Towards Understanding a Basic Stereo VO SLAM Framework by Building It from Scratch

Akshay Badagabettu
Carnegie Mellon University
Email: abadagab@andrew.cmu.edu

Shubhra Aich
Carnegie Mellon University
Email: saich@andrew.cmu.edu

Sai Sravan Yarlagadda
Carnegie Mellon University
Email: saisravy@andrew.cmu.edu

I. Aim/Motivation

In this project, instead of playing with any particular SLAM implementation or trying out a new idea, we have decided to understand the engineering implementation of a basic SLAM framework by developing it in Python from scratch. This is because from the initial exploratory search, we perceive that none of the SLAM implementations are essentially the same which oftentimes makes it difficult to choose just one implementation over others and build on top of that. Therefore, we have decided to build a simple one from scratch to get a complete understanding of the subtleties associated with building a basic pipeline. We believe this will provide us a better foundational knowledge to pursue cutting edge research on SLAM in the future

Technically, we plan to implement a simple stereo Visual Odometry SLAM on the KITTI odometry dataset. The reason behind choosing stereo is twofold. First is the strong dependency on the stereo camera information for the projects with which the authors are affiliated. Second is the simplicity of the stereo vision in terms of implementation and initialization.

Following the classic SLAM framework, we will have two main modules in our framework – frontend and backend. A high-level overview of the features that we plan to include in these modules is briefed below:

II. Front End

A. Dataloader

A calibrated stereo pair as well as the left camera image of the next frame, is taken as the input to our pipeline. From the calibration file provided in the dataset, we get our left camera intrinsic matrix.

$$K_{left} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 718.856 & 0 & 607.1928 \\ 0 & 718.856 & 185.2157 \\ 0 & 0 & 1 \end{bmatrix}$$

B. Feature detection and matching

There are many algorithms for feature detection, such as SIFT, BRIEF, FAST, and ORB. SIFT and ORB stand out from these as they are scale and rotation invariant. We make use of the OpenCV library to implement these algorithms. We have written the code for both SIFT and ORB in the pipeline, and based on the method defined in the main file, the feature detector is selected. Lowe's ratio test can be used if required to keep only the best features to reduce computational time. After the implementation and testing of both algorithms, we saw that ORB is able to keep up with the 10fps speed of KITTI dataset and hence can be used for real time operation but SIFT is approximately at 2fps.

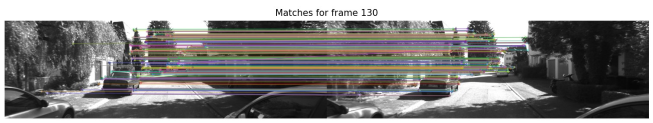

Fig. 1: Feature matches in left images of frame 130 and 131

C. Disparity and depth maps

Disparity is nothing but the distance between two corresponding points in the left and right images of a single frame. By drawing out the stereo camera position and using similar triangles, we can formulate depth as a function of focal length, baseline, and disparity.

$$Depth(Z) = \frac{focal\ length * baseline}{disparity} = \frac{f * b}{d}$$

We then loop over all the matched keypoints from two successive left camera images (frame k and k+1) and compute the x, y, and z points in the camera's coordinate frame. Let (u, v) be the image pixel locations in the left camera image of frame k.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \frac{(u-c_x)*Z}{f_x} \\ \frac{(v-c_y)*Z}{f_y} \\ Z \end{bmatrix}$$

D. Minimizing Reprojection Error

The above step, where we did 3D-2D correspondences, brings in errors, and the extrinsic matrix needs to be calculated to minimize this reprojection error. The 3D points that we computed in the above section using the (u, v) values of the left camera of frame k and the depth using both cameras of frame k are projected onto the left camera of frame k+1. These projections are error-prone, and the extrinsic matrix that minimizes the error between the projected pixel locations and actual pixel locations is to be found. Let (u, v) be the actual pixel locations of the matched keypoints, (Xw, Yw, Zw) be the 3D points we found from the above section. Then,

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \prod T_w \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

where K is the intrinsic matrix as defined before, and the projection matrix is defined as

$$\prod = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

And Tw is the extrinsic matrix.

$$T_w = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We use OpenCV's PnPRansac() function to solve this least-squares problem. A minimum of 4 points is required to solve this problem; the more good points, the better, but it is solvable even with 3 points. This gives us 4 solutions, and the best solution is to be chosen from them. However, in our case, we do not need to worry about the lack of points.

As it is evident by the above equations, the inverse of the extrinsic matrix will transform the 2D feature keypoints into the world frame coordinates. This is fine for the first two frames, but as we are considering the first camera frame to be the world coordinate frame, we need the dot product of all the inverses of the extrinsic matrix calculated up until that frame. The pose can then be extracted from this matrix where the translation part of the matrix gives us the (x,y,z) coordinates of the camera, and the roll, pitch, and yaw can be calculated as shown below.

$$\begin{bmatrix} roll \\ pitch \\ yaw \end{bmatrix} = \begin{bmatrix} \arctan 2(r_{21}, r_{11}) \\ \arctan 2(-r_{31}, \sqrt{r_{32}^2 + r_{33}^2}) \\ \arctan 2(r_{32}, r_{33}) \end{bmatrix}$$

III. Backend

In this midterm report, we describe the almost complete sketch of our backend covering both the theory we learned in the class and some high-level practical aspects of the future implementation that will be done before the project deadline.

A. Theory

Assuming there are M landmarks in the complete environment, the state of our environment at time t is given by $\mathbf{x}_t = \{p_t, l_1, \ldots, l_M\}$ where p_t is the pose of the robot at time t and $\forall j \in \{1, \ldots, M\}$ l_j denote the landmark positions in 3D which are constant.

The SLAM problem is formulated as follows [1]:

$$\mathbf{x}_t = g(\mathbf{x}_{t-1}, \mathbf{u}_t) + \omega_t$$
$$\mathbf{z}_t = h(\mathbf{x}_t) + \nu_t$$

Here, the first equation represents the motion model and the second one depicts the measurement model. Also, $\mathbf{z}_t$ is the measurement received at time t, and $\mathbf{u}_t$ is the control action taken at time t. Moreover, ω_t and ν_t are the additive process and measurement noise terms, respectively. The posterior is, therefore, can be expressed as follows [1]:

$$p(\mathbf{x}_t|\mathbf{x}_0, \mathbf{u}_{1:t}, \mathbf{z}_{1:t}) = \eta p(\mathbf{z}_t|\mathbf{x}_t) p(\mathbf{x}_t|\mathbf{x}_0, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})$$

where η is the normalizer which is constant in terms of optimization. Applying the Markov assumption that the state at time $t-1$ is a sufficient representation to predict the state at time t, we can simplify the above equation as:

$$\begin{aligned} & p(\mathbf{x}_t|\mathbf{x}_0, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1}) \\ & = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1}) p(\mathbf{x}_{t-1}|\mathbf{x}_0, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1}) d\mathbf{x}_{t-1} \\ & = \int p(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{u}_t) p(\mathbf{x}_{t-1}|\mathbf{x}_0, \mathbf{u}_{1:t-1}, \mathbf{z}_{1:t-1}) d\mathbf{x}_{t-1} \end{aligned}$$

Here, $p(\mathbf{x}_{t-1}|\mathbf{x}_0, \mathbf{u}_{1:t-1}, \mathbf{z}_{1:t-1})$ is the distribution at time $t-1$. Note that $p(\mathbf{x}_t|\mathbf{x}_0, \mathbf{u}_{1:t}, \mathbf{z}_{1:t-1})$ is our state estimation prior to incorporating the measurement and $p(\mathbf{x}_t|\mathbf{x}_0, \mathbf{u}_{1:t}, \mathbf{z}_{1:t})$ is the posterior after the measurement is taken into account.

Different realizations exist for the above recursive Bayesian estimation problem in the literature, including the linear Gaussian case of Kalman filter, and its nonlinear variants, such as extended Kalman filter (EKF), unscented Kalman filter (UKF), etc. EKF works well in practice despite its limiations with the linearization via first order Taylor expansion. However, if the number of landmarks is large, EKF turns out to be expensive, and so not suitable for the stereo visual odometry (VO) SLAM, where the measurement $\mathbf{z}_t$ generally contains lots of detected feature points in the pixel coordinates.

Considering this situation, we resort to optimization based approaches for our stereo VO SLAM framework. To this end, we will first go for the bundle adjustment (BA) to jointly optimize for the camera parameters and 3D landmarks with the stereo pairs. In other words, we will use BA to adjust the camera poses such that the detected 2D feature points in the stereo pairs matches the 2D projection of the 3D landmarks. As already mentioned, the projection here is essentially the measurement model as follows:

$$\mathbf{z} = h(\mathbf{T}, \mathbf{l})$$

where $\mathbf{T}$ refers to the pose and $\mathbf{l}$ denotes the set of landmarks in 3D, and $\mathbf{z}$ are set of measurements in the pixel coordinate.

Following the least-squares, BA optimizes the following cost function:

$$\mathcal{E} = \frac{1}{2} \sum_{i=1}^{N} \sum_{k=1}^{M} ||z_{ik} - h(T_i, l_k)||_2^2$$

The above equation iterates over all N poses and M landmarks. Since the function $h(T_i, l_k)$ is nonlinear, we will use nonlinear optimization and exploit sparsity as follows to solve this equation.
To explain our solution approach for the above equation, let $\mathbf{x} = \{T_1, \ldots, T_N, l_1, \ldots, l_M\}$ is the set of our optimization variables stacked together. We will solve by numerically adding incremental updates $\Delta\mathbf{x}$ and minimizing

$$\mathcal{E} = \frac{1}{2} \mid\mid \mathbf{e} + H\Delta\mathbf{x} \mid\mid^2$$

where $\mathbf{e} = \mathbf{z} - h(\mathbf{T}, \mathbf{l})$ is the error vector in measurement. Here, H is the Hessian of the objective with respect to the optimization variables $\mathbf{x}$ as shown above.
Note that in our case of visual SLAM, a single stereo pair contains many feature points, and so the dimensions of H will be large. This will make the inversion of H computationally intensive. Here, we will exploit the sparsity pattern in H to accelerate the SLAM framework.

Exploiting Sparsity: The Hessian H is given by $H(\mathbf{x}) = J(\mathbf{x})^T J(\mathbf{x})$, where $J(\mathbf{x})$ is the Jacobian with respect to $\mathbf{x}$. Here, $J_{ik}(\mathbf{x})$ contains the error derivative with respect to pose i (T_i) and landmark k (l_k). This gives the Jacobian J as well the Hessian H a block diagonal structure:

$$H = \begin{bmatrix} H_{pp} & H_{pl} \\ H_{lp} & H_{ll} \end{bmatrix} = \begin{bmatrix} H_{pp} & H_{pl} \\ H_{pl}^T & H_{ll} \end{bmatrix}$$

Here, H_{pp} and H_{ll} are only associated with poses, and landmarks, respectively. They have the block diagonal structure. The other two off-diagonal sub-matrices may be sparse or dense. To exploit the sparsity, we will use the Schur trick. From the above error equation for BA, let

$$H\Delta\mathbf{x} = \begin{bmatrix} H_{pp} & H_{pl} \\ H_{pl}^T & H_{ll} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_p \\ \Delta\mathbf{x}_l \end{bmatrix} = \begin{bmatrix} u \\ v \end{bmatrix}$$

We will solve for $\Delta\mathbf{x}_p$ first with the Schur trick

$$(H_{pp} - H_{pl}\ H_{ll}^{-1}\ H_{pl}^T)\ \Delta\mathbf{x}_p = u - H_{pl}\ H_{ll}^{-1}\ v$$

Since H_{ll} has the block diagonal structure, computing its inverse is computationally much cheaper. Next, we will solve for $\Delta\mathbf{x}_l = H_{ll}^{-1}\ (v - H_{pl}^T\ \Delta\mathbf{x}_p)$.

Implementation: We will employ g2o [2] to implement the BA method described above. At the time of writing this report, the authors familiarized themselves with the corresponding techniques and the g2o API. A full Bundle Adjustment involves considering all the keyframes and corresponding landmarks. This would take a lot of time to compute especially at the end of a big sequence, and hence we may choose to do only pose optimizations without landmarks. The experimental results will be provided in the final report.

IV. Loop Closure

The frontend possesses short-term information which is compensated by the long-term optimization by the backend. In our stereo VO case, we are only considering the adjacent keyframes, and so, the accumulated drift will be severe. To ameliorate this issue, we will get loop closure based on image similarities. We will use DBoW3 [3] library on top of a fraction of the KITTI dataset [4] to build our visual dictionary first. This library does efficient matching based on k-d tree. Moreover, to enhance the reliability of our loop closure detection, we will make sure that the loop is detected for several consecutive frames since the consecutive frames are in fact quite similar in terms of content for high FPS datasets like KITTI [4].

V. Final Remarks

Overall, the frontend of our stereo VO framework will perform the optical flow based feature matching from the past to the current frame. If the number of tracked features in the current frame is less than a certain threshold compared to the total number of current detections, that will flag the current frame as a keyframe. For each keyframe, we will then extract the feature points in the right camera image, perform triangulation to get the 3D points or the 3D landmarks, and then register this keyframe pose and the landmarks into the map. Moreover, the backend optimization will be triggered each time a keyframe is detected based on bundle adjustment. In additon, we will run the loop closure at a lower frequency than the frontend module since the consecutive frames are quite similar. We will not attempt to control the complexity of the BA solver in this project since the available libraries like g2o [2] do not provide such functionalities and so extending the solvers there with such heuristics will be quite time-consuming.

TABLE I: Project Timeline

Task	Description	Dates
1	Literature Review	10/24 - 11/02
2	Preparing the Data Loader and Pre-processing	11/02 - 11/07
3	Feature Extraction and Mapping	11/05 -11/08
4	Initial Pose Estimation	11/08 - 11/11
5	Bundle Adjustment	11/11 - 11/18
6	Loop Closure Detection	11/18 - 11/28
7	Visualization	(Should be done all along the project)
8	Performance Evaluation compared to other SLAM systems	11/28 - 12/4

VI. Contributions

We have worked on the front-end module and have divided the work accordingly. Collectively, everyone has worked on the keyframe heuristics. In Table 1, we have shown the timeline

that we plan to follow.
1. Sai Sravan Yarlagadda has worked on the data loading from the Kitti dataset, computed the disparity and the depth maps, and contributed to debugging the codes. In the handout, Sravan has focused on working on the aim and motivation behind the project and the timeline of the project
2. Shubhra Aich has worked on the project proposal and has been working on Pangolin to work with Python for visualization. Shubhra has written the backend, Loop closure, and final remarks in the handout.
3. Akshay Badagabettu has worked on Feature Extraction, Mapping, 3D/2D Correspondence, and Minimizing Reprojection errors. In the handout, he has written the Front end.

VII. Future Contributions

In the future, we plan to divide work based on topics. Shubhra will work on Loop Detection and Bag of Words (BOW). Sravan and Akshay will be working on how to get the G2O Wrapper running. Collectively, all three will work on Loop Closure and compare the system's performance to others.

VIII. Changes in scope of the project

There is only one change in terms of the scope of the project as described in the project proposal. We shifted the coding language from C++ to Python as initially we wanted to work with a low-level language to get a more complete understanding of the SLAM system, but most of the authors are not familiar enough with C++ to be able to build the pipeline in the expected duration of the project. Hence, the coding language was shifted to Python.

References

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT Press, 2005.
[2] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "G2o: A general framework for graph optimization," in *ICRA*, 2011.
[3] "DBoW3," https://github.com/rmsalinas/DBow3.
[4] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *CVPR*, 2012.