Visual Odometry

Arjun Srinivasan Ambalam, Praveen Menaka Sekar, Arun Kumar Dhandayuthabani

1 Introduction

Visual Odometry (VO) is defined as the process of estimating the robot motion (translation and rotation with respect to a reference frame) by observing a sequence of images of its environment. VO is a particular case of Structure From Motion (SFM) that tackles the problem of 3D reconstruction of both the structure of the environment and camera poses from sequentially ordered or unordered image sets. In this project we implement the pipeline for Visual Odometry (VO) from scratch on the Oxford dataset given, and compare it with the implementation using OpenCV built-in functions.

2 Visual Odometry Pipeline

2.1 Dataset Preperation

1. We are given frames of driving sequences taken by a camera in a car in Bayer format i.e. the image has incomplete color samples.
2. In order to carry on with our pipeline, we need to reconstruct full color image from the incomplete color samples present in Bayer format.
3. We demosaic the Bayer image using **cv2.cvtColor(img, cv2.COLOR_BayerGR2BGR)** to obtain fully reconstructed color image.
4. This fully constructed color image is distorted, hence we utilize the camera parameters to undistort the image.
5. We extract camera parameters using the given **ReadCameraModel.py** and then undistort the image using the given **UndistortImage.py**.

2.2 Feature Detection and Matching

1. Features are specific structures in an image which helps in solving certain applications.
2. Applications like SFM, VO, SLAM require localized features or keypoint features i.e. features present in certain locations such as building corners, trees.
3. Hence, it is important to detect such features in order to move further down the pipeline.
4. In this project, for 2 successive frames we tried detecting both **SIFT (Scale-Invariant Feature Transform)** and **ORB (Oriented Fast and Rotated Brief)** features using built-in OpenCV functions ,
   - **cv2.xfeatures2d.SIFT_create()**
   - **cv2.ORB_create()**
5. Once the features and their descriptors have been extracted from two successive frames, we establish some preliminary feature matches between these images.
6. We use FLANN based matcher from OpenCV to perform feature matching in our pipeline.

2.3 Fundamental Matrix Estimation

The fundamental matrix, denoted by F, is a 3×3 (rank 2) matrix that relates the corresponding set of points in two images from different views (or stereo images). The F matrix is only an algebraic representation of epipolar geometry and hence, we can obtain a correspondence equation called the epipolar constraint,

$$X^{'}{}_{\mathrm{i}}{}^{\mathrm{T}} F X_{\mathrm{i}} = 0 \tag{1}$$

$$\begin{bmatrix} x^{'}{}_{\mathrm{i}} & y^{'}{}_{\mathrm{i}} & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} x_{\mathrm{i}} \\ y_{\mathrm{i}} \\ 1 \end{bmatrix} = 0 \tag{2}$$

Simplifying for m correspondences,

$$\begin{bmatrix} x_1 x^{'}{}_1 & x_1 y^{'}{}_1 & x_1 & y_1 x^{'}{}_1 & y_1 y^{'}{}_1 & y_1 & x^{'}{}_1 & y^{'}{}_1 & 1 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ x_{\mathrm{m}} x^{'}{}_{\mathrm{m}} & x_{\mathrm{m}} y^{'}{}_{\mathrm{m}} & x_{\mathrm{m}} & y_{\mathrm{m}} x^{'}{}_{\mathrm{m}} & y_{\mathrm{m}} y^{'}{}_{\mathrm{m}} & y_{\mathrm{m}} & x^{'}{}_{\mathrm{m}} & y^{'}{}_{\mathrm{m}} & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{21} \\ f_{31} \\ f_{12} \\ f_{22} \\ f_{32} \\ f_{13} \\ f_{23} \\ f_{33} \end{bmatrix} = 0 \tag{3}$$

Each point only contributes one constraints as the epipolar constraint is a scalar equation. Thus, we require at least 8 points to solve the above homogeneous system i.e. eight point algorithm.

2.3.1 Eight-point algorithm

With $\mathrm{N} \geq 8$ correspondences between two frames, the fundamental matrix, F can be obtained as:

1. Stack the above equation in a matrix A to obtain Ax=0.
2. Apply SVD to matrix A and obtain U, S, V.
3. The singular values σ_{i} where $\mathrm{i} \in [1,9], \mathrm{i} \in \mathrm{Z}$, are positive and are in decreasing order with $\sigma_9 = 0$ since we have 8 equations for 9 unknowns. Thus, the last column of V is the true solution given that $\sigma_{\mathrm{i}} \neq 0$.
4. However, due to noise in the correspondences, the estimated F matrix can be of rank 3 i.e. $\sigma_9 \neq 0$. If F has a full rank then it will have an empty null-space i.e. it won't have any point that is on entire set of lines.
5. So, to enforce the rank 2 constraint, set last singular value of the estimated F to zero.

However, in order to improve the accuracy of epipolar lines generated, we recenter the point correspondences $\mathrm{X_i} = (\mathrm{x_i}, \mathrm{y_i})$ and $\mathrm{X^{'}_i} = (\mathrm{x^{'}_i}, \mathrm{y^{'}_i})$ in both frames to their respective centroids. After recentering the image points, we must scale the points to be a fixed squared distance from the origin. Using these normalized points we estimate the fundamental matrix.

2.4 Outlier Rejection using RANSAC

Since the point correspondences are computed using some feature descriptors, the data is bound to be noisy and contains several outliers. Thus, to remove these outliers, we use RANSAC algorithm to obtain a better estimate of the fundamental matrix. The F matrix with maximum number of inliers is chosen.

Algorithm 1: RANSAC

Result: Best F matrix, inlier features
n = 0 ;
for *i = 1 : M* **do**
  Choose 8 correspondences x_1, x_2 randomly ;
  Estimate Fundamental Matrix F ;
  S = 0 ;
  for *j = 1 : N* **do**
    if $|x'_{2j}{}^T F x_{1j}| < \epsilon$ **then**
      S ∪ j ;
    end
  end
end
if *n <* | *S* | **then**
  n = | S | ;
  S_{in} = S ;
end

2.5 Essential Matrix Estimation

We can then find the relative camera poses between the two images using the Essential Matrix, E. 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).

$$E = K^{\mathrm{T}} F K \tag{4}$$

Where, K is camera calibration matrix. Due to noise in K, singular values of E are not necessarily [1, 1, 0]. This can be corrected as follows,

1. We first apply SVD to E and obtain U, S, V.
2. Then substitute S by W which is defined as shown below,

$$W = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{5}$$

3. Recompute E using U, W, and V.

$$E = UWV^{\mathrm{T}} = U \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^{\mathrm{T}} = 0 \tag{6}$$

2.6 Extract Camera Pose

Since the E matrix is identified, the four camera pose configurations: (C_1,R_1),(C_2,R_2),(C_3,R_3) and (C_4,R_4) can be computed.

1. First we apply SVD to E and obtain U, S, V.
2. Then substitute S by W which is defined as shown below,

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7}$$

3. The four configurations can then be computed as shown below,

   - C_1 = U[:, 3], R_1 = UWV^T
   - C_2 = -U[:, 3], R_2 = UWV^T
   - C_3 = U[:, 3], R_3 = UW^TV_T
   - C_4 = -U[:, 3], R^4 = UW^TV^T

It is important to note that the det(R) = 1. If det(R) = -1, the camera pose must be corrected i.e. C = -C and R = -R.

2.7 Linear Triangulation

Now that we know the position of a point in both the frames along with the camera calibration and pose, our next goal is to compute the position of the point in space i.e. 3D coordinates of the point. In order to obtain this information we perform triangulation. For this project we have done both linear and non linear triangulation. Here we discuss about linear triangulation. This problem is basically a least squares minimization problem which can be formulated as shown below, Given:

$$x = PX \; x^{'} = P^{'}X \tag{8}$$

Where,

- x, $x^{'}$ - position of points in both frames
- P, $P^{'}$ - projection matrix pertaining to both frames

We then can write the above given information as linear equations in X,

$$\begin{bmatrix} xP^{3T} - P^{1T} \\ yP^{3T} - P^{2T} \\ x^{'}P^{'3T} - P^{'1T} \\ y^{'}P^{'3T} - P^{'2T} \end{bmatrix} X = 0 \tag{9}$$

We then apply SVD to the above equation and solve for X. The last column of V obtained from the SVD is the solution for X.

2.8 Triangulation Check for Cheirality Condition

Now that we have our 3D points and 4 possible camera pose configurations, we need to find the unique camera pose that satisfies the Cheirality condition. The Cheirality condition intuitively tells that the reconstructed points must be in front of the cameras. A 3D point X is in front of the camera if,

$$r_3(X - C) > 0 \tag{10}$$

$$X[3] >= 0 \; (\textit{depth of 3D point is positive}) \tag{11}$$

Where, r_3 is the third row of rotation matrix representing the z axis of the camera. Not all triangulated points satisfy this condition due of the presence of correspondence noise. The best camera configuration, (C,R,X) is the one that produces the maximum number of points satisfying the Cheirality condition.

2.9 Plot Camera Center

The final part of the pipeline is to plot the unique camera centers for each frame using the unique rotation matrices obtained. We do this by finding the composite homogeneous transformation (H) from one frame to another and using it to find the projection of point p_0 pertaining to the first frame. We consider p_0 to be origin i.e.,

$$p_0 = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} \tag{12}$$

$$H_{\text{i}} = H_{(\text{i - 1})}p_0 \tag{13}$$

$$Where \; H_{(\text{i - 1})} = H_{(\text{i - 2})}H_{(\text{i - 3})}....H_{(1)} \tag{14}$$

3 OpenCV vs Custom Implementation Performance Comparison

The drift between OpenCV implementation and custom implementation is evident from the below plot. We calculated the drift by finding squared difference between the points obtained in OpenCV implementation and custom implementation.

$$accumulated\ drift = \sqrt{(\mid X - X' \mid)^2 + (\mid Y - Y' \mid)^2} \tag{15}$$

Where,

- X, Y - point from custom implementation
- X', Y' - point from OpenCV implementation

Hence, the accumulated drift in trajectory was found to be **901164.0053000601**.

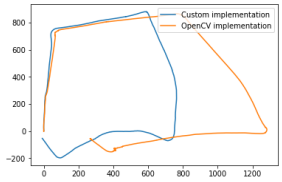

Figure 1: OpenCV vs Custom Implementation

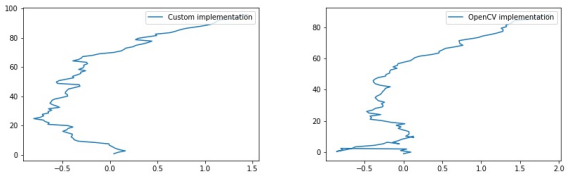

Figure 2: OpenCV vs Custom Implementation for frame 20

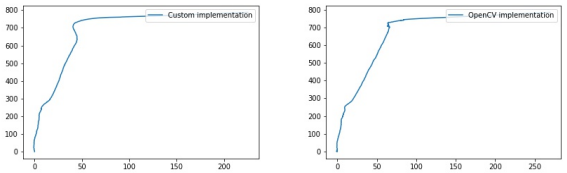

Figure 3: OpenCV vs Custom Implementation for frame 100

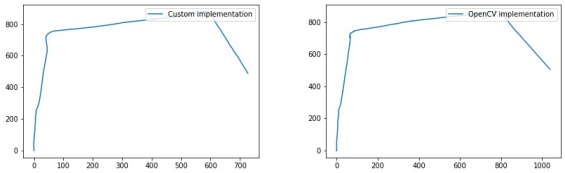

Figure 4: OpenCV vs Custom Implementation for frame 1000

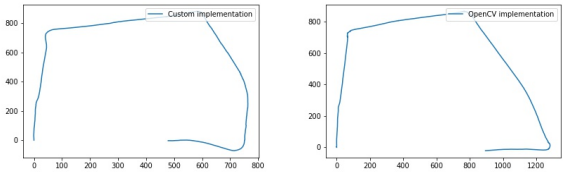

Figure 5: OpenCV vs Custom Implementation for frame 1500

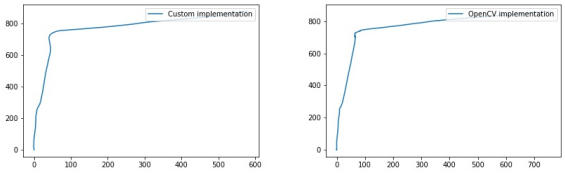

Figure 6: OpenCV vs Custom Implementation for frame 2000

4 Non Linear Triangulation

As mentioned earlier we also implemented non linear triangulation in our pipeline. The linear triangulation minimizes algebraic error which is not meaningful or intuitive and also, it is not optimal. Hence, we implement non linear triangulation minimizing reprojection error which is geometrically meaningful. The reprojection error can be calculated as shown below,

$$\underset{x}{\text{minimize}} \; \Sigma_{\mathrm{j}\,=\,1,\,2}(u^{\mathrm{j}} - \frac{P^{\mathrm{jT}}{}_1\hat{X}}{P^{\mathrm{jT}}{}_3X})^2 + (v^{\mathrm{j}} - \frac{P^{\mathrm{jT}}{}_2\hat{X}}{P^{\mathrm{jT}}{}_3X})^2 \tag{16}$$

Where,

- j - index of each camera
- $\hat{X}$ - homogeneous representation of X
- $\mathrm{P^T}_{\mathrm{i}}$ - each row of camera projection matrix

The initial guess of the solution, X_0, is estimated via the linear triangulation to minimize the cost function. We used **scipy.optimize.least_squares** built-in function to solve this non linear optimization problem.

5 Non Linear PnP

As we have a set of n 3D points in the world, the 6 DOF camera pose can be estimated using linear least squares. This fundamental problem, is known as Persepective-n- Point (PnP) and for there to exist a solution,

n3. The non linear PnP minimizes the reprojection error with respect to camera pose as follows,

$$\underset{C,R}{\text{minimize}} \; \Sigma_{\mathrm{j}\,=\,1,\,\mathrm{J}}(u^{\mathrm{j}} - \frac{{P^{\mathrm{j}\mathrm{T}}}_1 \hat{X}}{{P^{\mathrm{j}\mathrm{T}}}_3 X})^2 + (v^{\mathrm{j}} - \frac{{P^{\mathrm{j}\mathrm{T}}}_2 \hat{X}}{{P^{\mathrm{j}\mathrm{T}}}_3 X})^2 \tag{17}$$

The initial guess of the solution, C_0, R_0 is estimated via the linear PnP to minimize the cost function. We used **scipy.optimize.least_squares** built-in function to solve this non linear optimization problem.