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

Akshay Badagabettu
Master's in AI Engineering
Carnegie Mellon University
Pittsburgh, USA
abadagab@andrew.cmu.edu

Shubhra Aich
Research Staff - AirLab
Carnegie Mellon University
Pittsburgh, USA
saich@andrew.cmu.edu

Sai Yarlagadda
Master's in AI Engineering
Carnegie Mellon University
Pittsburgh, USA
saisravy@andrew.cmu.edu

***Abstract*—This paper focuses on understanding the engineering intricacies of Simultaneous Localization and Mapping (SLAM) by constructing a fundamental framework using Python from scratch. There are a lot of SLAM implementations, the absence of uniformity often complicates selection for further development. To surmount this, a simplistic yet comprehensive stereo Visual Odometry SLAM system is built and evaluated on the KITTI odometry dataset, specifically sequence 00. A comprehensive description of the theory behind the numerous components of our system such as frontend, backend, and loop closure, has been given. The results of our system are qualitative for now. Limitations have been identified in our system, and a discussion on how to improve it further is done.**

***Index Terms*—Visual Odometry, SLAM, KITTI, frontend, backend, loop closure**

I. Motivation and Introduction

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, particularly on KITTI sequence 00. 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 four main modules in our framework – dataloader, frontend, backend, and loop closure. Our proposed pipeline is shown in Fig. 1.

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.

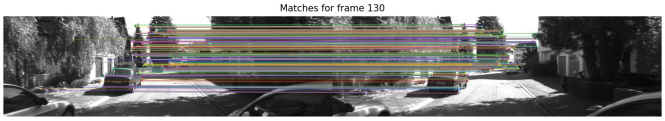

Fig. 2: Feature matches in left images of frame 130 and 131

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. We use Semi Global Block Matching [1] for computing the disparity. Even though it is a bit more compute heavy than Block Matching [2], we get much more accurate results. By drawing out the stereo camera position and using similar triangles, we

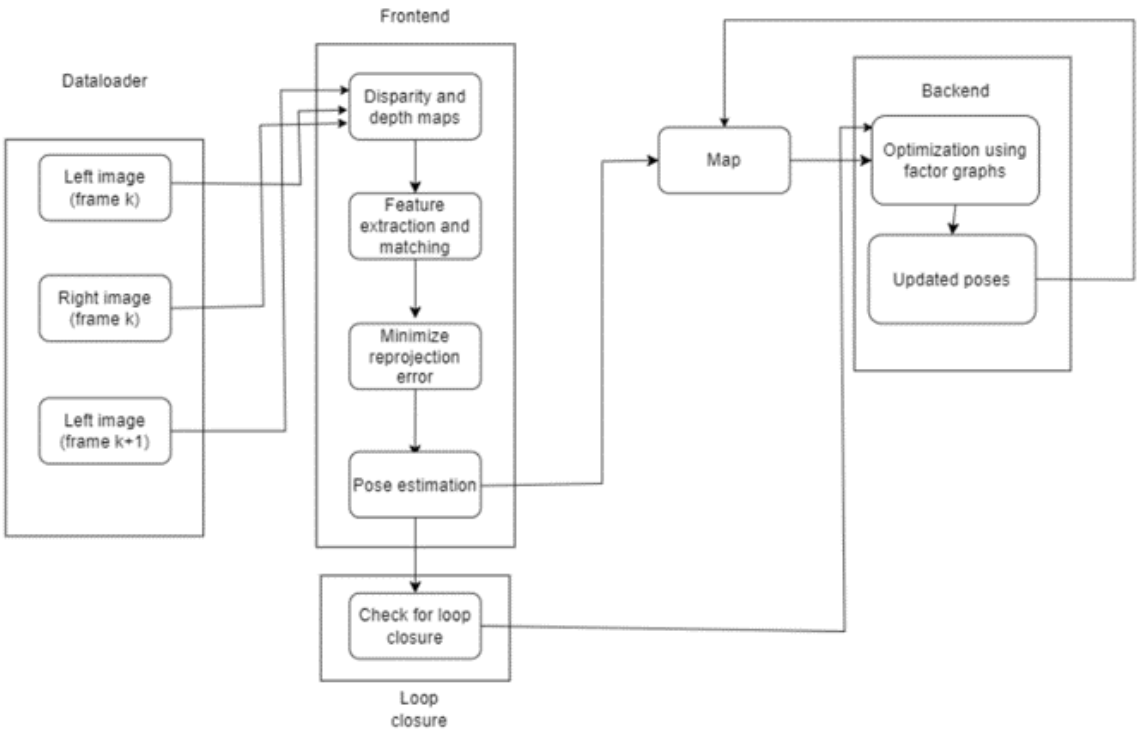

Fig. 1: Proposed pipeline

can formulate depth as a function of focal length, baseline, and disparity.

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

A parameter called max_depth is set to 3000. If we look at the disparity, we see a black strip on the leftmost side. This is because the right camera does not have the information for that part of the image. Hence, we get massive depth values in that region, which is wrong, and so we ignore those values. 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, (X_w, Y_w, Z_w) 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 T_w 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.

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

The final output of the frontend system is shown in Fig. 3. The black line indicates the ground truth poses and the orange line indicates the VO poses. As expected, there is substantial amount of drift and this is because VO takes only adjacent frames into consideration when determining the pose.

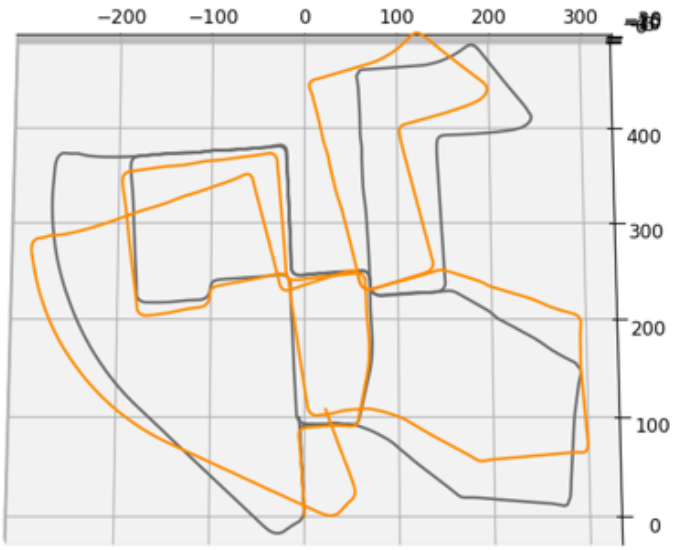

Fig. 3: Visual Odometry output

III. Backend

In this section we will describe the complete theory part of our backend system.

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 [3]:

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

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 [3]:

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

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

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

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 $\Delta\mathbf{x}_l = H_{ll}^{-1} \, (v - H_{pl}^T \, \Delta\mathbf{x}_p)$.

We now use factor graphs to represent this system instead of the big matrices. The measurement jacobian is a tall and skinny matrix that consists of the states (poses and landmarks) in the column and the measurements in the rows. from [4], we see that a factor graph is a bipartite graph with two types of nodes: factors and variables. An edge of a factor graph is always between a factor node and a variable node. In our system, we chose to do pose graph optimization only to save computational time. We can now represent the measurement jacobian as a factor graph, as shown in Fig. 4.

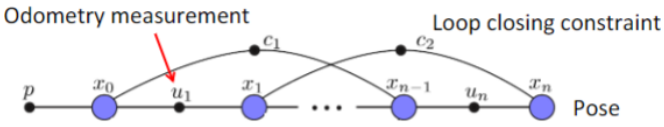

Fig. 4: Pose graph

According to [4], the Maximum a Posteriori inference for SLAM problems with Gaussian noise models is equivalent to solving a nonlinear least squares problem.

$$X^{MAP} = \arg\min_X \sum_i ||h_i(X_i) - z_i||^2_{\sum_i}$$

We can now linearize it at the linearization point and solve it using the numerous nonlinear optimizers available.

Levenberg-Marquardt (LM) optimization: We use the LM algorithm for the pose-graph optimization, which is a variant of the Gauss-Newton method [5]. In short, the normal equation $J^T J\Delta\mathbf{x} = -J^T\epsilon$ is augmented with λI as

$$(J^T J + \lambda I)\Delta\mathbf{x} = -J^T\epsilon$$

where I is the identity matrix, ϵ is the error term, and λ is the hyperparameter initially set to $1e^{-3} \ \texttt{diag}(J^T J)$. If the solution $\Delta\mathbf{x}$ to the augmented equation reduces the error, then we accept the solution and divide λ by 10 in the next iteration. On the other hand, λ is multiplied by 10 if our $\Delta\mathbf{x}$ increases the error during the next iteration.

IV. Loop Closure

The frontend of the conventional SLAM pipeline estimates the relative poses of the ego-vehicle over a short period of time – usually several adjacent frames. On the other hand, the backend, in theory, can access the complete history to run the nonlinear optimization pipeline. Thus, for backend optimization, if we only consider the relative pose estimates provided by the frontend for several adjancent frames, the error in estimation causes the trajectory to drift over time and thus our map estimation lacks global consistency.

Overall, the backend optimization depends on the pose graph structure. Detecting a loop or a visit to the same location after long time adds a strong pose constraint to correct the pose graph corresponding to those time stamps involved in the loop. The crux of this long-term pose correction or loop closure is thus the successful detection of the loop.

In this project, we employ the well-known bag-of-visual-words (BoVW) trick [6] for the visual loop detection. The idea behind BoVW is to create a visual dictionary first where each visual

word (or feature descriptor) in the dictionary ideally represents a somewhat distinctive concept. In the context of object matching in the video stream for example [6], the individual concepts may represent different objects, such as vehicles, pedestrians or humans, etc. The more representative this dictionary is, the better the detection performs in general. We use SIFT [7] as the candidate feature descriptor for loop closure. Below we outline the steps in our loop closure detection procedure.

A. Building the Dictionary

To create this dictionary, we iterate over all the KITTI sequences. However, using all the features from all the images is computationally prohibitive. Note that the consecutive frames in each sequence contains pretty much the same set of features. Combining these observations, for each sequence, we extract the features from the image with a predefined stride (50 or 100 in our case). Another strategy would be to collect a random percentage of the features from each image in the dataset. Empirically we do not find any notable difference with the later, and so, we simply use the former scheme for the initial collection of feature descriptors to generate the visual words. Also, both these options are available in our implementation.

With the above mentioned strategy, we now have a $\mathcal{X} \in \mathbb{R}^{n \times d}$ matrix of features on which we want to form our visual dictionary. Here, n and d denotes the number and dimension of the descriptors, respectively. For the BoVW model, we need to specify the dimension or length of the dictionary k. Based on our initial pilot study with $k \in \{500, 1000, 2000, 3000\}$, we choose $k = 1000$ in our final implementation. Our choice of this hyperparameter is driven by the trade-off between numerical performance and runtime of the loop detection procedure. Next, we approximate our dictionary $\mathcal{D} \in \mathbb{R}^{k \times d}$ using the k-means algorithm [8]. We are giving a brief account of k-means algorithm here for completeness that can be skipped without interrupting the flow of this section.

k-means algorithm: Given the data matrix $\mathcal{X} \in \mathbb{R}^{n \times d}$, the objective is to find the k centroids $\mu_j \quad \forall j \in \{1, \ldots, k\}$ for these data vectors of dimension d. Here, we are considering hard assignment of the data points to the clusters which means $\forall i \in \{1, \ldots, n\} \quad \mathbf{x}_i$ belongs to only one cluster in the list $\{1, \ldots, k\}$. Denoting the set of indicator variables $\mathbb{I}_{ij} \in \{0, 1\}$, we can then write the k-means objective function as follows:

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{j=1}^{k} \mathbb{I}_{ij} ||\mathbf{x_i} - \mu_j||^2$$

where $\{\mathbb{I}_{ij}\}$ and $\{\mu_j\}$ are the variables involved in the minimization of $\mathcal{L}$. The minimization is performed in an iterative process that alternates between $\mathbb{I}_{ij}$ and μ_j to minimize $\mathcal{L}$ starting from some initialization of $\{\mu_j\}$. Note that the objective function is linear in terms of $\mathbb{I}_{ij}$, solving it eventually makes $\mathbb{I}_{ij} = 1$ only if assigning $\mathbf{x}_i$ to cluster j leads to the minimum value for $||\mathbf{x_i} - \mu_j||^2$. Next, the objective function with respect to $\{\mu_j\}$ is quadratic for which a global optimum exists as follows [8] :

$$\mu_j = \frac{\sum_{i=1}^{n} \mathbb{I}_{ij} \mathbf{x}_i}{\sum_{i=1}^{n} \mathbb{I}_{ij}}$$

The iterations are performed until convergence, i.e. the cluster assignments are unchanged. Despite the existence of the global optimum in the minimization process, the solution to the k-means algorithm is random due to the initial cluster assignment. In some cases, the solution may be degenerate if the cluster initializations are not well-distributed. To overcome this limitation, we employ the variant with better initialization based on farthest points [9].

B. Loop Detection

Checking for the loop or visual similarity detection is fairly straightforward once the dictionary is stored. During inference, we extract the feature descriptors from each left-camera image and get the normalized histogram via cluster assignment based on the prebuilt dictionary. The number of bins in this histogram is the same as the length k of our vocabulary ($k = 1000$ as mentioned above). Converting the arbitrary number of detected features into this fixed-length representation is another notable advantage of the BoVW trick. We append this fixed-length histogram feature for each frame into a list. We employ the histogram correlation metric to compute the similarity score between histogram h_1 and h_2 as follows:

$$sim(h_1, h_2) = 1 - \frac{\sum_{b=1}^{k} (h_1(b) - \overline{h_1})(h_2(b) - \overline{h_2})}{\sqrt{\sum_{b=1}^{k} (h_1(b) - \overline{h_1})^2 \sum_{b=1}^{k} (h_2(b) - \overline{h_2})^2}}$$

where $\overline{h_l} = \frac{1}{k} \sum_{b=1}^{k} h_l(b)$.

Additional heuristics: We have had to employ some additional heuristics to make the loop detection work in practice. First, the consecutive frames in the sequence have pretty high similarity. To avoid matching with nearby frames, we only compare against the frames with a certain offset (such as 100 frames before). Moreover, to accelerate the detection process, we compare the frames with a predefined stride (e.g. 3 in our case) rather than comparing each consecutive frame in the list with the target frame. In addition, we empirically set a threshold for the similarity score to account for successful loop detection. As for the second check, we compare the number of matched features in the frame pairs identified by BoVW as a potential closure. If that number is greater than a predefined threshold (200 in our implementation), we finally recognize that pair as a true positive.

V. Implementation

The frontend implementation has been written in detail in the Frontend section. As part of the frontend, we solve the least squares problem to find the pose and then we invert it and transform the pose to the first camera coordinate frame which is also the world frame. The loop detection is currently run at

the same frequency as the frontend. This means that for every frame, we are checking whether there is a loop closure detected. Only after the frame passes both the loop closure checks i.e, the BoVW check and the number of features check, the two frames are listed as loop closure frames and the backend gets triggered.

The backend implementation is done with the help of a library called Georgia Tech Smoothing and Mapping (GTSAM) [10]. We are doing pose graph optimization only instead of the conventional pose and landmark optimization. The reason behind this is that there are much more landmarks than poses. Including them during the optimization process makes it very compute heavy and the run time is slowed down significantly.

Take for example a loop closure is detected at frame k. Then, all the poses and loop closure frames until frame k are passed to the backend. We then loop through all the frames and add odometry factors between each pose. A certain amount of noise was added to each measurement to account for the measurement uncertainty. The backend was very sensitive to the amount of noise added and more details will be provided in further sections. The pose given by the visual odometry was made to be the initial estimate. If we had set the initial estimate to say the origin i.e, (0, 0, 0), there is a very high chance that the optimizer will get stuck in a local minimum.

After adding the initial estimates for all the frames and the odometry factors between them, we add the factors between the loop closure frames too. After the nonlinear factor graph and the initial estimates are done, we use the inbuilt Levenberg-Marquardt [11] optimizer to obtain the optimized pose. Lastly, the final pose is where the VO starts from again. This means that after a loop closure is detected, the backend is triggered and the poses are corrected. From now until the next loop closure, VO takes over again, but the starting point for VO is the corrected pose. The advantage of running backend at every loop closure instead of doing a full adjustment only at the end is that drift keeps getting corrected regularly. Running the optimization only at the end of the sequence can result in poor results sometimes due to irrecoverable drifts.

VI. Results and Discussions

As mentioned in the above sections, our SLAM system has been evaluated on KITTI sequence 00. Before we move to the results of the full system, it should be noted that the noise added to the odometry factors played a huge role. Adding a very small amount of noise i.e., small sigmas for the (x, y, z, roll, pitch, yaw), does not allow the backend system to find the optimal solution. For this reason, the updated poses are extremely similar to the VO poses, as seen in the Fig. 5, where the backend system was triggered every 15 frames. On the flip side, if we add too much noise to the system, the system completely diverges, and the optimizer cannot find a good solution, as shown in Fig. 6, where the optimizer was run after every 15 frames. A relatively good noise value was found after

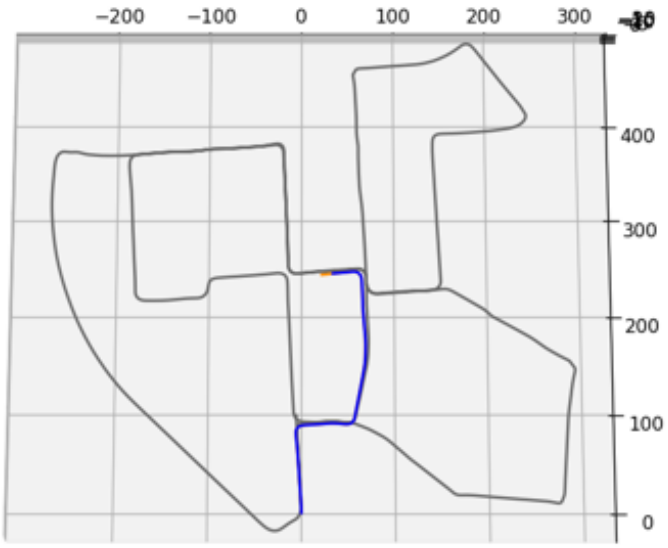

Fig. 5: Optimization with very small odometry noise

Fig. 6: Optimization with very large odometry noise

running the system for multiple iterations. It should be noted that most of the time, in real-life scenarios, the noise values are determined randomly and then adjusted.

Now that we have attained reasonably good parameter values for the numerous variables in our SLAM system, such as noise, stride, and two thresholds (one for loop closure, another for frontend feature matching), a single adjustment at the end of the sequence was done. This means that only at the last frame of the sequence the backend is triggered and optimization takes place. The results are shown in Fig. 7.

Fig. 7: Optimization with very small odometry noise
The pink line indicates the optimized poses, and as usual, the orange and black lines represent the VO poses and ground truth, respectively. We can clearly see that the updated poses match the ground truth much closer than VO.

Now, we run the system as intended i.e., the backend gets triggered only at loop closure detection, and the drifts keep getting corrected. Two iterations were run, one without any measurement noise and the other with the optimal value of noise that was found. Fig. 8 and Fig. 9 show the results obtained. The

Fig. 8: No Noise

Fig. 9: Optimal noise

blue line represents the updated poses. The orange VO poses are different from before because after the backend optimization takes place, the poses are corrected, and VO again starts from the corrected pose. Visually, we can see that the SLAM system with the optimal measurement noise performs the best.

Initially, we had planned to use iSAM2 [12] as our optimizer, but surprisingly, for the very first backend optimization, i.e., at the very first loop closure, it took more than 20 minutes to give a solution. The optimization was then run every 15 frames to check if the implementation was correct. iSAM2 gave the best results in that short run, but it was much slower than Levenberg-Marquardt or Powell's Dogleg optimizer [13]. It was also quite surprising to see that the runtime of the LM optimizer was extremely quick, even at the end of the frame, with all the VO poses and loop closure constraints (less than a couple of seconds). The entire system as a whole cannot run in real time which is 10fps in the case of KITTI sequence 00. Most of the computation time is taken by the frontend system, and because we are using SIFT and Python compiler, we cannot achieve real-time operation.

VII. Future Work

Currently, our SLAM system is not as good as the state-of-the-art systems such as ORB-SLAM [14], or VLOAM [15]. The future work to be done to improve upon this SLAM system is:

- As of now, the frequency of the loop closure detection is the same as that of the frontend. We can run it at a lower frequency to save some computation time. Also, we can create a heuristic wherein if a loop closure is detected, we do not check for it again for the next 100 frames. This number is again a hyperparameter and should be adjusted according to the use case.
- In our system, the frontend, loop closure detection, and backend are not running in parallel. Parallelizing and running all three processes on GPUs will make the system much faster.
- A deep learning based approach can be considered when designing the frontend system, improving accuracy and computational time.
- Even though the loop closure works perfectly in our experiments, we can add an additional heuristic that checks whether the poses are close enough in 3D. This will make the system more robust.

VIII. Conclusions

This study presents a detailed exploration of the frontend and backend implementations within the context of the SLAM framework. The front end solves the least squares problem and is coupled with loop closure checks to trigger backend optimization. Our SLAM system focuses on optimizing poses rather than both poses and landmarks, making it computationally efficient. The initial estimates derived from the visual odometry were taken as starting points for backend optimization. Following the construction of the nonlinear factor graph and initial estimates, the Levenberg-Marquardt optimizer is employed to refine the poses. The optimized pose marks the new starting point for the subsequent visual odometry process post-loop closure, ensuring continual drift correction. In summary, the objective of this paper was to construct a VO SLAM system from the ground up and is aimed to provide comprehensive insights into intricacies of building a fundamental SLAM pipeline.

IX. Github Link

Please click here to access the Github link to view the code. As mentioned in Piazza, we are not uploading anything on the gradescope code submission link.

REFERENCES

[1] H. Hirschmuller, "Accurate and efficient stereo processing by semi-global matching and mutual information," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 2. IEEE, 2005, pp. 807–814.

[2] A. Barjatya, "Block matching algorithms for motion estimation," *IEEE Transactions Evolution Computation*, vol. 8, no. 3, pp. 225–239, 2004.

[3] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT Press, 2005.

[4] F. Dellaert, M. Kaess, *et al.*, "Factor graphs for robot perception," *Foundations and Trends® in Robotics*, vol. 6, no. 1-2, pp. 1–139, 2017.

[5] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge University Press, 2003.

[6] Sivic and Zisserman, "Video google: a text retrieval approach to object matching in videos," in *ICCV*, 2003.

[7] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *IJCV*, 2004.

[8] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006.

[9] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *ACM-SIAM SODA*, 2007.

[10] F. Dellaert and G. Contributors, "borglab/gtsam," May 2022. [Online]. Available: https://github.com/borglab/gtsam)

[11] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *Quarterly of applied mathematics*, vol. 2, no. 2, pp. 164–168, 1944.

[12] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, "isam2: Incremental smoothing and mapping using the bayes tree," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 216–235, 2012.

[13] M. J. Powell, "A hybrid method for nonlinear equations," *Numerical methods for nonlinear algebraic equations*, pp. 87–161, 1970.

[14] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, "Orb-slam: a versatile and accurate monocular slam system," *IEEE transactions on robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[15] W. Shao, S. Vijayarangan, C. Li, and G. Kantor, "Stereo visual inertial lidar simultaneous localization and mapping," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 370–377.