Distributed SLAM - Using Multiple Cameras for Localization and Mapping

Justin Yue
UC-Irvine CS217

Abstract

In this project, I cover my process on how to perform distributed slam, where multiple cameras can be used to create the map of an environment. After listing the tools that I use to solve this problem and my setup environment, my final report covers the step by step pipeline that I use, which includes calibration, visual odometry, the backend optimization, and merging the individual maps into a final map. For each step, I demonstrate they have an impact on the final result and where applicable, go over the theory. After discussing the final result, I will also talk about future works and experiments to improve on my current progress.

1 Introduction

Problem The goal of this project is to run a SLAM (simultaneous localization and mapping) algorithm on multiple entities such as robots. Traditionally, SLAM is runned on only 1 robot with a variety of sensors, and as it traverses its environment, it is able to create a map of the environment while determining its position within the environment. Within the past decade, there has been more interest in researching how to distribute the SLAM problem across multiple robots. Most researchers focus on the localization problem, which is coordinating the team of robots accurately enough to determine each of their positions on the map. Distributed SLAM can also be generalized to a wide variety of agents besides robots. These agents can be self-driving vehicles on the ground, aerial drones, a group of users of augmented reality (AR)/ virtual reality (VR) headsets, and more.

Software I primarily used the C++ programming language. This is because of GTSAM, a library that implements factor graphs for optimization and has better documentation in C++. As a result, I also used other C++ libraries such as OpenCV (an open sourced computer vision library), Eigen (a library for linear algebra computations), and Pangolin (a tool that I use for visualizing a sparse map of camera poses).

Project Setup For my project setup, I created a grid board of Aruco markers. Aruco markers are synethetic square markers that are useful for pose estimation because for each marker, a camera can recover its rotation and translation information, which I will discuss more in-depth during the visual odometry portion of this report. But they are useful since they contain an inner binary matrix that determines each Aruco marker's identifier (ID) while their wide black borders help facilitate fast detection of the markers, identification of ID, and error correction as needed. The inner binary matrix and IDs information are captured in a dictionary, and in this project, I use *6x6_250*, meaning each inner binary matrix consists of 6 by 6 bits and can have an ID from 0 to 249. Multiple IDs is helpful since I can assign a position to be the marker's place in the world coordinate plane that a camera can move through.

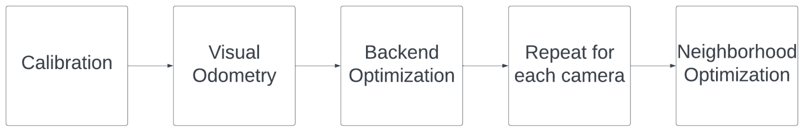

Figure 1: Distributed SLAM Pipeline

Type	Final Reprojection Error
Aruco Grid Board	55.7823
Checker Board	0.07567
Charuco Board	0.2284

Figure 2: Reprojection error for each board after calibration

2 Pipeline

2.1 Calibration

Distortion In order for my visual odometry to work, calibration was needed to find the intrinsics camera matrix and distortion coefficients. To briefly summarize, the widely accepted pinhole camera model can suffer from significant distortions. Radial distortion causes the image to appear curved where objects near center of the image appear closer while objects at the edges will appear farther away. Below, radial distortion can be represented and come with parameters k_1, k_2, & k_3.

$$x_{\text{distorted}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{\text{distorted}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Another type, tangential distortion, causes some areas to look even closer than other areas due to the lenses not being perfectly parallel to the image sensor. Tangential distortion can be formulated with parameters p_1 and p_2.

$$x_{\text{distorted}} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$

$$y_{\text{distorted}} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

Then, the distortion coefficients found by calibration would be $(k_1 k_2 p_1 p_2 k_3)$. But after correcting distortion coefficients, any further visual odometry processing would require the intrinsics matrix, which captures the focal length f and principal point c measurements.

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Reprojection Error To ensure the accuracy of my pose estimation during visual odometry, I reprojected the points from Aruco boards and checker boards into my camera. Initially, I believed Aruco boards would give me better results since each marker on an Aruco board could provide better correspondences compared to a checkerboard's squares. However, checker board calibration provided better reprojection error but not well enough since ideally, the error should be as close as possible to 0. With Charuco boards, I combined the previous two approaches by detecting the markers and checker board pattern, and the Charuco approach provided me a much better projection error.

2.2 Visual Odometry

Visual odometry is where I estimate the camera pose, rotation R and translation t, as I move a camera in the environment. As mentioned earlier in the introduction, Aruco markers are useful in estimating a camera's pose since its 4 corners provide enough correspondences, namely its 4 corners. Therefore, we know p, the 2D image points of the Aruco markers as seen by the camera, and P, the 3D points of the Aruco markers. Recall the following formula for projecting P to p:

$$p = CP \quad (1)$$

$$= KRtP \quad (2)$$

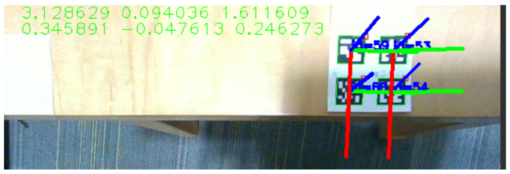

Figure 3: An example of a detected aruco marker; The pose, which is the transformation from the marker to the camera, is drawn.

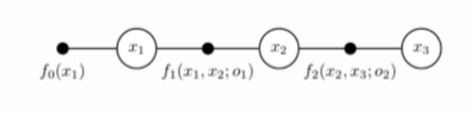

Figure 4: A simple example of a factor graph to measure agent motion.

Because the camera is calibrated, the only unknowns are R and t, so this problem can be solved using PnP (Perspective-n-Point) for each aruco marker i. One thing to note is that in R in OpenCV's implementation is returned as a vector but can easily be converted to a rotation matrix using the Rodrigues formula. Additionally, t by itself does not give my world coordinate since t is the translation from marker i to my camera, so the marker assumes its coordinate is at the origin. To correctly solve for my pose, I assigned each marker their true world coordinate, m_{world}, and used a process similar to triangulation.

$$t_{\text{world}} = R^{-1} * (m_{\text{world}} - t)$$

While reading OpenCV's documentation on Aruco markers, I found that detecting more markers provides a more accurate pose estimation, so I set up my environment such that my camera would view a group of markers at a time. I chose to take the average of t_1, t_2, t_3, $...t_n$ since averaging the translation from a group of markers should provide a better consensus of where the camera is. I also attempted to do the same with the rotation by averaging the $x, y, \& z$ euler angles that I found using the $R_1, R_2, R_3..R_n$ vectors, but this approach resulted in the calculated world coordinate to be very far away from the Aruco markers. As a result, I only used the rotation of the first marker that I detected, and my final formula for calculating the world pose is the following:

$$t_{\text{world}} = R_0^{-1} * (m_{\text{world}} - t_{\text{average}})$$

2.3 Pose Graphs for Backend Optimization

Factor Graphs and iSAM My project focuses mainly on creating a sparse map of poses, and even though Aruco markers are good for pose estimation, the calculated camera poses might still be noisy. Consequently, backend optimization is needed to make my poses more accurate, and I choose to use factor graphs to optimize my poses as a pose graph[2]. First, factor graphs are defined as bipartite non-directed graphs where each node is either a variable or factor. The variables $x_1, x_2, x_3...x_n$ are the unknown poses that are refined through optimization while the factors $f_1, f_2, f_3...f_n$ are geometric constraints on the variables For example, let ϵ be Gaussian noise. If x_1 is estimated to be at $(0, 0) + \epsilon$ and x_2 is at $(2, 0) + \epsilon$, then the factor that connects them is about $(2, 0)$; the factor z_1 reflects the change in position. Ideally, optimization will help the estimated x_1 and x_2 converge at the origin and $(2, 0)$ respectively. Additionally, a prior factor is provided for x_1, and for all the following factors, a noise model o_i of the odometry measurements is included as well. As typical of SLAM literature, the noise is assumed to be Gaussian.

Optimizing the factor graph can be formulated as maximizing the factor graph's MAP (maximum-a-posteriori) inference.

$$f(x_1, x_2, x_3...x_n) = \Pi f_i(X_i) \tag{3}$$
$$MAP = \arg\min -logP(x_1...x_n|z_1...z_n) \tag{4}$$

To accomplish optimization, I use iSAM (incremental smoothing and mapping), another nonlinear optimization technique proposed by Dellaert et. al[3]. For brevity, I simply state that the above MAP

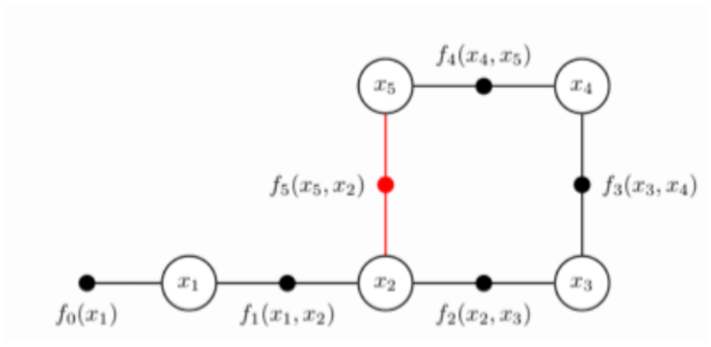

Figure 5: The red edge is where loop closure takes place since the agent is revisiting variable x_2.

estimate can be optimized by converting the SLAM problem to a least squares problem, $||A\theta - b||^2$ where θ is the vector of poses we wish to find and can be solved with QR factorization. This is important since iSAM avoids unnecessary calculations by directly updating a square root factor when a new odometry measurement is added, and this is done with an incremental QR matrix factorization.

Loop Closing Loop closure also plays a key role during backend optimization because the optimization of the pose graph can still be affected by sensor drift, where the error in pose estimation begins to grow over time. To mitigate sensor drift, my backend optimization checks if the camera is viewing something it has already seen before. Since Aruco markers are used for visual odometry, this part is trivial; the program only needs to check if it is viewing an Aruco ID that it has seen t amount of time before. Once the camera determines a revisited marker, a factor z_n is inserted between the nth pose and the closest pose that represents the revisited marker. Having this loop closure events adds another geometric constraint that the optimization process must respect, thus solving loop closure.

2.4 Nearest Neighbor Optimization

After confirming that SLAM for a single agent would work, I then considered 2 approaches to accomplish a distributed SLAM. My first naive approach was that each new odometry measurement would be sent to the other agents. Each robot would then optimize and maintain a sparse map of poses that not only includes its trajectory but also that of the other robots. The downside of this approach lies in the redundancy of how each agent is performing the same task, which adds to the computational task. Dellaert et. al came to the same conclusion but found there are still advantages to this type of approach[1]. Among them are a) the idea of maintaining a sparse graph of poses, b) having measurements aggregated into a single source, and c) robustness to loss of data in case one of the agent fails. As such, Dellaert et. al proposed DDF-SAM (Decentralized Data Fusion Smoothing and Mapping). This approach consisted of 3 parts: a standard optimization module that executes iSAM for an agent's local map and compressing said local map, a communication model to cache and distribute each agent's local map to each other, and a neighborhood optimizer to merge the condensed graph.

Due to time constraints, I modified the Dellaert et. al's final approach to be only the first module of standard iSAM, which is discussed already in the previous subsection, and a neighborhood optimizer. For the neighborhood optimizer, I performed a similar procedure to loop closure where detected Aruco markers are compared to each other but between different local maps instead of the same local map. Thus, this adds an additional constraint for optimization in the final map.

3 Results & Discussion

Note that for this project, I do not have a ground truth to compare my results to. Consequently, this discussion leans more into qualitative observations of the results but still has some quantitative results.

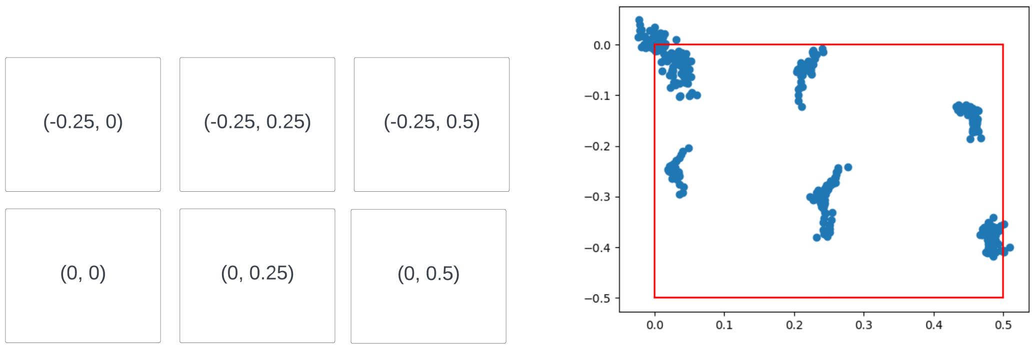

Figure 6: A top-down view of the Aruco map. Left illustrates the assigned world coordinate of each marker; Right illustrates a scatter of xy-poses from each aruco marker. The red lines show where the marker centers are.

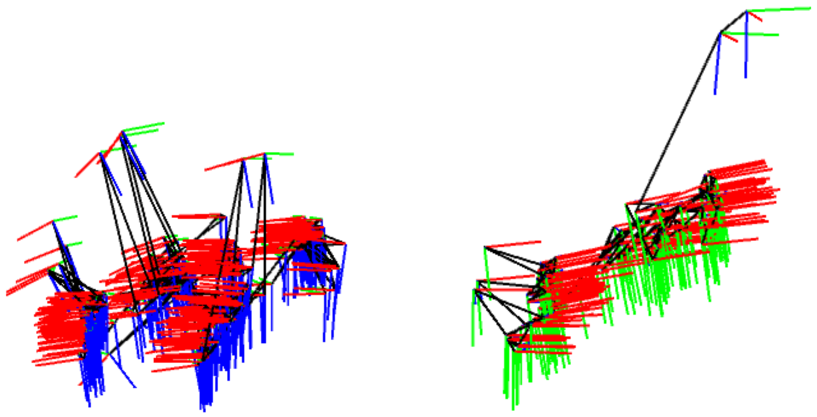

Figure 7: Left: Map of horizontal scan, unoptimized, Right: Map of horizontal scan, optimized.

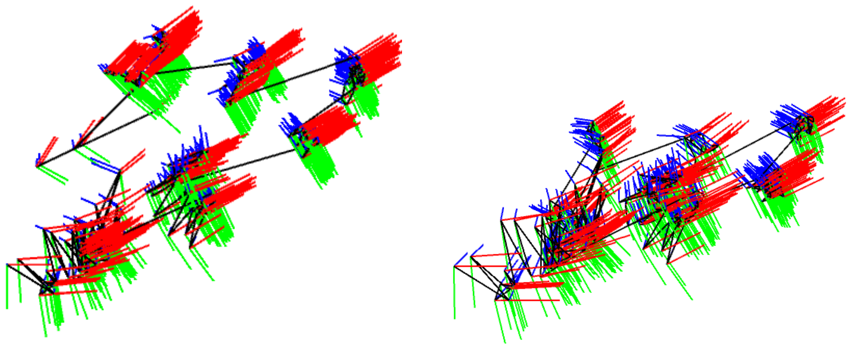

Figure 8: Left: Map of rectangular scan, no loop closure, Right: Map of rectangular scan, loop closure.

Visual Odometry Testing Figure 6 illustrates how I create the environment for this project, and to ensure SLAM would work, I tested the visual odometry to see if reliable poses were calculated.

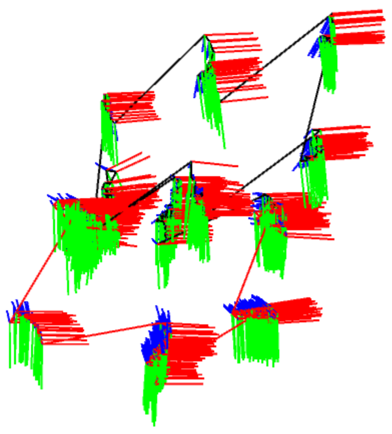

Figure 9: The merged map between 2 local maps. The black edges denote the map from the first camera while the red edges denote the map from the second camera.

In the right plot of Figure 6, I did not always hold the camera directly above the Aruco markers, so the poses do look certainly reasonable from visual observation, albeit somewhat noisy. More specifically, I found the x-coordinate standard deviation σ to be in the range of $(0.0080, 0.0370)$, and the y-coordinate σ is $(0.0240, 0.0370)$. The greater variation in the y-coordinate σ suggests that finding the correct y-coordinate is slightly harder. In the end though, this noise was not enough to have a detrimental impact on the distributed SLAM pipeline as demonstrated in the next few figure.

Single SLAM Testing I then tested SLAM on a single agent by panning a camera over the markers in a horizontal fashion. From the initial camera poses, noise can also be observed in the poses that are higher than the other poses. These outliers did affect the final map after iSAM optimization, which is why outlier rejection was implemented. For a minimum and maximum thresholds τ_1 & τ_2 that I determine ad-hoc, I reject any camera pose not within the specified range as any pose outside will not add significant or useful information to the pose map. The final map saw the removal of about 80% of the outliers in the initial map, resulting in a mostly smoother map that better illustrates the horizontal scan that I made. On the other hand, the pose map is not as sparse as I would hope since many poses are still very close together, and this characteristic of my calculated pose maps persist in the next few figures too.

I then moved onto scanning my markers in a rectangular fashion, moving from $(0, 0) \rightarrow (0, 0.25) \rightarrow (0, 0.5) \rightarrow (-0.25, 0.5) \rightarrow (-0.25, 0.25) \rightarrow (-0.25, 0) \rightarrow (0, 0)$. As seen in Figure 8, we can observe the effect of sensor drift on the left where the final camera compose does not converge back to $(0, 0)$. By adding the constraint that the final poses of the map should be the origin, the map becomes much tighter and is closer to the real world trajectory.

Distributed SLAM Testing For the final results of distributed SLAM, I reused my Aruco marker map and translated its y-coordinates down by -0.25. As such, the second map should appear lower than the first, and 3 markers should overlap between them since they share the same ID. Figure 9 illustrates the final result, and interestingly, only 2 out of the 3 expected markers overlap together. Note how there exists no constraint between the 2 unmerged markers of the black-edges map and red-edges map. This suggests an issue with how I close the loop and optimize between the 2 maps, perhaps related to detecting revisited Aruco markers.

Upon closer inspection of Figure 10, I notice that the batch of poses at the coordinate $(0.275, 0)$ is ever so slightly closer to the poses at $(0.5, 0)$. At the same time, the batch of blue poses should be closer to $(0.5, -0.25)$ but seems to be shifted down, making it farther away from the green batch of poses that it should be looped back to. In my implementation, I close the loop with the closest aruco marker that is revisited, and it is likely that my code is enforcing the wrong geometric constraint. To fix this bug, I would have to further tune my parameters to account for these difference in distances.

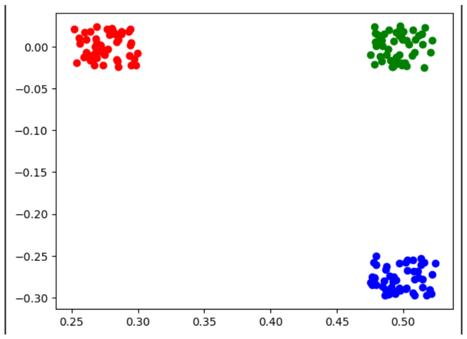

Figure 10: A top-down view of where the final map does not merge well. Red's center is at about (0.275, 0), green's center is at (0.5, 0), and blue's center is at about (0.5, -0.275).

4 Conclusion

In conclusion, I completed a successful pipeline that 1) calibrates a camera, 2) performs visual odometry, 3) successfully optimizes each pose graph, and 4) merges them together. The backend optimization approach, iSAM, works quickly by incrementally updating the pose map with each new odometry measurement. It also has the advantages of Dellaert et al's original approach by maintaining a sparse map, maintaining a single map, and being robust to failure in case one of the agent fails. With some further improvements that are discussed in the Future works section, this distributed SLAM approach could be deployed onto real-world agents.

5 Future Works

In terms of small changes, I would improve on my calibration process to decrease the reprojection error; this would reduce some of the noise I find in my z-axis measurements. Switching to traditional visual odometry from Visual SLAM would help in producing more detailed maps of the camera's pose too. However, if I were to continue with Aruco markers, I would leverage OpenCV's built-in functionality to estimate pose from a grid of markers, which addresses my issue with averaging R for a better estimate. I also would like to leverage simulation tools such as Gazebo from the ROS (Robotics Operating System) suite of tools; using Gazebo would give me a ground truth that better illustrates the accuracy of my SLAM implementation. I would've also liked to show the covariance data to analyze the uncertainty of my poses, but unfortunately, I had difficulties retrieving the data from GTSAM. Most importantly, I would spend more time reading the literature on DDF-SAM and implement the neighborhood optimization using constrained factor graphs.

References

[1] Frank Dellaert Alexander Cunningham, Manohar Paluri. Ddf-sam: Fully distributed slam using constrained factor graphs. 2010.

[2] Hans-Andrea Loeliger Frank Kschischang, Brendan Frey. Factor graphs and the sum-product algorithm. In *In proceedings with IEEE Transactions on Information Theory, Vol. 47*, 2001.

[3] Frank Dallaert Michael Kaess, Ananth Ranganathan. isam: Incremental smoothing and mapping. In *In proceedings with IEEE Transactions on Robotics*, 2008.

6 Appendix

The *main.cpp* file serves as the entrypoint to my distributed SLAM program. It takes command line arguments using the Boost module and either records or runs SLAM on 1 or more pre-recorded videos. If more than 1 is provided, then the program runs distributed SLAM. The remaining files are split into the following 3 folders. For brevity, I refer to only the *cpp* files.

1. backend
    (a) *poseGraph.cpp* provides the code to create the factor graph by adding poses. When adding a pose, it also checks if any markers are revisited, and if so, loop closure takes place. For each pose added, the graph is optimized using iSAM. This file also has a way to retrieve the most optimized poses. Much of this code is inspired by the documentation and examples on Github.
2. frontend
    (a) *arucoFrontEnd.cpp* Much of this code is inspired by the OpenCV documentation. Using Aruco markers, the code can calibrate my camera by retrieving the intrinsics matrix and distortion coefficients. Once calibrated, camera pose can be returned.
    (b) *featureMatcher.cpp* This is not used in my final implementation since I use Aruco markers for visual odometry, but if I did use this, I would experiment with the ORB or SIFT features from the images. Since the camera is calibrated, I can extract the essential matrix to get the camera pose.
3. utils
    (a) *logger.cpp* This file only logs messages to the stdout or stderr, depending on the log status. While I wrote this, I did not use it in my final implementation.
    (b) *calibrateWithCharuco.cpp* This file uses the *arucoFrontEnd.cpp* file to take a picture when I press a key on my computer, then performs calibration after I finish taking pictures. This code is also inspired by the OpenCV documentation.
    (c) *generateAruco.cpp* The code here contains functions to generate Aruco markers, Aruco grids, and Charuco boards. For some functions, I can specify which specific IDs to assign. Like the other OpenCV-related files, this is also inspired by the OpenCV documentation.
    (d) *pangolinDisplay.cpp* This runs a visualization tool called Pangolin to visualize a camera's pose. I referred to another repository that implements SLAM and Pangolin visualization as an example.
    (e) *realsense.cpp* Because my camera is a Realsense camera, the most convenient way to access the camera feed is through the Realsense SDK, which also provides documentation. In this file, I warm up the camera by double checking 30 frames are loading before sending the frames to be processed by OpenCV.
    (f) *get_trajectory.py* This serves as a quick and easy way to process the output poses from my SLAM program, so I can visualize the results for my final report.