras capturing the same object of interest. The essential and fundamental matrices are 3x3 matrices that "encode" the epipolar geometry of two views.

Since the fundamental matrix $\mathbf{F}$ is a 3×3 matrix determined up to an arbitrary scale factor, 8 equations are required to obtain a unique solution and for estimating the fundamental matrix, each point only contributes one constraint (row). Thus Eight-point algorithm is how we reach those 8 points which are required to find the fundamental matrix.

Homogeneous linear system in this case can be represented as:

$$\begin{bmatrix} x_i' & y_i' & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = 0$$

$$x_i x_i' f_{11} + x_i y_i' f_{21} + x_i f_{31} + y_i x_i' f_{12} + y_i y_i' f_{22} + y_i f_{32} + x_i' f_{13} + y_i' f_{23} + f_{33} = 0$$

Here **F** can be found by simplifying for m correspondences,

$$\begin{bmatrix} x_1x_1' & x_1y_1' & x_1 & y_1x_1' & y_1y_1' & y_1 & x_1' & y_1' \\ 1 & & & & & & & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & & & & & & & \\ x_mx_m' & x_my_m' & x_m & y_mx_m' & y_my_m' & y_m & x_m' & y_m' \\ 1 & & & & & & & \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{21} \\ f_{31} \\ f_{12} \\ f_{22} \\ f_{32} \\ f_{13} \\ f_{23} \\ f_{33} \end{bmatrix} = 0$$

We have used Sampson error method to calculate the inliers while implementing RANSAC using the following Formula.

$$\phi(F) = \sum_{j=1}^{n} \frac{\left(\tilde{\boldsymbol{x}}_2^{j\top} F \tilde{\boldsymbol{x}}_1^{j}\right)^2}{\left(F\tilde{\boldsymbol{x}}_1^{j}\right)_1^2 + \left(F\tilde{\boldsymbol{x}}_1^{j}\right)_2^2 + \left(F^\top\tilde{\boldsymbol{x}}_2^{j}\right)_1^2 + \left(F^\top\tilde{\boldsymbol{x}}_2^{j}\right)^2}$$

Step 3:

In this step we aim to find essential matrix from the fundamental matrix by accounting for calibration parameters.Essential matrix is another 3×3 matrix, but with some additional properties that relates the corresponding points assuming that the cameras obeys the pinhole model (unlike **F**).

More specifically, $\mathbf{E} = \mathbf{K^T F K}$ where **K** is the camera calibration matrix or camera intrinsic matrix. Clearly, the essential matrix can be extracted from **F** and **K**.

Finally Essential matrix can be constructed as,

$$\mathbf{E} = U \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T$$

Step 4:

If we consider that the camera pose has 6 DoF i.e. (Rotation (Roll, Pitch, Yaw), Translation (X, Y, Z)) and **E** matrix is identified by the four camera pose configurations i.e

$(C_1, R_1), (C_2, R_2), (C_3, R_3), (C_4, R_4)$, where we have computable $C \in \mathbb{R}^3$ camera center and $R \in SO(3)$ rotation matrix.

Then from the known equation for $\mathbf{E} = \mathbf{UDV^T}$ and $W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ we can calculate the camera poses as:

1. $C_1 = U(:, 3)$ and $R_1 = UWV^T$
2. $C_2 = -U(:, 3)$ and $R_2 = UWV^T$
3. $C_3 = U(:, 3)$ and $R_3 = UW^TV^T$
4. $C_4 = -U(:, 3)$ and $R_4 = UW^TV^T$

For $\mathbf{AX} = \mathbf{0}$, we can calculate the X for 3D triangulation by the following equations,

$$\begin{bmatrix} y\boldsymbol{p}_3^\top - \boldsymbol{p}_2^\top \\ \boldsymbol{p}_1^\top - x\boldsymbol{p}_3^\top \\ y'\boldsymbol{p}_3'^\top - \boldsymbol{p}_2'^\top \\ \boldsymbol{p}_1'^\top - x'\boldsymbol{p}_3'^\top \end{bmatrix} \boldsymbol{X} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Note that while calculating Camera Poses, $det(R) = 1$, the value for $det(R) = -1$, then the Camera Poses should also be modified.

Step 5:

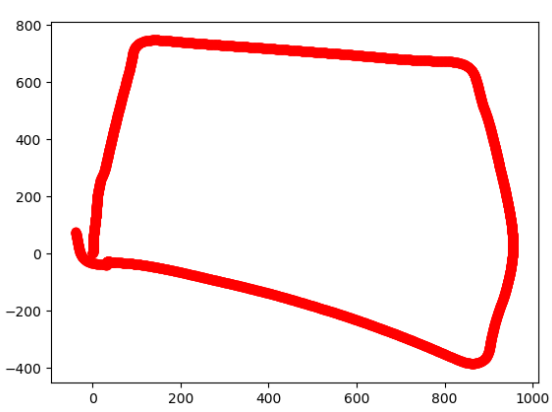
The task for this step was to find the correct T and R from the depth positivity. That is, estimate for each of the four solutions the depth of all points linearly using the cheirality equationsand then choose the R and T which gives the largest amount of positive depth values.

Given two camera poses, (C_1, R_1) and (C_2, R_2), and correspondences, $x_1 \leftrightarrow x_2$, 3D points can be triangulated using linear least squares. The cheirality conditions states that the reconstructed points must be in front of the cameras in order to find the correct unique camera pose.

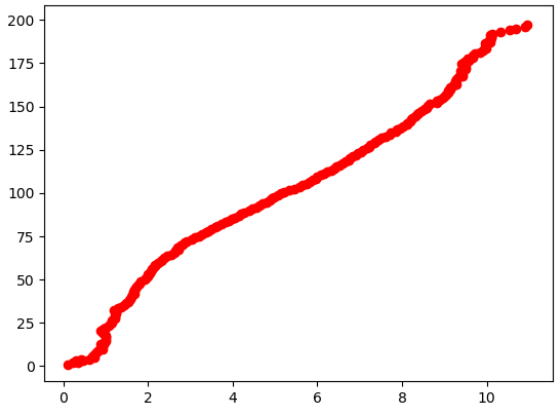
To check the cheirality condition, triangulate the 3D points (given two camera poses) using linear least squares to check the sign of the depth **Z** in the camera coordinate system w.r.t. camera center.

Step 6:

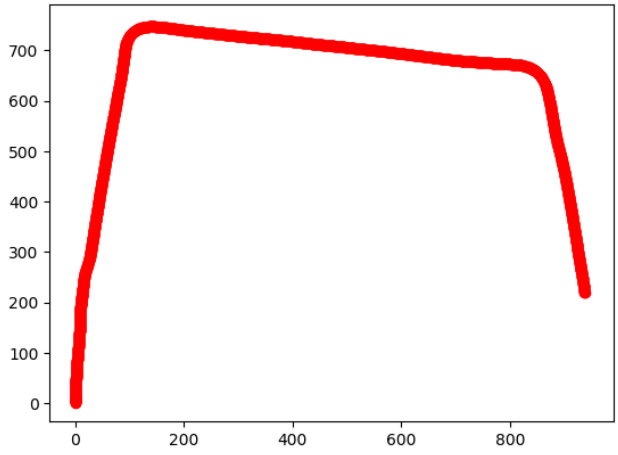
After we compute R and T the poses will be updated for the next frame. And that can be computed by the dot product between old pose and [R | T] matrix. Figure 2. shows the output of the implementation of basic pipeline constructed and coded for the project.



(a) Frame 3794



(b) Frame 227



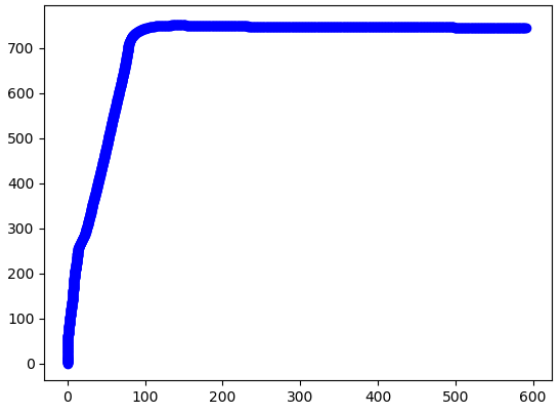
(c) Frame 2000

Figure 2: Plot for random frames from the implemented code

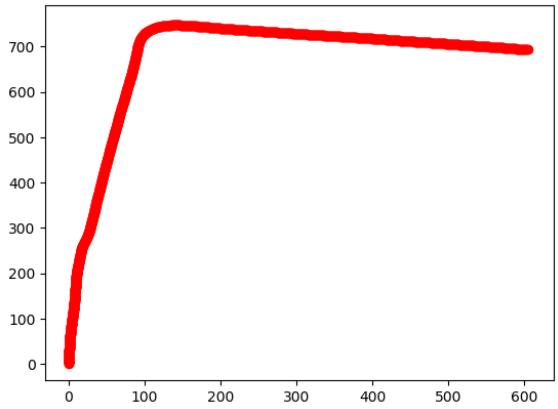
Part 2: Comparison

We wrote a code using cv2.findEssentialMat and cv2.recoverPose from opencv to compare our resulting data for rotation and translation parameters with the ones obtained from inbuilt functions.

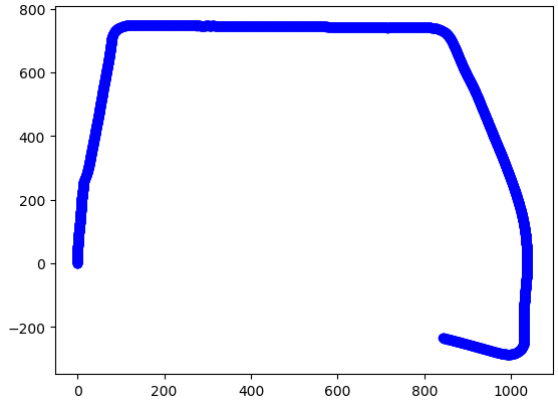
Figure 3. portrays the comparison between the data plotted for randomly selected 2 frames:



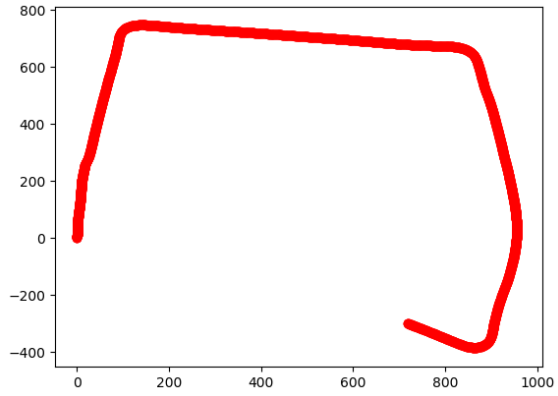
(a) cv2: Data plotted for frame 1300



(b) Implementation: Data plotted for frame 1300



(c) cv2: Data plotted for frame 2800



(d) Implementation: Data plotted for frame 2800

Figure 3: Comparison of data plotted for frames

The images from Figure 4 are captured from output video which reflects how the data is computed in inbuilt and implemented code.

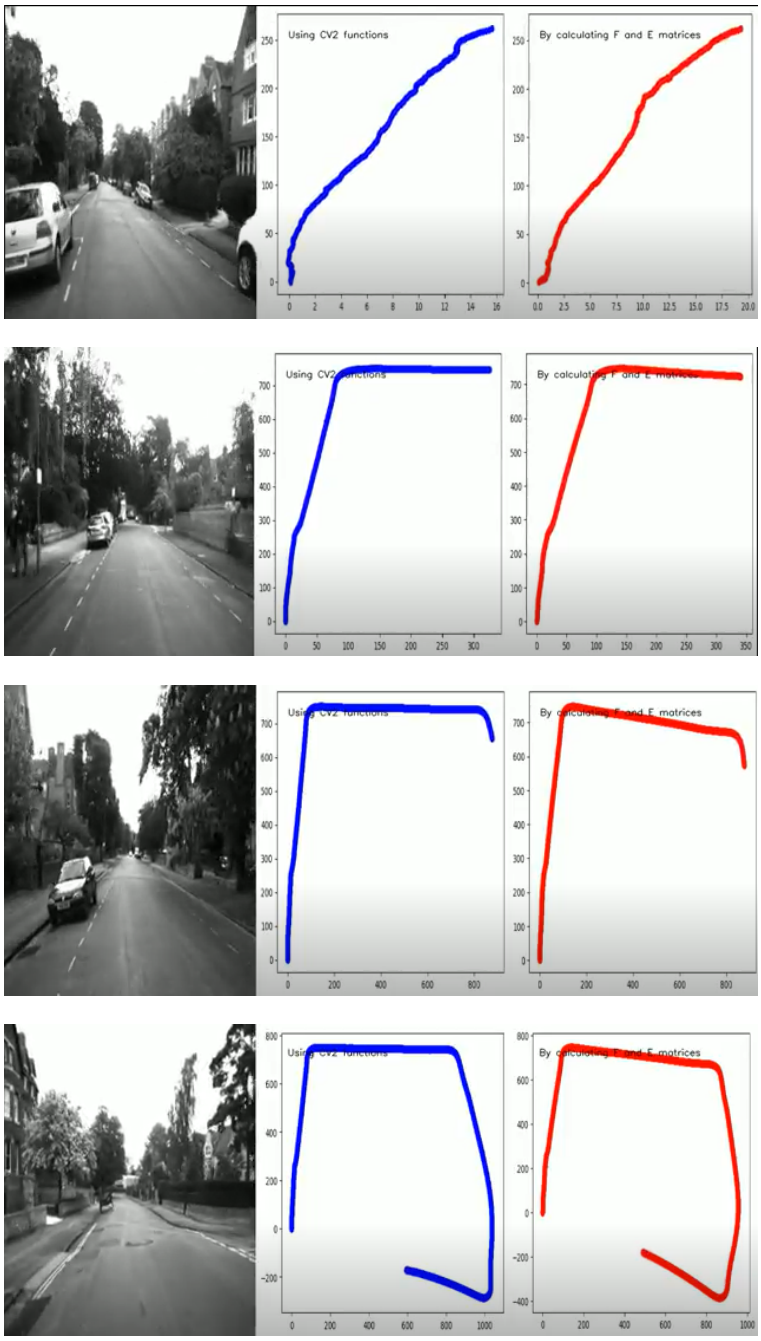


Figure 4: Comparison of calculated parameters with inbuilt function

Part 3: Non-Linear Triangulation

The linear triangulation minimizes the algebraic error. But, given the camera poses and linearly triangulated points, **X**, the points can be refined to obtain minimized reprojection error.

Geometrically this reprojection error can be calculated by measuring error between measured and 3D projected points, this can be done by,

$$\min_{x} \sum_{j=1,2} \left(u^j - \frac{P_1^{jT} \widetilde{X}}{P_3^{jT} X} \right)^2 + \left(v^j - \frac{P_2^{jT} \widetilde{X}}{P_3^{jT} X} \right)^2$$

which is,

$$\min_{C,R} \sum_{i=1,J} \left(u^j - \frac{P_1^{jT} \widetilde{X}_j}{P_3^{jT} \widetilde{X}_j} \right)^2 + \left(v^j - \frac{P_2^{jT} \widetilde{X}_j}{P_3^{jT} X_j} \right)^2$$

Given $N \geq 6$ 3D-2D correspondences, X, and linearly estimated camera pose, (C, R), refine the camera pose that minimizes reprojection error.

A compact representation of the rotation matrix using quaternion is a better choice to enforce orthogonality of the rotation matrix, R=R(q) where q is four dimensional quaternion, i.e.,

$$\min_{C,q} \sum_{i=1,J} \left(u^j - \frac{P_1^{jT} \widetilde{X}_j}{P_3^{jT} \widetilde{X}_j} \right)^2 + \left(v^j - \frac{P_2^{jT} \widetilde{X}_j}{P_3^{jT} X_j} \right)^2$$

This minimization is highly nonlinear because of the divisions and quaternion parameterization. The initial guess of the solution, (C_0, R_0), estimated via the linear PnP is needed to minimize the cost function. This minimization can be solved using a nonlinear optimization function such as **least_squares**

Figure 5. and 6. better explain the code written to implement Non-Linear Triangulation:

```python
def get_nonlinear_triangulated_points(points_3D, pose, point_list1, point_list2):
    P = np.eye(3,4)
    P_dash = pose
    tot_points = len(points_3D)
    approx_points = []
    for p1, p2, point_3D in zip(point_list1, point_list2, points_3D):
        point_3D_approx = least_squares(nlt_error, point_3D, args = (P, P_dash, point_list1, point_list2))
        approx_points.append(point_3D_approx)
    return np.array(approx_points)


def nlt_error(point_3D, P1, P2, points_left, points_right):
    X, Y, Z = point_3D
    point_3D = np.array([X, Y, Z, 1])

    p1 = P1 @ point_3D
    p2 = P2 @ point_3D

    projected_x1, projected_y1 = p1[0]/p1[2], p1[1]/p1[2]
    projected_x2, projected_y2 = p2[0]/p2[2], p2[1]/p2[2]

    dist1 = (points_left[0] - projected_x1)**2 + (points_left[1] - projected_y1)**2
    dist2 = (points_right[0] - projected_x2)**2 + (points_right[1] - projected_y2)**2

    error = dist1 + dist2
    return error
```

Figure 5: Non-linear triangulation to get a better approximation of the 3D points, given a pose and the corresponding points

```python
def get_approx_pose_by_non_linear_pnp(points_3D, pose, point_list1, point_list2):
    pose = np.reshape(pose, (-1,1)).T
    approx_pose = least_squares(non_linear_pnp_error, pose, args = (P, points_3D, point_list1, point_list2))
    return np.reshape(approx_pose, (3,4))

def non_linear_pnp_error(pose, points_3D, points_left, points_right):
    P1 = np.eye(3,4)
    P2 = np.reshape(pose, (3,4))

    ## we can have R in the form of R(q), i.e quaternions, to enforce orthogonality for better results

    X, Y, Z = point_3D
    point_3D = np.array([X, Y, Z, 1])

    p1 = P1 @ point_3D
    p2 = P2 @ point_3D

    projected_x1, projected_y1 = p1[0]/p1[2], p1[1]/p1[2]
    projected_x2, projected_y2 = p2[0]/p2[2], p2[1]/p2[2]

    dist1 = (points_left[0] - projected_x1)**2 + (points_left[1] - projected_y1)**2
    dist2 = (points_right[0] - projected_x2)**2 + (points_right[1] - projected_y2)**2

    error = dist1 + dist2
    return error
```

Figure 6: Non linear PnP to get better approximation of the pose, given approximated 3D points and their corresponding points in the image

For the implementation of Non Linear Triangulation and Non Linear PnP, please refer to the google drive link provided above.