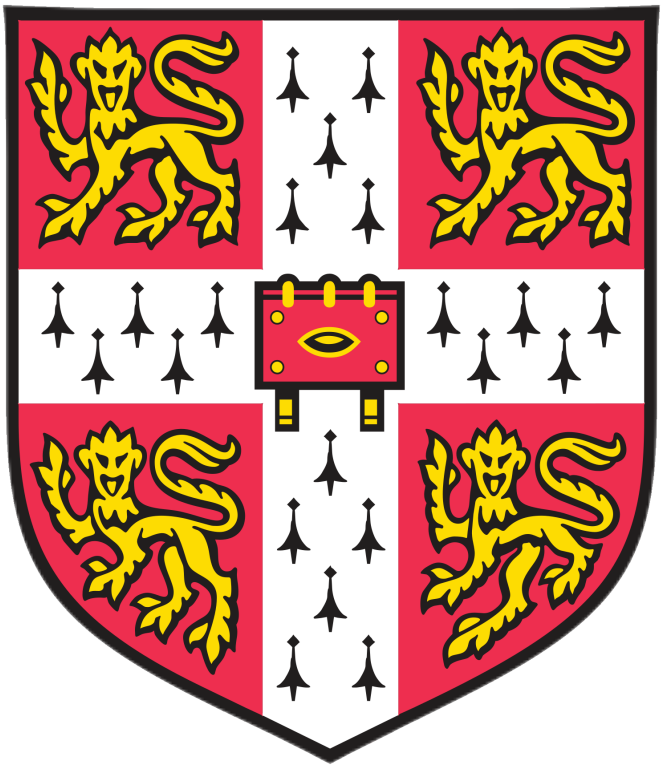
Joshua Bird

Distributed Visual Simultaneous Localization and Mapping



Computer Science Tripos – Part II
Queens' College

April 27, 2024

Declaration

I, Joshua Bird of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: Joshua Bird
Date: April 27, 2024

Proforma

Candidate number:	⟨CANDIDATE NUMBER⟩
Project Title:	**Distributed Visual Simultaneous Localization and Mapping**
Examination:	**Computer Science Tripos – Part II, 2024**
Word Count:	⟨WORD COUNT⟩[1]
Code Line Count:	⟨LINE COUNT⟩[2]
Project Originator:	Joshua Bird, Jan Blumenkamp
Project Supervisor:	Jan Blumenkamp

Original Aims of the Project

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Fusce ac turpis quis ligula lacinia aliquet. Mauris ipsum. Nulla metus metus, ullamcorper vel, tincidunt sed, euismod in, nibh.

Work Completed

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Fusce ac turpis quis ligula lacinia aliquet. Mauris ipsum. Nulla metus metus, ullamcorper vel, tincidunt sed, euismod in, nibh.

Special Difficulties

None.

[1] This word count was computed using `texcount`.

[2] This code line count was computed using `cloc` (excluding autogenerated test output).

Contents

Chapter 1

Introduction

Visual Simultaneous Localization and Mapping (visual SLAM) serves as the foundation of countless modern technologies, with self-driving cars, augmented reality devices, and autonomous drones just being a few examples. By using purely visual inputs, visual SLAM is able to create a 3D map of the surroundings while also localizing the camera's position within this map in real-time.

Unlike other SLAM systems which may use an expensive and heavy sensor such as LIDAR, RGB-Depth cameras, or Radar, visual SLAM only requires the ubiquitous camera sensor, allowing the technology to be used in many commercial applications such as Google's ARCore[1], Boston Dynamic's GraphNav system used on their robot Spot[2], and several models of DJI quadcopters[3] making this a particularly practical field of research.

1.1 Motivation

Multi-robot systems are becoming increasingly common as automation continues to grow in a variety of fields, such as self-driving cars, drone swarms, and warehouse robots. These systems require the agents to understand the world around them as well as their peer's locations within that world. This task is often achieved with technologies such as GPS or motion capture setups, however, not all environments have access to these systems. A few emerging examples include:

- Search and rescue operations in large indoor systems, assisted by drone swarms.
- Self-driving cars in underground road networks.
- Multi-agent cave/subsea exploration.

These are scenarios where multi-agent SLAM provides a compelling solution, as it enables us to build a map of an unknown environment and keep agents aware of their relative poses. However, the majority of existing multi-agent visual SLAM implementations are centralized systems, requiring the agents to maintain reliable communications with a central server in order to operate. This is extremely detrimental, as environments that do not have access to GPS or motion capture systems are also likely to have very poor communication channels – greatly limiting the use cases of these centralized multi-agent SLAM systems.

Naturally, this leaves us with distributed multi-agent visual SLAM systems that do not rely on a centralized management server, allowing the agents to be used in environments where network infrastructure may be lacking. Instead of a central node, the agents are able to communicate peer-to-peer when they come within close proximity to one another.

[1] `https://developers.google.com/ar/develop/fundamentals`
[2] `https://support.bostondynamics.com/s/article/GraphNav-Technical-Summary`
[3] DOI: 10.1016/j.vrih.2019.09.002

It is easy to see the broad-reaching use cases of such a system. Agents will be able to explore sections of the world independently or in small teams, sharing new world locations with their peers as they come into communication range using an ad-hoc network. Agents will be able to accurately determine their peer's locations when they are within communication range, allowing for collision avoidance and cooperative path planning.

1.2 Project Overview

In this project, I:

1. **Design and implement the first distributed monocular visual SLAM system available.** how do I ensure it really is the first? Ive looked at as many papers as I could find It is capable of localization, relative pose estimation, and collaborative mapping, all while being tolerant to degraded network conditions and not reliant on any single leader agent.

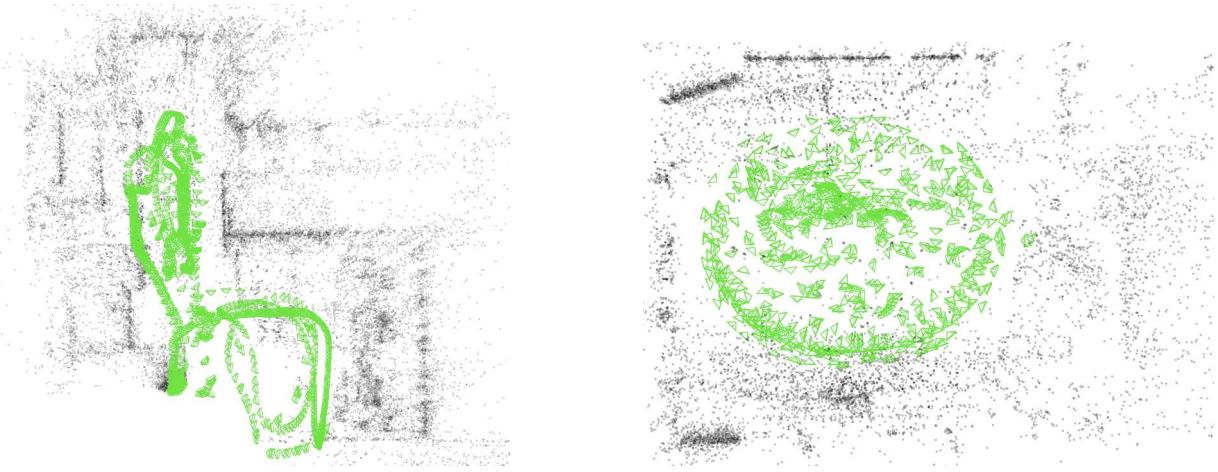
2. Evaluate the performance of my system on standardized datasets, **demonstrating its superior performance over comparable state-of-the-art systems**.

3. Create a simulation environment for testing and evaluating my system locally.

4. Deploy my system on physical robots, **demonstrating the practical use cases of this system and benchmarking real-world performance.** Additionally, **I was an author of the paper *The Cambridge RoboMaster: An Agile Multi-Robot Research Platform***, with my distributed SLAM system included as one of the experiments analyzed in the paper.

5. Develop *Multi-Agent EVO* – the first open-source evaluation library for multi-agent SLAM systems.

6. Develop the *Raspberry Pi Video Publisher* – a performant platform for SLAM data collection or augmented reality visualizations – and set up a continuous integration and deployment pipeline to automatically push the latest build to the hardware.



(a) EuRoC Machine Hall 01-03 **(b)** TUM-VI Rooms 1-3

Figure 1.1: Sparse maps built by my distributed SLAM system running industry-standard datasets.

Chapter 2

Preparation

2.1 Starting Point

Visual SLAM systems are a mature and well-researched subfield of computer science with many advanced implementations. To avoid spending the majority of my effort re-implementing a visual SLAM system from scratch, I instead used a **single-agent** visual SLAM implementation as the starting point for my project. The rationale behind this decision was that it would allow me to focus my efforts on the distributed multi-agent aspect of this project, which I believe is the novel and under-researched aspect of the field.

I chose ORB-SLAM3 [1] as the single agent SLAM system to base my system on top of, as it ranks at the top of benchmarks in a variety of environments [2] and its codebase is publically available. I primarily utilized the system's tracking and local mapping modules as well as some helper functions from the backend, which are all attributed during the discussion of my algorithms in the Implementation section.

While ORB-SLAM3 is an excellent SLAM system, it is fundamentally a single-agent system with no considerations in place for use in a multi-agent context. As I will later expand upon, a significant amount of time and effort was required to understand its extremely large undocumented codebase, especially since an almost complete understanding of its inner workings was needed to both extract and inject map information from the system. In retrospect, using an existing single-agent SLAM system as a foundation did not save as much time as it took almost a month to get ORB-SLAM compiled and running locally, and many more before I was able to make good progress on the distributed aspect of my project.

Nevertheless, using a cutting-edge single-agent SLAM system as a foundation for my project has allowed me to create a distributed SLAM system that performs better than comparable state-of-the-art systems, making it suitable for real-world applications.

At the time of submitting my project proposal, I had forked the ORB-SLAM3 git repository[1] and explored the codebase. ORB-SLAM3 is licensed under GPL-3.0, and as such, I have open-sourced my code under the same license.

I had no prior experience with SLAM systems but did research the current state of multi-agent visual SLAM systems to evaluate the feasibility of my project and to prevent it from being a duplication of prior work.

[1] `https://github.com/UZ-SLAMLab/ORB_SLAM3`

2.2 Visual SLAM Background

Before developing a distributed multi-agent SLAM system, we must first understand the basics of a visual SLAM. This is a topic on which numerous books [3][4] and research papers [5] [6] [7] have discussed in depth, which I will attempt to summarize here.

2.2.1 Visual Odometry

To grasp visual SLAM effectively, it is useful to start with a foundational concept – visual odometry (VO). VO is the process of estimating the trajectory of a camera based on the sequence of images produced by it. The core idea is to track features across consecutive images, and by observing how the features move relative to one another it is possible to infer the motion of the camera.

Take, for example, the two images from the KITTI00 dataset [8] presented in Figure 2.1. We can see that Figure 2.1b is taken a few meters in front of Figure 2.1a because the car on the left is closer in the second image, and the roof on the left becomes larger, et cetera. The goal of VO is to compute this change in pose between the two images.

(a) **(b)**

Figure 2.1: Example images from KITTI00 dataset for visual odometry example.

The first step towards determining the change in pose is identifying common features in both images. For this, we use feature descriptors, which find recognizable features within the image and represent them as a vector. A common feature descriptor used for VO and SLAM is the ORB descriptor [9], due to its low computational cost and invariance to rotation and scale, and we will use it for this example.

Figure 2.2 displays the ORB feature matches between the two images, however it is clear that many of them are incorrect. This is to be expected, as many elements of the images are repeated (windows, trees, etc).

We can remove the majority of incorrect correlations by constraining the matches to the epipolar geometry model – essentially telling the computer that the two images are the same scene captured from two different camera poses. Since there are a lot of outliers we use the RANSAC algorithm, later defined in subsection 2.6.3, to fit the data to the epipolar geometry model. This results in the matches identified in Figure 2.3, which are far more accurate. We can see that the window on the left is correctly matched in the two images even though there are many similar windows in the images, demonstrating the importance of fitting our matches to the two-camera epipolar geometry model using RANSAC.

We can now run the seven-point algorithm [10] on the matches to extract the rotation $\mathbf{R}$ and translation $\mathbf{t}$ between the two images, and consequently triangulate the world features found

in both images. This is displayed in Figure 2.4. By continuing this process for all image pairs in a video, we can estimate the trajectory of the camera as it moves through the world. It is important to note that the scale of the world is arbitrary when only using monocular video.

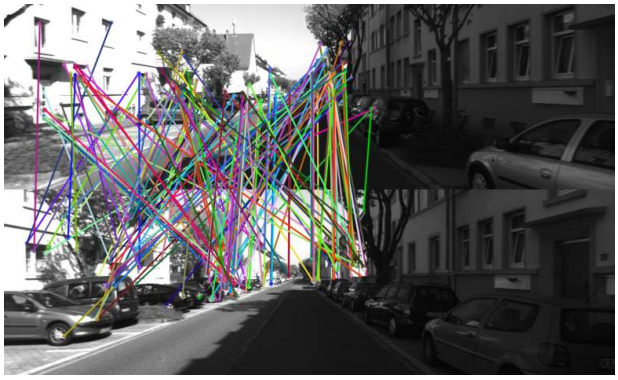


Figure 2.2: Feature descriptor matches between the two images.

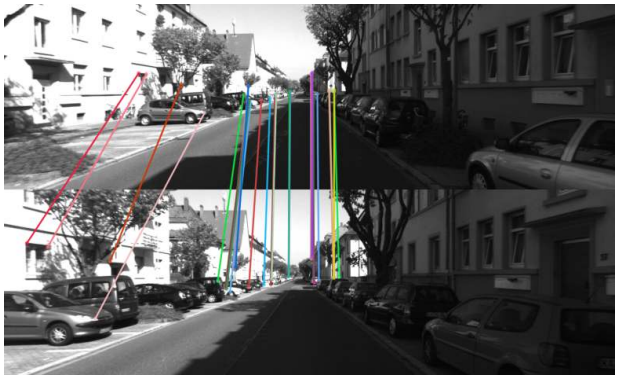


Figure 2.3: Inlier feature descriptors after fitting the data to the epipolar geometry model using RANSAC.

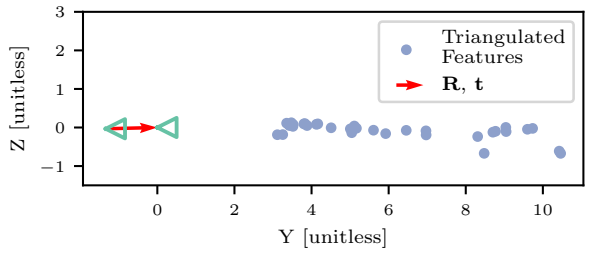


Figure 2.4: Side view of the estimated **R** and **t** between the two camera poses, and the triangulated feature points.

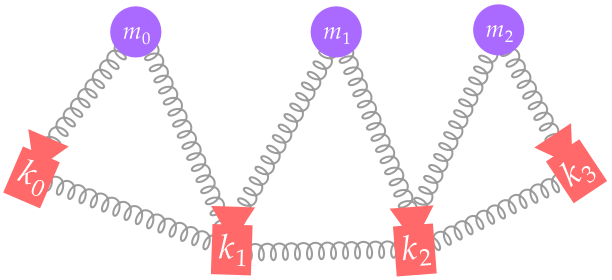


Figure 2.5: Graph pose optimization diagram, where k_i is a keyframe and m_i is a feature point. The springs represent constraints between the nodes.

2.2.2 Towards Visual SLAM

Visual odometry provides a local estimate of trajectories, however, it does not build a map of the world and therefore can not give an accurate global estimate of the agent's trajectory. Visual SLAM solves this problem by constructing a map of the environment as it moves through it, using the map to localize itself within the world. Modern systems break this down into three processes: Tracking, Mapping, and Loop Closing.

2.2.3 Tracking

The tracking step processes image data to compute the pose of the agent within the map. This step leverages many of the core ideas from visual odometry, using feature descriptors to find matches and fitting them to the epipolar geometry model using RANSAC. However, instead of matching features between consecutive images like visual odometry, we find feature matches between the input image and the map we have built. This is possible because the map is essentially a set of feature descriptors and their estimated position in 3D space, allowing our tracking module to find matches and localize the agent.

2.2.4 Mapping

Mapping extracts the data gathered during the tracking phase to create a 3D representation of the environment. Many popular visual SLAM implementations use a keyframe-based approach, where the map is estimated using only a few select image frames, ignoring the intermediate frames which provide little new information. This effectively decouples the mapping and tracking processes, allowing us to perform relatively costly but very effective pose graph optimizations.

Pose Graph Optimization is the process of minimizing the errors within our map, typically through the use of a graph optimizer. We represent keyframes and world features as nodes in our graph, with edges representing the constraints between nodes through a cost function. For example, if a keyframe observes a feature at position (-0.23, 0.64) in their image, an edge between the keyframe and feature will be created with a cost function that is minimized when the feature is re-projected onto the keyframe at the observed point. Similar edge constraints can be defined between keyframes using IMU measurements or wheel odometry. A graph optimizer can then be used to tweak the keyframe poses and feature locations to minimize the cost functions, therefore improving our map's accuracy.

We can intuitively think of the edge constraints as springs (Figure 2.5), all attempting to revert to their preferred length (given by the world observations). The graph optimizer moves the keyframes and features around until the system settles into its lowest energy state, where the map agrees with our observations as closely as possible.

2.2.5 Loop Closure

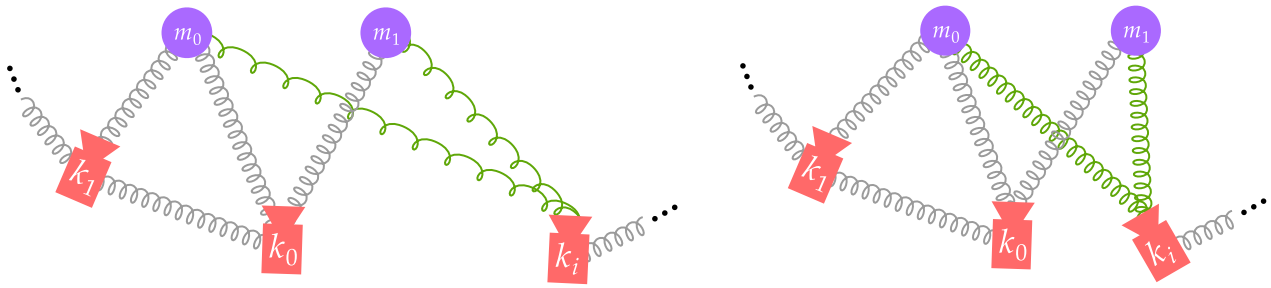
Loop closing is crucial for maintaining the long-term accuracy of the SLAM system. It involves recognizing when the agent re-visits a previously mapped location and "closes the loop" by correcting the errors that have accumulated over time. We often vectorize keyframes using the visual bag of words approach (later explained in subsection 2.6.2 is it bad to have the BoW explanation in the future? I think that moving the algorithm section to the beginning of this chapter messes with flow.) to identify when an agent re-visits an area. When a loop closure is detected, we fuse the duplicate features and use pose graph optimization to re-optimize the map and ensure global consistency, as illustrated in Figure 2.6.

2.3 Relevant Work

While single-agent SLAM systems are a relatively mature field of research, multi-agent systems are still very much in active development.

Centralized multi-agent systems such as CCM-SLAM [11] and COVINS [12] require a centralized server to perform map merges and PGO. While simpler to implement, this comes with the obvious limitations of centralized systems such as scalability issues and requiring existing networking infrastructure.

In recent years we have seen the emergence of a handful of decentralized multi-agent systems, however, many have various limitations. Systems such as [13] [14] [15] require the agents to



(a) A loop closure is identified and new constraints are added between k_i and m_0, m_1. The new constraints are initially "stretched" since m_0 and m_1's location in the map does not correlate with k_i's observation of them.

(b) The graph optimizer adjusts the keyframes and feature point poses so the map better aligns with the observations.

Figure 2.6: Loop closure pose graph optimization. The green edges represent the new constraints added by the loop closure.

be initialized with their ground truth poses, which greatly limits their real-world usability. In contrast, my system is able to provide accurate relative localization even when agents are initialized in arbitrary and unknown locations by identifying common landmarks in the world.

Figure 2.7 lays out the sensor configurations used by popular state-of-the-art multi-agent SLAM systems, showing that my system stands out as the only distributed system capable of operating with purely monocular visual data. This is advantageous, as LiDAR & RGBD sensors have considerable weight, and stereo cameras may require a minimum camera separation to operate, both of which limit their usage on small aerial robots.

System	Year	Collaboration Type	Monocular	Stereo	Monocular +IMU	Stereo +IMU	LiDAR +IMU	RGBD +IMU
My System	2024	Decentralized	X					
D^2SLAM[16]	2022	Decentralized				X		
CCM-SLAM[11]	2019	Centralized	X					
COVINS[12]	2021	Centralized			X			
Kimera-multi[17]	2021	Decentralized				X		X
Swarm-SLAM[18]	2024	Decentralized				X	X	X
DOOR-SLAM[19]	2020	Decentralized				X		

Figure 2.7: Comparison of modern multi-agent SLAM systems. My SLAM system is the only decentralized monocular system available.

Additionally, my system provides a novel approach to the Distributed Pose Graph Optimization (DPGO) problem. There are various approaches to DPGO. SWARM-SLAM [20] elects a single agent to perform the PGO for the entire swarm, which is simple but has a high communication overhead since all agents have to send their pose estimations before each optimization. Other systems perform DPGO by spreading computation across agents by using algorithms such as ARock [21] or Distributed Gauss-Seidel [22], however these methods still present a communication overhead and may stall in when presented with unreliable communications.

Instead of performing discrete optimization runs, my method of DPGO is performed incrementally. Each agent optimizes its pose graph as external data streams in, with a separate map alignment step. This method has no additional communication overhead, apart from the infrequent map alignment step, however, it comes at the cost of less verifiable global consistency. We evaluate my method's performance in the Benchmarking section, demonstrating its very

competitive real-world performance. The details of my implementation are presented in the Decentralized System Manager section.

2.4 Development Tools & Frameworks

Due to this project being a large software engineering undertaking, I knew that a well-structured development plan and carefully chosen frameworks would be essential to the successful implementation of this project. Much thought was put into using frameworks such as ROS and Docker, all of which allowed my system to easily be deployed to real-world robots.

In addition, an entire suite of infrastructure had to be developed to aid the development of my distributed SLAM system, including simulation environments, an evaluation library, and testing infrastructure.

2.4.1 Webots Simulation Software

Robotics projects work in the physical domain, however testing in the real world requires a large amount of setup and infrastructure. To ensure fast iteration, I decided to use simulations for the majority of my development. This allowed me to easily test my system in various environments and scenarios before deploying it to physical robots.

The details of integrating Webots into my project are explored in the section 3.6 section, including the ROS interfaces developed and agent controllers.

2.4.2 Testing Infrastructure

Along with being very useful for real-time testing, the simulation software enabled me to record numerous test cases which I have used as regression tests and benchmarks for my system throughout development.

I have built a central management interface that allows me to record and replay datasets from the simulation software, among many other functions. There are datasets for testing all core functionality of my SLAM system, which I would run at regular intervals during development to ensure that no features had regressed and to ensure performance was improving. The central management interface's implementation is further explained in section 3.4.

2.4.3 Robot Operating System 2 Communication Middleware

Robot Operating System (ROS) 2 is the glue holding my system together, allowing independent software processes and hardware to communicate through an abstract messaging interface.

ROS has long been the industry standard, being almost ubiquitous in both robotics research and the commercial sector. Confusingly, ROS is not an operating system at all, but instead, a cross-platform development framework that provides a middleware to facilitate reliable communication between independent processes called *nodes*. These nodes can be on the same device or

a device within the local area network and may be written in C++ or Python. Nodes communicate by *publishing* and *subscribing* to different *topics*, allowing both peer-to-peer communication and broadcasting.

Since every node is abstracted away behind the interface provided by the various messaging topics, we can easily swap out nodes in this system. For example, we can substitute the real camera for a simulated camera to test my system in a virtual environment without having to change any other part of the system – as shown in Figure 2.8. This makes transitioning between the real and simulated world almost seamless, which I knew would be essential for the eventual deployment on to physical robots.

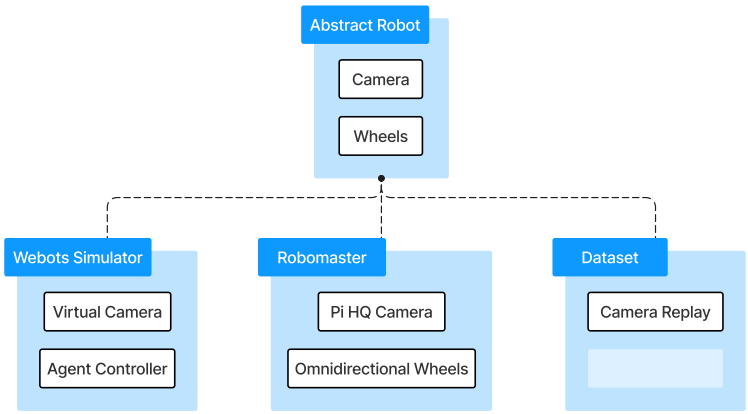


Figure 2.8: All robot types inherit the abstract robot interface as they all use the same ROS message types. This allows my SLAM system and its supporting infrastructure to be completely agnostic to the actual type of robot used.

Furthermore, using the ROS framework allows my code to be far more portable, as anyone can download my nodes, link the camera topics up to their robot's camera, and run my SLAM system with minimal effort. This allows my system to be easily reproducible and not tied to particular hardware.

There are two versions of ROS: ROS 1 and ROS 2. I chose to use ROS 2 due to its improved decentralized properties, which align with the goals of this project. ROS 2 conforms to the Data Distribution Service (DDS)[1] specification, which guarantees a reliable broadcast and, unlike ROS 1, it does not require a leader node when used in a multi-agent setup.

2.4.4 Repository Setup

It was essential for me to set up my repository properly before beginning work on this project. Since the majority of my initial effort was spent understanding and modifying ORB-SLAM3, I set up clangd to add smart code suggestions to my project based on my CMake profile. This was extremely helpful, as ORB-SLAM3 had no documentation or comments, and variables were often not given useful names. Additionally, I set up formatters to autoformat my C++ and Python code on save.

Throughout my project, I made extensive use of the GNU Project Debugger to identify bugs within my codebase, especially issues brought about by multithreading.

The five ROS packages in my repository also follow the standard ROS file structure, allowing others to easily understand and modify them.

[1] `https://en.wikipedia.org/wiki/Data_Distribution_Service`

2.4.5 Docker Containers

Docker containers are used on all physical deployments of my software, including the Cambridge RoboMasters used for real-world testing of my SLAM system and my Raspberry Pi Video Publisher platform which is used for custom dataset collection and augmented reality visualizations.

Docker allows the software to be isolated in self-sufficient environments, making it easy to deploy on different computing environments. Additionally, using Docker allows my code to be built once and then deployed on multiple robots, preventing the need to rebuild the system on every robot which saves a significant amount of time. These aspects of Docker are expanded upon in the section below.

2.4.6 Continuous Integration / Continuous Deployment

My GitHub repositories are set up to perform continuous integration via GitHub Actions. Every time code is pushed to the repository, all 5 core packages (`controller`, `interfaces`, `motion_controller`, `orb_slam3_ros`, `webots_sim`) are built to ensure there are no compile time errors.

Additionally, a Docker container is cross-compiled to `arm64` and uploaded to Docker Hub. These Docker images can then be downloaded to the Cambridge RoboMasters (the platform used for real-world testing) and immediately run. The pipeline is illustrated in Figure 2.9. This greatly speeds up development, as compiling the codebase locally on the Cambridge RoboMasters takes over 20 minutes for each robot.

Aside from the core packages, we also perform continuous integration as well as continuous deployment for the Raspberry Pi video publisher package. A Docker container is similarly cross-compiled to `arm64` and uploaded to Docker Hub, and it is automatically deployed to the Raspberry Pis so they will run the latest version of the package.

Continuous deployment makes sense for this use case as the Raspberry Pis are designed to be plug-and-play, starting video streaming as soon as they are turned on. Unlike the Cambridge RoboMasters which are frequently ssh'ed into to pull specific Docker images and start experiments, we want the Raspberry Pis to use the latest software as soon as it is pushed to the GitHub repository.

The ease of use of the Raspberry Pi ROS video publisher platform that I have developed has made them an invaluable tool and resolves the time-syncing issues previously faced when using the existing cameras in the Prorok Lab.

2.5 Datasets

A variety of industry-standard datasets were utilized to benchmark the performance of my SLAM system, including the EuRoC Machine Hall [23] and TUM-VI [24] datasets. While these datasets provide a variety of sensors, we only utilize a monocular video as input when evaluating performance. Further information about each dataset is provided in the Benchmarking section.

In addition to standard datasets, I also used the aforementioned custom datasets generated in the simulator simulated environment to use for testing.

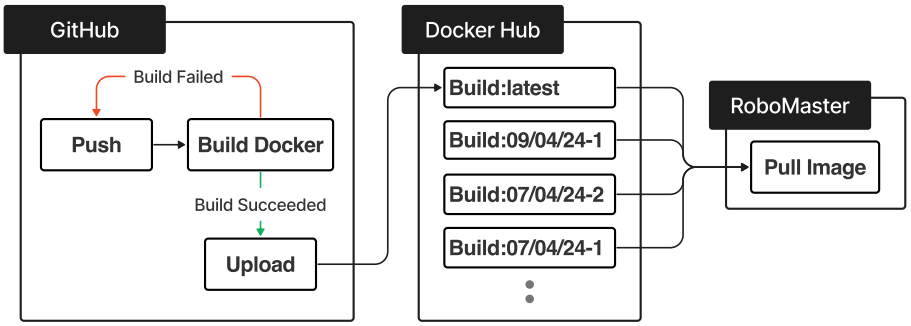


Figure 2.9: Automated continuous integration pipeline for the Cambridge RoboMasters. Continuous deployment is implemented for the Raspberry Pi video publishers by pulling from Docker Hub and running the container on startup.

2.6 Algorithms

This section will briefly cover some of the existing algorithms utilized within my SLAM system and can be used as a reference when reading the Implementation section.

2.6.1 Kabsch-Umeyama Algorithm

The Kabsch-Umeyama algorithm (Algorithm 1) is used to align a set of target points Q to a set of source points P with a SIM(3) transform. Within the field of SLAM, this is primarily used to align the estimated trajectory with the ground truth, as it may be translated, rotated, and scaled differently to the ground truth. Additionally, this algorithm is used for my multi-agent map alignment process.

Algorithm 1 Kabsch-Umeyama Algorithm

Input: Source points P, target points Q
Output: Rotation R, scale s, and translation t that best aligns Q to P
$n \leftarrow$ number of points in each set
$\mu_P \leftarrow \frac{1}{n}\sum_{i=1}^{n} p_i$
$\mu_Q \leftarrow \frac{1}{n}\sum_{i=1}^{n} q_i$
$P' \leftarrow P - \mu_P$ ▷ Centering P
$Q' \leftarrow Q - \mu_Q$ ▷ Centering Q
$\sigma_P^2 \leftarrow \frac{1}{n}\sum_{i=1}^{n} ||p_i'||^2$ ▷ Variance of P
$H \leftarrow \frac{1}{n}\sum_{i=1}^{n} p_i'^T q_i'$ ▷ Covariance matrix
$U, D, V^T \leftarrow \text{SVD}(H)$ ▷ Singular Value Decomposition
$S \leftarrow \text{diag}(1, 1, \ldots, 1, \det(U)\det(V^T))$
$R \leftarrow USV^T$ ▷ Rotation matrix
$c \leftarrow \frac{\sigma_P^2}{\text{tr}(DS)}$ ▷ Scale factor
$t \leftarrow \mu_P - cR\mu_Q$ ▷ Translation
return R, s, t

2.6.2 Visual Bag of Words

Visual bag of words representation is used to vectorize an image based on its features, allowing us to compare how similar two images are. My system utilizes this to efficiently detect potential map merges between agents.

To convert our image to a vector, we must first build a *vocabulary* of features – common and unique features that will each go on to represent a dimension of our image vector. We do this by first extracting features from a large dataset of images and grouping their descriptors using an algorithm such as K-Means clustering. The center of the largest clusters can then be used as our feature vocabulary. This vocabulary generation step only needs to be performed once, preferably using a wide variety of representative images (for visual SLAM this could be images from a variety of environments).

We are now able to vectorize images by counting the number of times each feature within our vocabulary exists in the image.

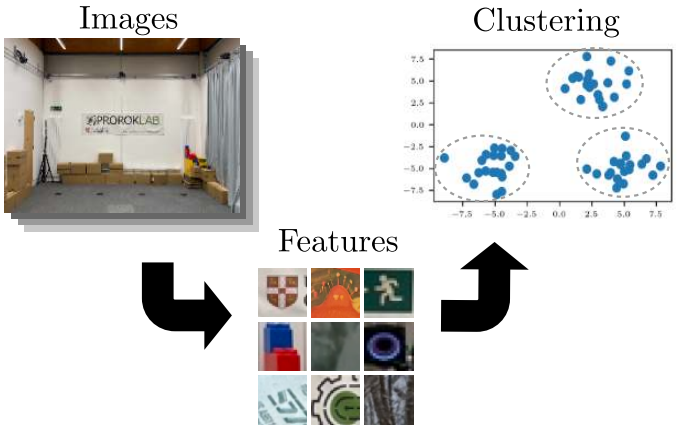


Figure 2.10: Process of generating the visual bag of word vocabulary. The center of each cluster is added as a feature in the vocabulary and represent a distinct vector dimension.

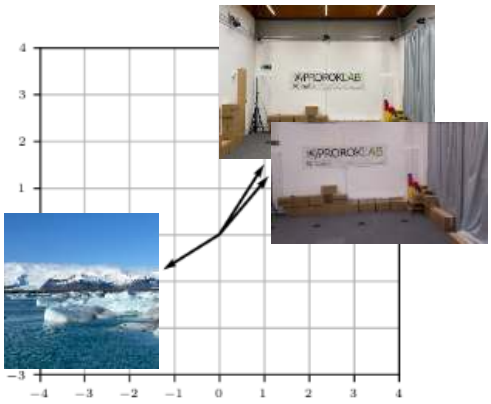


Figure 2.11: Visualization of image vectorization using the visual bag of words method. Images sharing common features result in closely aligned vectors.

2.6.3 RANSAC

RANdom SAmple Consensus (RANSAC) is an iterative method used to robustly fit a mathematical model to a dataset with a large number of outliers, with the outliers having no influence over the estimated model. This is particularly useful in visual SLAM as observations are extremely noisy, often containing far more outliers than inliers. Pseudocode for the algorithm is presented in Algorithm 2.

Any other algorithms you think i should explain?

2.7 Requirements Analysis

After further background reading on the subject and a review of other multi-agent SLAM systems, I broke my project down into the requirements shown in Figure 2.12. Features vital

Algorithm 2 RANSAC Algorithm

Input: dataPoints, model to explain data, number of iterations k, threshold τ, minimum number of data points needed to fit model n
Output: Best fitting model
bestmodel ← null
bestInliers ← {}
for $i = 1$ to k **do**
    samples ← n randomly selected data points
    model ← model fit to samples
    inliers ← {}
    for each point in dataPoints **do**
        if point fits model with error $< \tau$ **then**
            Add point to inliers
    if number of inliers > size of bestInliers **then**
        bestInliers ← inliers
        bestModel ← model
return bestModel

to achieving my core deliverables are given high importance, while extensions are given low importance. Additionally, features are given a risk level depending on how research-heavy they were, which helped me when planning and allocating time.

Feature	Importance	Risk
Simulation environment	High	Low
ORB-SLAM3 integration	High	Medium
Map serialization/deserialization	High	Medium
Decentralized system state manager	High	Low
Map merge finder and merger	High	Low
Handle losing communication	High	Low
Management interface	Medium	Low
Distributed pose graph optimization	Low	High
Map compression	Low	Medium
Visualization tools	Low	Low
Additional sensors	Low	High
Intelligent map sharing	Low	Medium
Motion controller[1]	Low	Medium
Real-world deployment[1]	Low	Medium
Augmented Reality visualization[1]	Low	High

Figure 2.12: Risk analysis.

[1]Requirements added after work had begun.

2.7.1 Development Model

This project lends itself best to the Spiral Development Model [25], as it consists of multiple well-defined features but requires frequent review and planning to decide what direction to push the project forward next. Developing my SLAM system was a large software engineering undertaking, involving the development of 5 ROS packages, an evaluation library, and even

hardware. I knew that a rigid plan would quickly fall apart as roadblocks occurred, and therefore, I chose the spiral model as it allowed me to adapt my plan as risks were identified and priorities were changed.

After implementing each core feature of my SLAM system, I would evaluate its performance with my testing and evaluation infrastructure to identify any unexpected issues and identify potential risks. This would influence my plan for the next iteration around the spiral, as well as my long-term plan for the project.

A good example of a pivot taken after re-evaluating my project was in late January, when the core features of my project were complete and I was planning the next stage of development. Discussions with my supervisor, as well as my own analysis of the project, revealed that the original extensions were no longer very relevant to my project, as they either had implicitly been achieved while developing my core features or would not yield particularly interesting results. We instead decided to focus on deploying to real robots to demonstrate real-world usability and performance. I am glad that I made this choice, as I believe it greatly strengthens the credibility of my SLAM system.

2.7.2 Licensing

All my code is available as open-source software on GitHub. The core SLAM system and Multi-Agent EVO library are released under the GPL-3 license [26] as it builds upon code released under that license. My Raspberry Pi Video Publisher is published under the MIT license [27].

Chapter 3

Implementation

3.1 Agent Architectural Overview

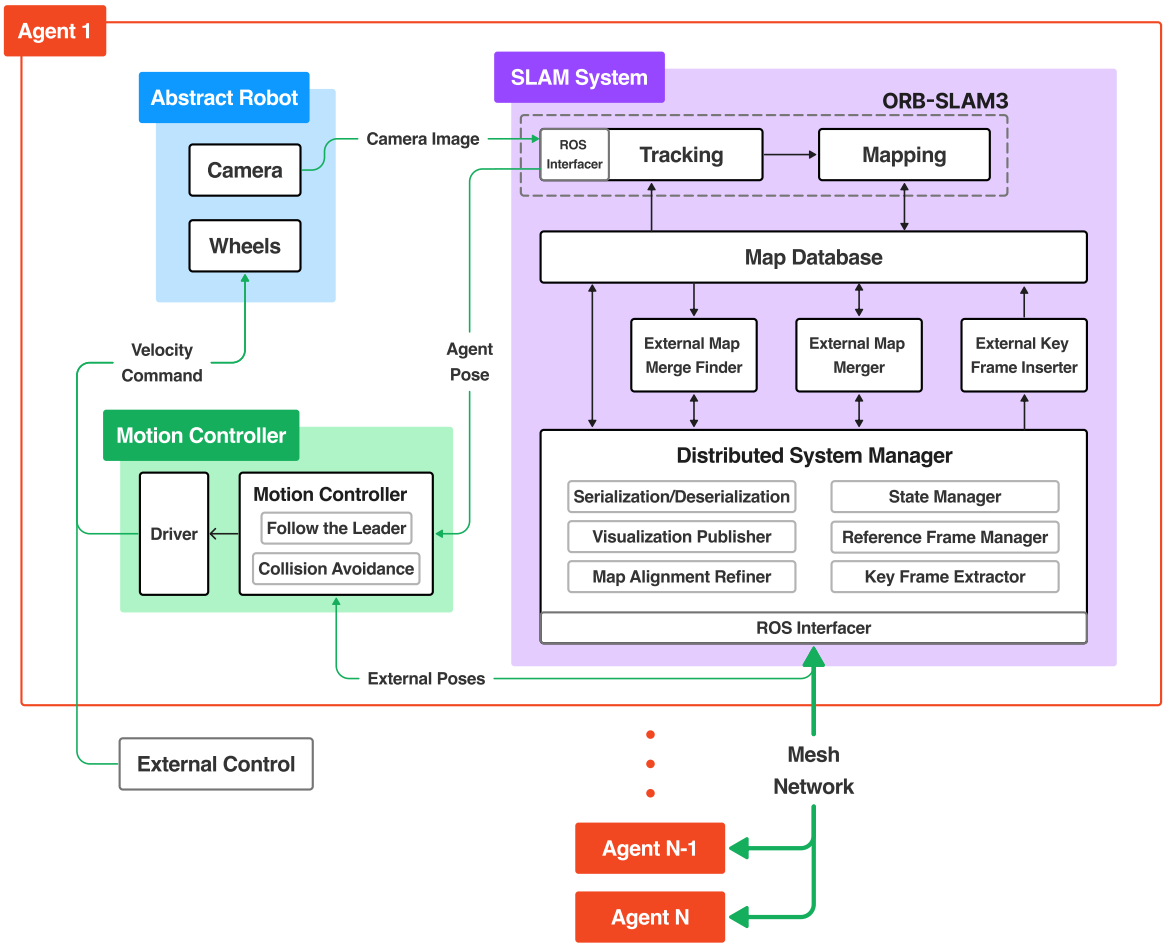


Figure 3.1: Agent diagram. Green arrows represent messages sent over ROS topics, while black arrows represent internal communications within a node.

Figure 3.1 gives an architectural overview of an agent, showing how the `Abstract Robot`, `SLAM System` and `Motion Controller` ROS nodes interconnect.

From a high level, we have an `Abstract Robot` node that provides an interface to the robot's hardware. This sends camera images to the `SLAM System` node, which builds a map of the world in collaboration with its peers. The `Motion Controller` node receives agent pose information from both the local `SLAM System` and the external peers to perform tasks such as collision avoidance by sending velocity commands back to the `Abstract Robot` node, closing the control loop.

In the following sections, we will explore these nodes in detail.

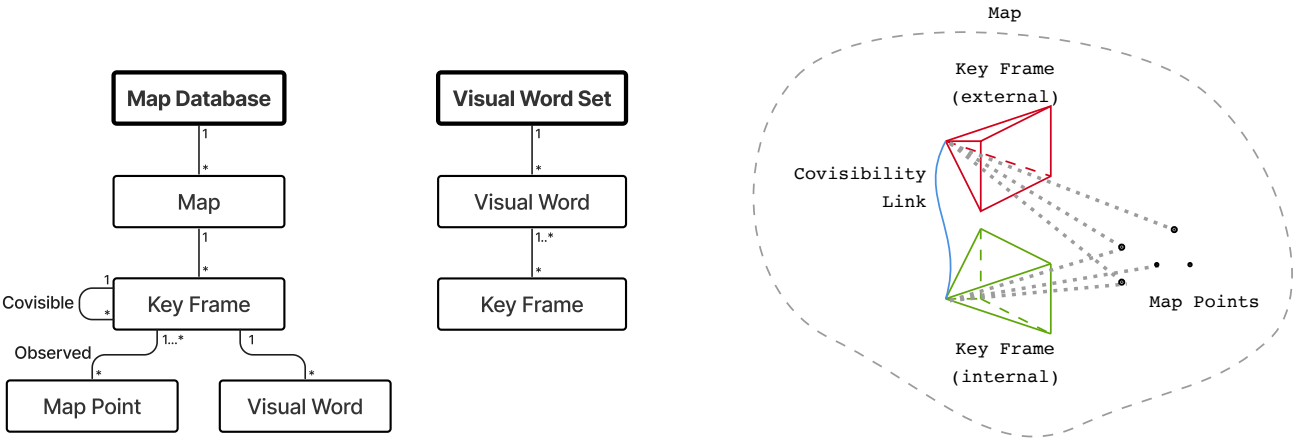
3.2 SLAM System

The SLAM System node is the majority of this project's implementation. It processes monocular images from the camera to localize the agent while also collaboratively building up a map of the world with its peers. As discussed in section 2.1, my system is based on an existing single-agent SLAM system that performs the `Tracking` and `Mapping` tasks. While substantial modifications were made to the base single-agent system, I will generally focus on the decentralized aspects of the system in the interest of space.

3.2.1 Datastructures

A brief discussion of my SLAM system's internal data structures is required to understand the following sections. Figure 3.2a shows the two key data structures: the map database and the visual word set. The map database contains multiple maps, which primarily consist of keyframes and map points. Keyframes are "snapshots" of the world, containing the predicted pose of the agent at that time along with the observed world features which we call "map points". These map points can be observed by multiple keyframes, as shown in Figure 3.2b, and when this happens we connect the keyframes with a "co-visible" link.

Keyframes also have a "visual bag of words" representation, which is the vector describing the visual contents of the keyframe. These are essential for detecting potential map merges, so we also build a "visual word set" that allows us to find all keyframes that contain a particular visual word[1].



(a) Data structure diagram

(b) Visual representation of data structure

3.2.2 Decentralized System Manager

Decentralized SLAM systems have significantly more complexity than single-agent or even centralized systems, due to the complex interactions between agents as they merge maps, lose localization, or lose connection with the network. Therefore, a robust framework must be put in place to ensure the correctness of the system, which I have implemented in the `Decentralized System Manager` component.

[1] A "visual word" is a feature from the bag of words vocabulary.

For the sake of simplicity, the next few sections will explore the interactions between just two agents: a *local* and *external* agent. In the Generalizing to $N \geq 3$ Agent Systems section, we will show how this easily generalizes to a system with an arbitrary number of agents.

3.2.2.1 State Manager

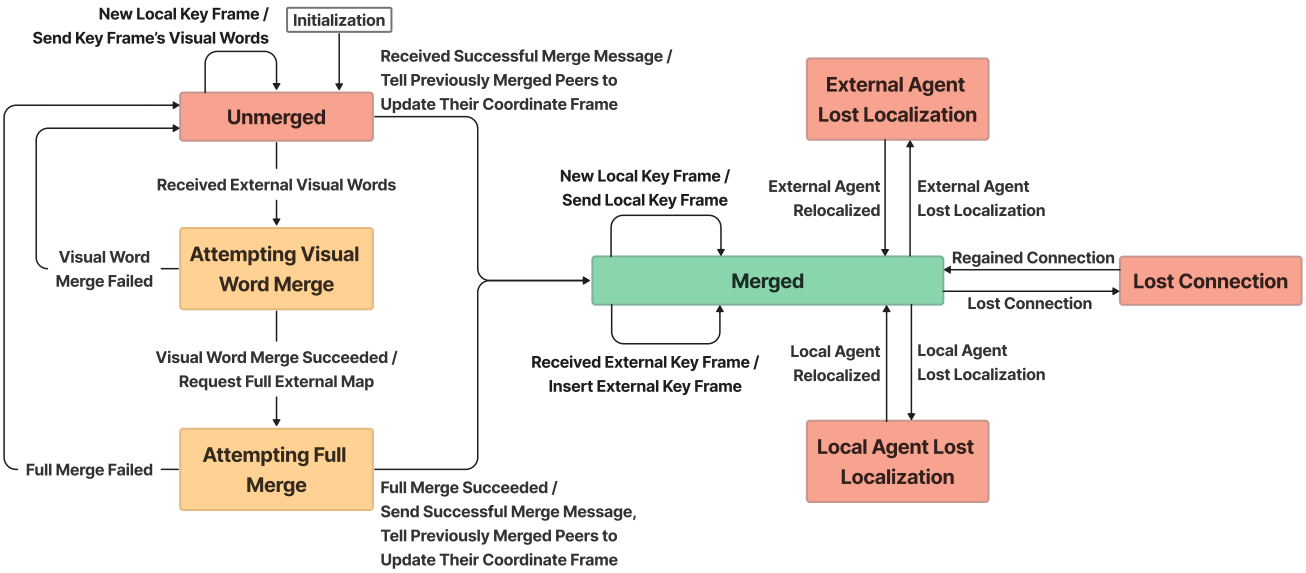


Figure 3.3: SLAM system state machine for a single peer. Mealy machine notation is used, where each transition is defined as: `input/output`.

Each agent's `Decentralized System Manager` maintains a state machine for every peer in the system, shown in Figure 3.3. All peers are initialized in the `unmerged` state, meaning that they are in a different coordinate frame and can not collaboratively build a map together. As the local agent starts to explore the same locations as its peers, the system recognizes the visual overlaps and merges their maps, bringing us to the `merged` state where the agents share the same coordinate frame and map, enabling relative positioning and collaborative map building.

3.2.2.2 External Map Merge Finder

A naive approach to merging maps with an external agent is to constantly exchange our full maps, each time trying to identify if a map merge is possible. While simple, this approach does not scale well from both a networking and computational perspective, as maps are often 1MB in size, and computing a full map merge is extremely computationally expensive.

Instead, we first identify if a map merge is even feasible by using visual words. This eliminates superfluous map merge attempts that have no chance of succeeding because the agents' maps have no visual overlap.

As the agents generate new keyframes, we use DBoW2 [28] to calculate the visual bag of words seen by that keyframe and send them to our peers. These visual words are significantly smaller than sending over the complete map (as shown in Figure 3.4) and enable agents to detect if there is a significant amount of visual overlap between its local map and the external agent's map.

Algorithm 3 calculates if a merge is found by comparing the merge score of the external visual words to a dynamic baseline score, allowing the system to generalize to different environments. Algorithm 4 is the subprocedure used to calculate the merge score of a visual bag of words into the local map, and exploits the spatial locality of keyframes to give a more robust score.

This method gives a recall of almost 100%, at the cost of potentially lower precision. This is a worthwhile tradeoff, however, as it is essential to have very few false negatives so we can merge maps as soon as possible and the agents can begin collaborating.

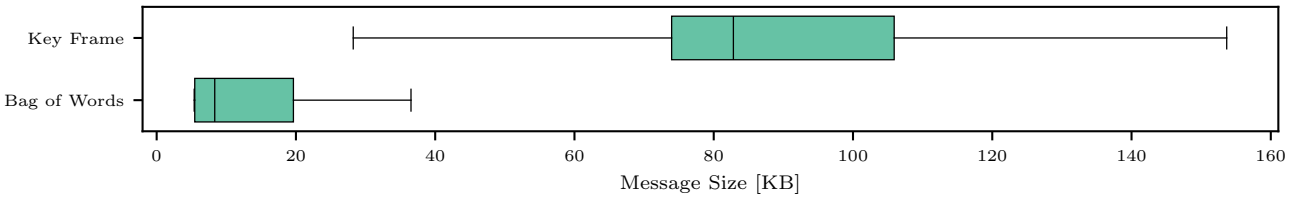


Figure 3.4: Comparison of the message size difference between a raw keyframe and a bag of words representation of a keyframe.

Algorithm 3 Map merge finder using visual words.

Input: VisualWords_{ext}: Set containing external keyframe's visual words
Output: Success: Boolean value signaling if a merge is possible based on visual words

1: $(\text{MergeScore}_{ext}, \text{BestMatchKeyFrame}) \leftarrow \text{CalculateMergeScore}(\text{VisualWords}_{ext})$
2: $(\text{MergeScore}_{baseline}, \_) \leftarrow \text{CalculateMergeScore}(\text{BestMatchKeyFrame's visual words})$
3: $\text{Success} \leftarrow \text{MergeScore}_{ext} \geq 0.7 \times \text{MergeScore}_{baseline}$

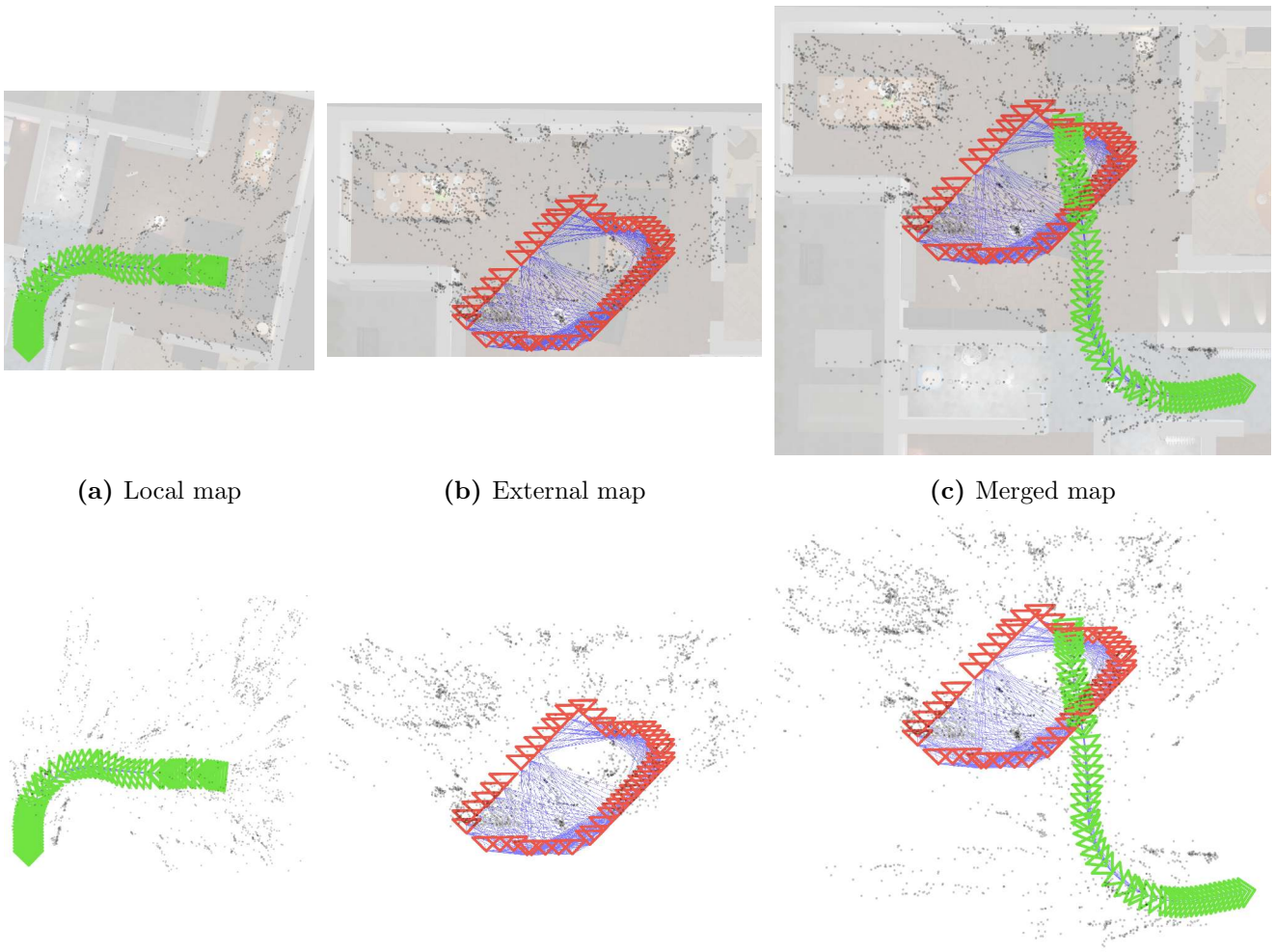
Algorithm 4 Calculate how well a bag of visual words merges with the local map.

```
1:  procedure CalculateMergeScore(VisualWords)
2:      PotentialMatches ← query Visual Word Set data structure for similar keyframes
3:      BestMatchKeyFrame ← null
4:      BestMergeScore ← 0
5:      for each KeyFrame_0 in PotentialMatches do
6:          MergeScore ← KeyFrame_0's similarity to VisualWords
7:          Covisible ← 5 keyframes with highest covisibility with KeyFrame_0
8:          for each KeyFrame_cov in Covisible do          ▷ Exploit spatial locality
9:              MergeScore += KeyFrame_cov's similarity to VisualWords
10:         if MergeScore ≥ BestMergeScore then
11:             BestMergeScore ← MergeScore
12:             BestMatchKeyFrame ← KeyFrame_0
        return (BestMergeScore, BestMatchKeyFrame)
```

3.2.2.3 External Map Merger

Performing full map merges is perhaps the most important component of the decentralized SLAM system, as a bad map merge will make it impossible for agents to collaborate. Additionally, it is one of the most computationally intensive parts of this system, therefore, it has gone through numerous iterations to optimize performance and reduce map merge errors.

The external map merger leverages modifed components of the ORB-SLAM3 loop closure module, as it is a fundamentally similar problem. The final version of the map merger works as follows:



(a) Local map **(b)** External map **(c)** Merged map

(d) Local map **(e)** External map **(f)** Merged map

Figure 3.5: Visual overview of the map merge process. We apply the translation $T_{loc \to ext}$ to the local map (a), and merge it with the external map (b), resulting in the merged map (c). The merged map contains information from both maps, creating our unified map.

The apartment environment being mapped is overlayed behind the visualization. overlay or no overlay?

1. **Request external agent's full map.**
   Once a potential map merge with the external agent is identified by the external map merge finder, we can begin a full merge attempt. We first request the full map from the external agent. Once the map is received, we deserialize it and temporarily place it into our `map database` data structure as a separate map.

2. **Confirm that a map merge is possible using the full map data, and find the translation $T_{loc \to ext}$ that aligns the local map to the external map.**
   The visual word map merge finder identifies potential merges, however, we still need to confirm that a map merge is possible using the full map data from both agents. This is performed using a modified version of the ORB-SLAM3 loop closure detector.

   If we determine the merge to not be possible, we delete the external map from our map database and exit this process.

3. **Apply translation $T_{loc \to ext}$ to our local map.**
   This shifts the local agent to be in the external agent's coordinate frame.

4. **Move a subset of the external map's keyframes and map points into our local map**
   Given the external keyframe k_0 whose visual words triggered this map merge attempt, we extract a *local window* K_{wind} of keyframes connected to k_0 in the co-visibility graph and move them to our local map. Moving only a small number of keyframes at first allows the map point merge and pose graph optimization process to be completed faster, which is important as they block local mapping from being performed.

5. **Merge local and external map points, connecting the maps**

6. **Run pose graph optimization to optimize map point and keyframe positions.**

7. **Repeat steps 5-7 with the entirety of the external map.**

8. **Broadcast successful merge message.**
   If the full map merge is ultimately successful, we broadcast a `/successfully_merged` message to tell our peers that we have successfully merged with the external agent. The external agent will then move to the `merged` state and both agents will begin sharing keyframes with each other, allowing the external agent to receive the local agent's keyframes and keeping the agent's maps in sync.

It is important to note that this system requires only one agent to identify and calculate the map merge, significantly reducing the computational overhead of map merging, especially in systems with many agents (further explained in the Generalizing to $N \geq 3$ Agent Systems section).

3.2.3 Map Serialization and Deserialization

Map serialization and deserialization are essential and non-trivial components of this SLAM system, allowing agents to share their maps across the network. For this task, I used the Boost [29] Serialization C++ library since it supports standard library collections and other common classes.

As discussed in section 3.2, maps contain keyframes and map points. Individually, these objects are relatively easy to serialize with boost – in fact, the single agent SLAM program my system is based on already supported saving and loading maps allowing me to leverage some of their serialization helper functions. To serialize these objects we simply need to define a serialization scheme for each class, describing which instance variables should be serialized and which shouldn't. Strategically selecting only the parameters that need to be serialized allows us to cut back on communication overhead. For example, there is no need for us to serialize a keyframe's raw image, as it is not used in the rest of our SLAM pipeline.

Complexities arise when we try to serialize/deserialize the connections between these objects, especially when we are only sending map fragments. For example, if we send a new keyframe k and its map points M_k to an external agent, due to the many-to-many relationship between keyframes and map points, some of the map points in M_k may have already been sent before. We can define $M_k = M_{k\_sent} \cup M_{k\_unsent}$, where M_{k_sent} are the map points that have already been sent to the external agent and M_{k_unsent} are the ones that have not. Map points in M_{k_sent} will need to be connected to k when it is deserialized by the external agent, and map points in M_{k_unsent} will need to be sent to the external agent and connected to any existing keyframes in

the external map that observe the map points. These connections are further explained in the External KeyFrame Inserter section.

We manage these connections by giving every keyframe and map point a universal unique identifier (UUID), allowing us to use these UUIDs as references to a specific object in our multi-agent system. We use UUIDs since they do not require a centralized server or any communication with our peers to assign a unique ID to every object[1].

This method of using UUIDs as references is used to rebuild all connections after deserialization, including the keyframe-to-keyframe connections that build the co-visibility graph and many others that I have not had the space to discuss in this dissertation.

3.2.3.1 External KeyFrame Inserter

Once the local and external agents have merged their maps and are in the same coordinate frame, they can begin sharing keyframes with each other.

Each agent maintains a set of unsent keyframes K_{unsent} and map points M_{unsent}. Once $\#K_{unsent}$ exceeds a certain threshold, we serialize K_{unsent} and M_{unsent} and send them to the external agent. Finally, we set $K_{unsent} = \emptyset$ and $M_{unsent} = \emptyset$.

Upon receiving the serialized keyframes and map points, the external agent deserializes them and adds them to a queue to await insertion into their local copy of the shared map using the `external keyframe inserter`.

The external keyframe inserter is run whenever we have spare cycles on the CPU, to prevent impacting the local tracking and mapping performance. The insertion process involves the following operations:

(A visual overview of the process is presented in Figure 3.6)

1. **Pop external keyframe k_{ext} from front of queue.**
2. **Move k_{ext} and its external observed map points M_{ext} to the local map**. Since the local and external agents are merged and in the same coordinate frame, we can simply move k_{ext} and M_{ext} to our local map without any transformations.[1]
3. **Relink k_{ext} with co-visible keyframes and observed map points in the local map**.
   k_{ext} contains references to its co-visible keyframes and child map points that have already been sent or were generated by another agent. We search our local map for objects that match these references, reconnecting them.

[1]While not *verifiablly* unique, UUIDs are unique within practical limits. A commonly cited anecdote is that it is far more likely for a cosmic ray to cause a bug than a UUID collision.

[1]I had previously used a *keyframe anchor* method, where instead of k_{ext} having an absolute pose we would send its pose relative to the previous k_{ext}. The thought process behind this was to prevent minor misalignments between the local and external maps from preventing external keyframes from properly integrating with the local map. However, experimental testing showed that this method instead caused the local and external maps to frequently diverge.

4. **Relink M_{ext} with keyframes in the local map which observe it**.
   M_{ext} contains references to keyframes that observe it which have already been sent or were generated by another agent. We search our local map for keyframes that match these references, reconnecting them.

5. **Merge M_{ext} with map points in the local map**.
   M_{ext} will already be correctly linked to observing keyframes in the local map from step 4., however, due to communication latency some map points in M_{ext} may be duplicates of existing map points in the local map. Therefore, we exploit spatial locality to combine duplicate map points that describe the same physical feature. A key observation from testing was that this step is essential to ensuring the local and external keyframes stay well connected, ensuring the local and external maps do not diverge.

6. **Perform a local pose graph optimization around k_{ext}**.
   Since we have merged map points in the previous step, we need to a local perform pose graph optimization to tweak the keyframe and map point locations to minimize reprojection error. This helps create a more accurate map, minimizing trajectory error. Since we only perform this optimization with spatially local keyframes and map points, the computational overhead is relatively low.

3.2.3.2 Local KeyFrame Inserter

As mentioned in the External KeyFrame Inserter section, it is essential that the local and external maps stay well connected, sharing the majority of their map points. In other words, we must ensure that the external map data is integrated throughout the whole SLAM pipeline.

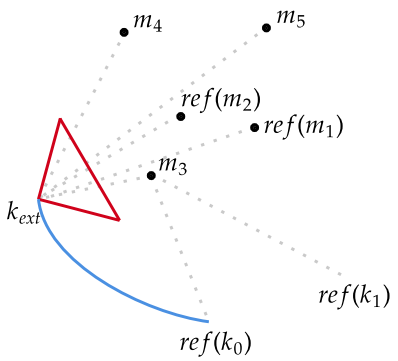
The `Tracking` and `Mapping` modules, which are responsible for localizing the robot and generating new keyframes, only interact with the `Map Database`, as seen in the previously shown architectural diagram in Figure 3.1. Our external keyframe inserter does all the work of properly reconnecting external map points and merging duplicate map points, leaving the `Map Database` data structure looking as if all the map points and keyframes were generated locally. This abstracts the `Tracking` and `Mapping` modules away from the distributed aspects of the system, ensuring that they fully utilize the external map points when localizing the agent and generating new keyframes.

3.2.4 Generalizing to $N \geq 3$ Agent Systems

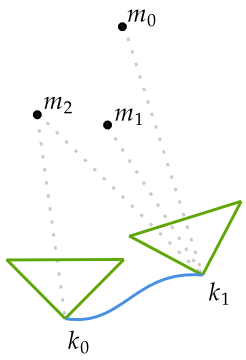
Now that we have explained how a pair of agents interact, we can explore how this can be generalized to a system with an arbitrary number of agents.

We assume a system with N agents $A = \{agent_1, agent_2, ..., agent_N\}$, where $agent_i$ is the agent with ID = i. Within this system we maintain a state $S_i = state_{n-m}$ for every agent pair $(agent_n, agent_m)$, giving us a total of $N(N-1)/2$ states. Conceptually, this is a fully connected graph with agents as the nodes and states between agents as the edges. We also have a set G which contains all the groups of merged agents. The *group leader* is defined as the agent in a group with the lowest agentId.

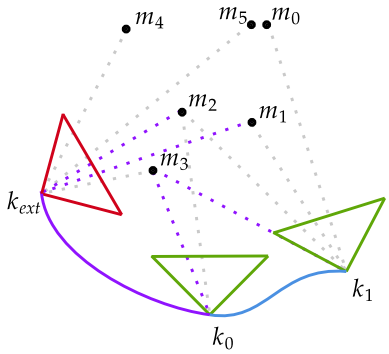
In the case where agents can lose communication with one another, we also assume that if any given $agent_n$ is able to communicate with an $agent_m$, $agent_n$ can also communicate with all of



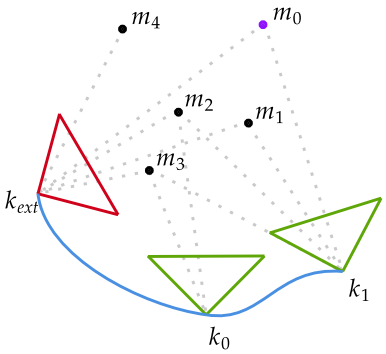
(a) External keyframe k_{ext} with $M_{ext} = \{m_3, m_4, m_5\}$. Note the references to existing keyframes and map points.



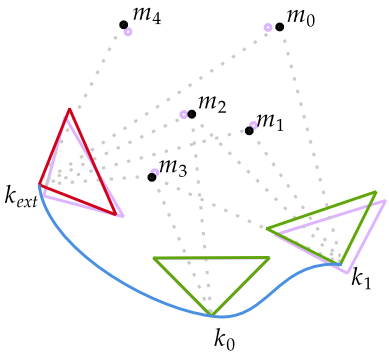
(b) Existing local map.



(c) Step 2, 3&4: Move k_{ext} and M_{ext} to the local map and relink references. Relinked connections are drawn in purple.



(d) Step 5: Merge duplicate map points. m_5 and m_0 have a similar feature descriptor and location, therefore they are merged.



(e) Step 6: Local pose graph optimization around k_{ext} to refine the map using the new information. Original keyframe and map point locations are shown in purple.

Figure 3.6: Visual overview of inserting external keyframes and map points into the local map. External keyframe (a) and initial local map (b) are combined to create our final local map (e).

$agent_m$'s connected peers. This is held if the agents are using a mesh network to communicate, for example.

Initially, every agent pair is unmerged so every state in S is set as *unmerged* and $G = \{\{agent_1\}, \{agent_2\}, ..., \{agent_N\}\}$

A key insight of my distributed SLAM system is to delegate all merge operations to group leaders. This means that instead of all agents attempting merges with every other agent, only group leaders have to attempt merges amongst themselves. This is significant, as the computational load of these merge operations scale proportional to the square of the number of agents involved, and in most use cases the number of group leaders quickly drops to be much lower than the total number of agents.

Additionally, having all merge operations performed by the group leader prevents potential race conditions introduced by communication latency within a group, for example two agents within a group both merging with different agents at the same time.

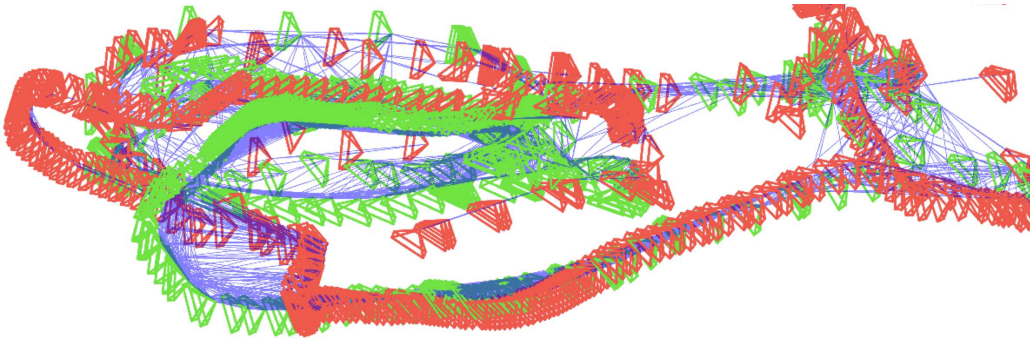


Figure 3.7: Section of the keyframes generated when running the EuRoC Machine Hall dataset. The local (green) and external (red) keyframes are well connected by covisibility links (blue), demonstrating that they are tracking the same map points.

As discussed in the External Map Merge Finder and External Map Merger sections, the two major operations needed to merge are (1) exchanging visual words to identify visual overlap and therefore merge opportunities and (2) sending over the map and attempting a full map merge.

Tackling (1) first, instead of the group leader only sending the visual words of its own keyframes, it will send the visual words of all keyframes generated by agents within its group. This introduces no additional intra-group communication, as all agents within a group already send each other all new keyframes.

Moving on to (2), once a pair of group leaders $(agent_n, agent_m)$ (with $n < m$) have successfully merged their maps, the agent with the larger ID ($agent_m$) will change its coordinate frame to the agent with the smaller ID ($agent_n$). $agent_m$ will then send the transform from $agent_m$ to $agent_n$'s coordinate frame $T_{m \to n}$ to the other agents in its group, allowing them to also change to $agent_n$'s coordinate frame. After this has been completed, $agent_m$'s group merges into $agent_n$'s group by updating S according to Equation 3.1, and agents begin exchanging keyframes to update each other's maps. Once all agents have merged together, they will all be in $agent_0$'s coordinate frame, as it has the lowest ID.

Agents are able to keep track of the current groups and group leaders within the decentralized system since all successful merge messages are broadcast on the shared `/successfully_merged` ROS topic.

$$\begin{gathered} \forall i \in g_n.\ \forall j \in g_m.\ state_{i-j} \in S \text{ and } state_{i-j} = \text{“}merged\text{”} \\ \text{where } g_n, g_m \in G \text{ are the groups that } agent_n \text{ and } agent_m \text{ respectively lead.} \end{gathered} \tag{3.1}$$

This whole process is perhaps best represented in visual form by the simple 3-agent merge example given in Figure 3.8 which shows the messages sent between the agents.

3.2.4.1 Map Alignment Refiner

As our shared map grows, the maps stored locally by the agents may "fall out of alignment". By this, we mean that the maps are slightly translated, rotated, or scaled with respect to the lead agent's map. This is largely a side effect of our aggressive early merge strategy which

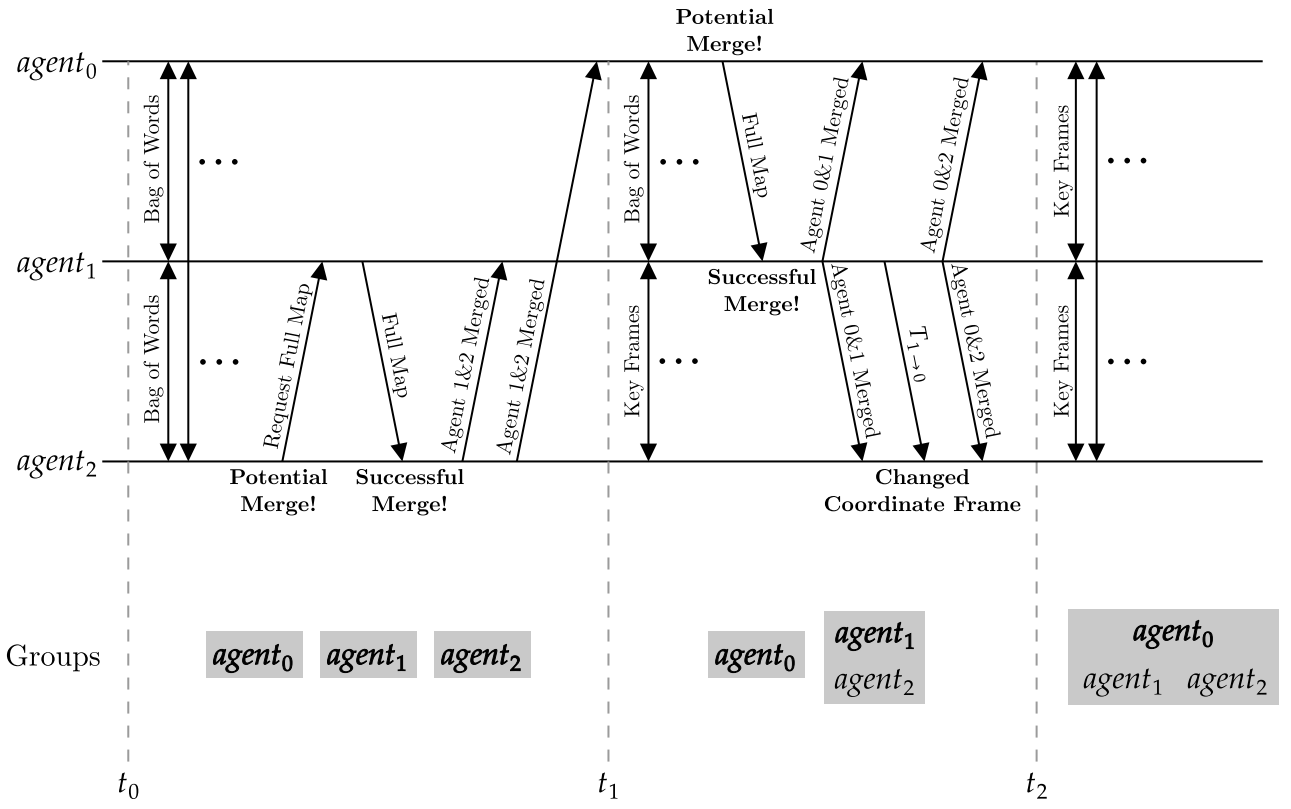


Figure 3.8: This is a simple 3 agent merge example where $agent_1$ and $agent_2$ merge first, and then $agent_0$ and $agent_1$. Agents in the same group are shown in the same rectangle, and the group leader is bolded.

Initially at t_0, all agents are unmerged and therefore share the bag of words representations of new keyframes with each other. A potential merge with $agent_1$ is then detected by $agent_2$. Since $2 > 1$, $agent_2$ needs to perform the merge so it requests $agent_1$'s full map. It successfully merges with $agent_1$'s full map and broadcasts this information to all other agents.

At t_1, $agent_1$ and $agent_2$ are merged therefore they begin exchanging keyframes. $agent_0$ and $agent_1$ are the two remaining group leaders, so they continue to exchange the new keyframe's bag of words. $agent_0$ detects a potential merge, and since $0 < 1$ it sends its full map to $agent_1$. $agent_1$ is able to successfully merge with $agent_0$'s full map, so it broadcasts that $agent_0$ and $agent_1$. Additionally, it sends the transform $T_{1\to 0}$ ($agent_1$ to $agent_0$'s coordinate frame) to $agent_2$ allowing it to change to $agent_0$'s coordinate frame. $agent_1$ then broadcasts that $agent_0$ and $agent_2$ have implicitly merged.

At t_2, all three agents are merged and in $agent_0$'s coordinate frame, and therefore they all exchange keyframes with one another.

This example is slightly simplified as all potential merges result in successful full merges. In reality, multiple potential merges may be identified before two agents successfully merge.

may merge two agents' maps before there is significant overlap, causing the estimated map alignment (ie. transformation between the two agents' origins) to have some error.

These small alignment errors are completely acceptable when maps are small, but may cause the maps to diverge as they grow.

To remedy this problem, we continuously refine our map alignment using RANSAC and the Kabsch-Umeyama point alignment algorithm. Map alignment is performed as follows:

1. **Request map point locations from the lead agent**.
   This is defined as the set TaggedMP_{ext} where $\text{TaggedMP}_{exti} = (\text{uuid}, (x, y, z))$

2. **Extract local map point locations**.
   This is defined as the set $TaggedMP_{local}$ where $\text{TaggedMP}_{locali} = (\text{uuid}, (x, y, z))$

3. **Use the Kabsch-Umeyama and RANSAC algorithms to find the transform $T_{local \to ext}$ which best aligns TaggedMP_{local} to TaggedMP_{ext}.**
   The Kabsch-Umeyama algorithm finds the SIM(3) transformation T from TaggedMP_{ext} to TaggedMP_{local}, minimizing the RMSE. However, our input data may contain a large number of outliers so we use RANSAC to find a good fit while ignoring outliers.

   The RANSAC and Kabsch-Umeyama algorithms are described in detail in subsection 2.6.3 and subsection 2.6.1 respectively.

4. **Apply transformation $T_{local \to ext}$ to our local map.**

We use an *additive increase multiplicative decrease* methodology to control how often map alignment is performed. Given t_i is the time between the i-th and $(i+1)$-th map alignments, we set $t_{i+1} = t_i + 1$ if the maps were well aligned, and $t_{i+1} = t_i/2$ if the maps were not well aligned. This prevents agents from continuously performing map alignments when their maps are already well aligned.

3.2.4.2 Losing Localization / Connection

An agent can lose localization within the map (for example, if the camera is blocked for a short period of time). When this happens, the agent signals to the other agents that it has lost localization and they temporarily stop sharing new keyframes. When the agent relocalizes itself, keyframe sharing resumes, and the agents send all the stored-up K_{unsent} and M_{unsent}.

Losing connection is very similar, where keyframe sharing is stopped until the connection is regained, at which point the agents update each other with the stored-up unsent keyframes.

3.2.4.3 Visualization Publisher

All 3D visualizations are created by publishing ROS markers, a standardized ROS message type that represents some object in 3D space such as lines, points, or meshes. These ROS markers can then be displayed by a variety of third-party visualization tools such as RViz [30] and Foxglove Studio [31].

Using ROS markers allows me to leverage these mature visualization tools instead of creating one from scratch, and also allows visualizations to be easily saved and replayed for later analysis.

TODO: add cool visualization

3.3 Motion Controller

While not a part of the core SLAM system, the motion controller node closes the control loop and demonstrates the real-world usability of my system. From a high level, the motion controller node consumes the local and external agents' poses and outputs a command velocity to the robot's movement system – all via ROS topics. For this project, I have implemented two different motion control systems which can be switched between seamlessly.

3.3.1 Follow The Leader

The *follow the leader* motion controller consists of two agents: one leader and one follower. The follower is given a position and rotation offset to the leader which it attempts to maintain as the leader is moved around.

For example, you could set the follower to be 1 meter behind the leader with a 180° rotational offset. This would mean the leader and follower never have visual overlap at any given moment, demonstrating that my SLAM system is truly building a shared map.

Spinning the agents around in a circle demonstrates that the two agents' maps are properly merged and that the agents are using map points generated by an external agent to localize themselves, giving accurate relative positioning even when there is no visual overlap at any given moment.

TODO: add figure of the map built

3.3.2 Multi-Agent Collision Avoidance

The *multi-agent collision avoidance* example is more complex, employing a non-linear model predictive controller (NMPC) to avoid collisions with both static and dynamic obstacles. My NMPC system is improved version of [32] and is defined as follows:

3.3.2.1 Non-Linear Model Predictive Controller Formulation

We assume an agent with current pose $\boldsymbol{p} \in \mathbb{R}^2$ and radius r. Firstly, we define our agent's control input $\boldsymbol{v_{cmd}}$ as a function describing its velocity over time. We can then define our agent's state as a function of time $\boldsymbol{x}(t)$ as the control input $\boldsymbol{v_{cmd}}$ applied to its current pose $\boldsymbol{p}$:

$$\boldsymbol{x}(t) = \boldsymbol{p} + \int_0^t \boldsymbol{v_{cmd}}\, dt \tag{3.2}$$

Solving the following minimization problem will yeild an optimal $\boldsymbol{v_{cmd}}$:

$$\begin{aligned} &\min_{\boldsymbol{v_{cmd}}} \int_{t=0}^{T} J_x(\boldsymbol{x}(t), \boldsymbol{x_{ref}}(t)) + J_s(\boldsymbol{x}(t)) + J_d(\boldsymbol{x}(t))\, dt \\ &\text{subject to } \boldsymbol{v_{min}} \leq \boldsymbol{v_{cmd}} \leq \boldsymbol{v_{max}} \end{aligned} \tag{3.3}$$

where T is our horizon length, $\boldsymbol{x_{ref}}$ is our target trajectory, and J_x, J_s, J_d are the cost functions for this system.

Cost function J_x rewards following the target trajectory $\boldsymbol{x_{ref}}$ and is shown below:

$$J_x(\boldsymbol{x}, \boldsymbol{x_{ref}}) = \|\boldsymbol{x} - \boldsymbol{x_{ref}}\|^2 \tag{3.4}$$

J_s and J_d penalize collisions with static objects and dynamic objects respectively. They are defined as:

$$J_s(\boldsymbol{x}) = \sum_{i=1}^{N_{static}} \frac{s_s * Q_s}{1 + \exp(d_i^{static}/s_s)} \tag{3.5}$$

$$J_d(\boldsymbol{x}) = \sum_{i=1}^{N_{dynamic}} \frac{s_d * Q_d}{1 + \exp(d_i^{dynamic}/s_d)} \tag{3.6}$$

where d_i^{static} is the distance between the agent and the $i-th$ static obstacle, and $d_i^{dynamic}$ is the distance between the agent and the $i-th$ dynamic obstacle. For example, if the $i-th$ dynamic obstacle is another agent with radius r then $d_i^{dynamic} = \|\boldsymbol{x}(t) - \boldsymbol{x_i}(t)\|^2 - r$. $Q_s > 0$ and $Q_d > 0$ are weights that define how adverse our agents are to collision with static and dynamic obstacles respectively. A visualization of how these parameters affect the cost function is presented in Figure 3.9a.

Additionally, we define s_s and s_d as scale normalizing parameters that ensure the minima of $J_x + J_s$ and $J_x + J_d$ are greater than agent radius r in the single obstacle case. Essentially, this guarantees that the optimized trajectory of the agent never comes within radius r of an obstacle unless it is being "squished" by two obstacles.

To calculate s_s, we first find the positive minima x_{min} of $J_x + J_s$ in the worst case where $d_i^{static} = J_x$ (ie. the static obstacle and x_{ref} are in the same place), with $s = 1$. This gives us x_{min} as defined in Equation 3.7. Our scaling factor can then be defined as $s_s = \frac{r}{x_{min}}$, which sets the positive minima of $J_x + J_s$ to be r in the above situation. The calculation of s_d is symmetric, simply using Q_d instead of Q_s.

$$x_{min} = \ln\left(\frac{\sqrt{Q_s^2 - 4Q_s} + Q_s}{2} - 1\right) \tag{3.7}$$

The key benefit of these cost functions is that the optimal distance to obstacles of r is invariant to parameters Q_s and Q_d. This is in contrast to [32] which requires the parameters κ and Q to be re-tuned after changing agent radius r. This is because their cost functions do not have their minima at the collision point, but instead at some point before the collision that varies with κ, Q, and r, making them very difficult to tune. This is visualized in Figure 3.9

3.3.2.2 Implementation Details

Solving the integral in Equation 3.3 is computationally inefficient, therefore we split our calculations into discrete time steps. Specifically, the time horizon is set as $T = 500ms$ with $100ms$ timesteps.

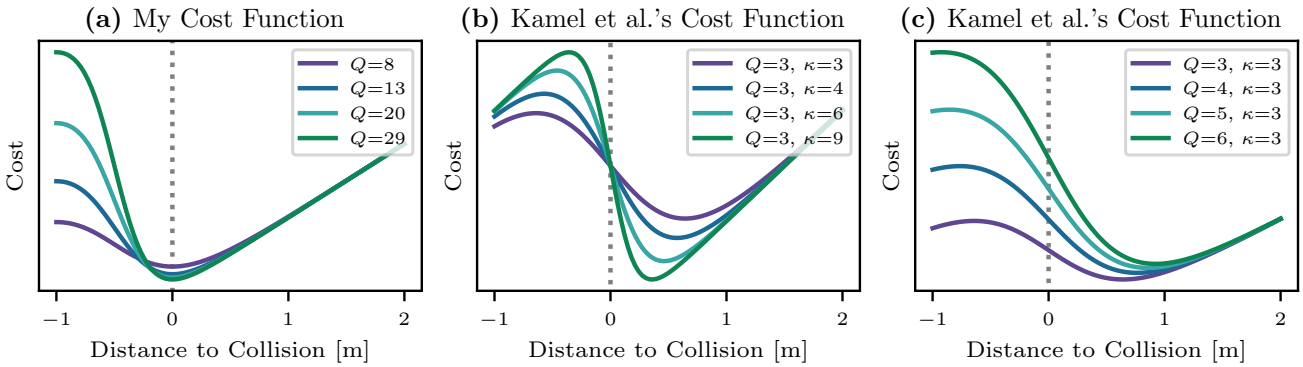


Figure 3.9: Comparison of my cost function (a) and Kamel et al.'s cost function (b), (c) when the obstacle and goal pose are at the same location. This demonstrates that my cost function's minima is always at the collision point and does not vary as the parameters are changed, as opposed to Kamel et al.'s function.

The *Sequential Least Squares Programming* minimization method is used due to its robustness and ability to constrain the optimization space, which we use to keep $\boldsymbol{v_{min}} \leq \boldsymbol{v_{cmd}} \leq \boldsymbol{v_{max}}$. We set parameters $Q_s = 8$ and $Q_d = 12$ to make the agent more adverse to approaching dynamic obstacles than static ones, as there is more uncertainty in the dynamic obstacle's location. To calculate the future locations of dynamic obstacles $\boldsymbol{x_i^{dynamic}}(t)$ we employ a simple constant velocity model.

Our SLAM system does not use a depth camera or LiDAR, and therefore only produces a sparse map. Consequently, we need to define the location of static obstacles manually. Both static obstacles and the goal pose are set using interactive ROS markers, allowing them to be changed "on the fly" using software such as RViz.

3.4 Central Management Interface

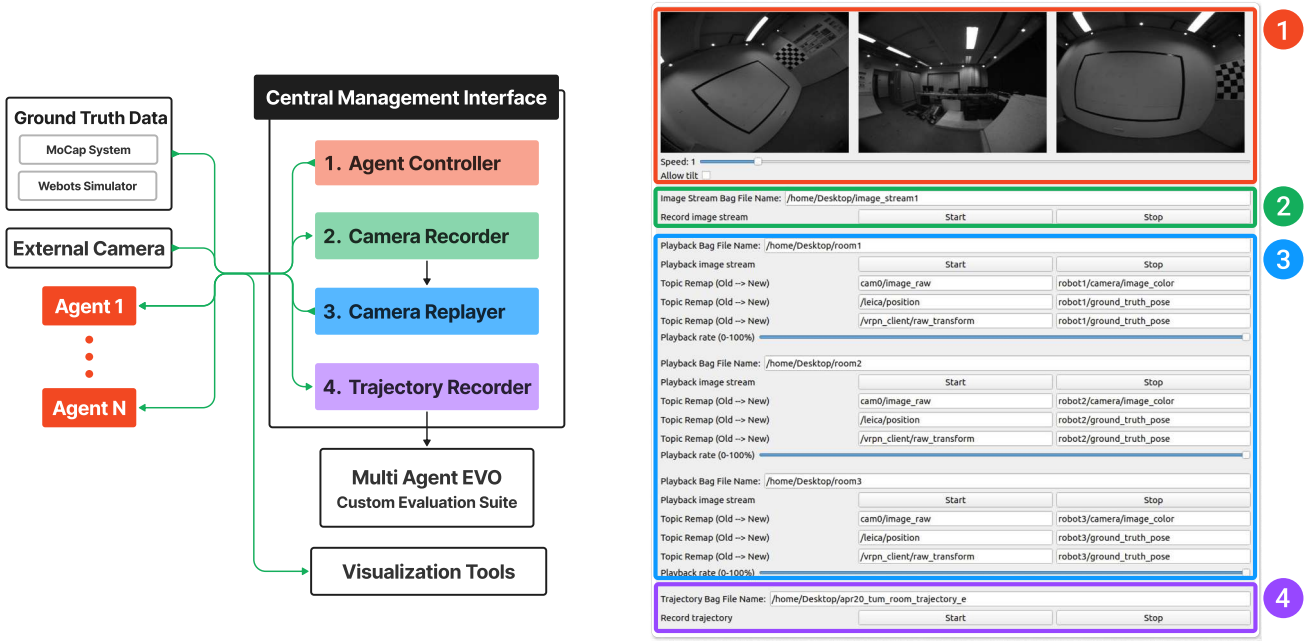
While my multi-agent system is fully decentralized, a significant amount of work was put into developing the supporting infrastructure needed to control, test, and evaluate this system.

This was part of a broader aim of this project to create more than just a research project that only runs on my machine. I have strived to develop a system that can be deployed in real-world use cases, and the significant infrastructure released alongside my SLAM system hopes to help achieve that goal.

The primary method of managing the distributed system is through the *Central Management Interface*, which can be used to:

1. **Manually control the agent's poses.**
   Users can click and drag on an image in area 1 to rotate the agent's camera and use the keyboard to move the agent around.
2. **Record camera data from the simulation software.**
3. **Play datasets back for testing and benchmarking purposes.**
   Additionally, there are settings to remap topics and control the speed of playback.

4. **Record trajectories generated by the SLAM system as well as ground truth data for later evaluation.**
   The recorded trajectory data can later be ingested by my custom evaluation library Multi-Agent Evo for analysis.



(a) Central Management Interface architectural diagram

(b) Central Management Interface

Figure 3.10: The central management interface, used for controlling agents, recording datasets, playing back datasets, and recording trajectories for later analysis.

The central management interface was built using the PyQt cross-platform user interface framework.

As a result of abstracting implementation details behind ROS topics, this central management interface is able to work seamlessly with both agents running in a simulator and agents running on real-world robots. This is true of every component of this project.

3.5 Custom Evaluation Suite – Multi-Agent EVO

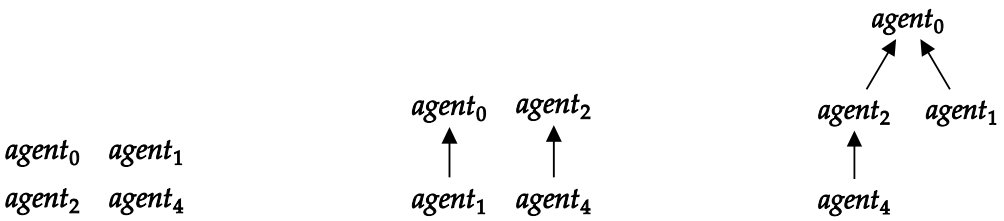
While there are several mature single-agent SLAM evaluation tools, I found there to be a complete lack of evaluation tools for multi-agent SLAM systems – to the best of my knowledge no papers have published their evaluation code. Continuing with my goal to develop and publish transferrable infrastructure alongside this project, I have created an open-source multi-agent SLAM evaluation tool: *Multi Agent EVO*, based on the popular single agent SLAM evaluation tool *EVO* [33].

Besides the simple data structure and data ingestion modifications needed to allow EVO to process multi-agent SLAM data, there is some additional nuance to evaluating data from multiple agents.

Initially, all agents are in separate reference frames until they explore an area previously seen by another agent, allowing them to merge their maps and share the same coordinate frame. We may also have cases where two independent groups of agents meet and merge maps, which requires multiple agents to simultaneously change coordinate frames. We also must note that these coordinate frames are part of the SIM(3) transformation group, which is composed of rotation, translation, and uniform scale in 3-dimensional space (scale being necessitated by the scale ambiguity of monocular visual SLAM).

Therefore, I have created a new data format to capture these changes in coordinate frames over time within our trajectory data, which Multi-Agent EVO is able to ingest. This allows us to properly compare the multi-agent SLAM trajectories to the ground truth data, giving us insights into how long it takes for agents to successfully merge maps, the accuracy of relative pose estimation, and much more.

Once agents m and n merge, they publish the translation $T_{m \to n}$ to the `si3m_transform` ROS topic. When we go to evaluate the trajectory error, Multi-Agent EVO ingests the `sim3_transform` data to build a directed acyclic graph (DAG) of all merged agents, as shown in Figure 3.11. This DAG can then be used to find if agents are merged with each other, and the transform between their coordinate frames. We then transform all the trajectories to be in $agent_0$'s coordinate frame for us to perform error analysis on the joint trajectory.



(a) All agents are initially unmerged.

(b) $agent_0$ and $agent_1$ merge, $agent_2$ and $agent_3$ merge.

(c) $agent_0$ and $agent_2$ merge.

Figure 3.11: Example of how coordinate frames are managed in Multi-Agent EVO. A path from $agent_m$ to $agent_n$ signifies that the agents are merged and there is an available translation from $agent_m$'s coordinate frame to $agent_n$'s.

This method also allows us to retroactively transform trajectories into $agnet_0$'s coordinate from before the agents merged, allowing us to analyze the error of the pre-merge estimated relative trajectories.

3.6 Simulation Environment

The Webots Simulation Software section in the Preparation chapter explains the motivation behind using a simulator. Essentially, leveraging simulations enables much faster iterations and the ability to test my system in a variety of environments. This section will focus on the details of integrating the simulation software into my system.

Webots is an open-source 3D robotics simulator with realistic rendering, which is essential for testing a visual SLAM system. Webots can not be built for the ARM Ubuntu VM I used for development, therefore it had to be run on MacOS, with data being streamed to the Ubuntu VM through a websocket bridge developed by Webots.

I then developed a ROS node that exposes the Webots cameras and agent controls as ROS topics, following the abstract robot interface specifications. This allowed my SLAM system to process the camera streams and the motion controller / central management interface to control the agent's movements.

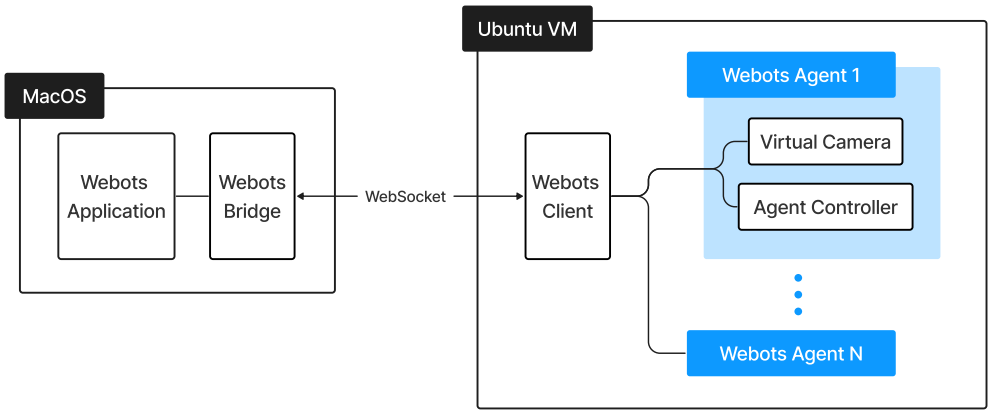


Figure 3.12: Simulation system architecture. The Webots application runs on MacOS and is exposed to the Ubuntu VM as ROS topics by the custom `Webots Client` ROS node.

3.7 Real World Implementation

After months of development within a simulator, my distributed SLAM system was able to seamlessly be deployed to a real-world multi-agent system. This was largely due to the decisions made during the preparation phase. For example, the choice to use ROS as a communication middleware allows the agents to easily communicate even when deployed on different physical devices. Another key choice was to abstract the implementation of nodes away behind the communication layer, as it allowed my system to run on the physical robots without making any changes to the SLAM system or motion controller nodes.

3.7.1 Cambridge RoboMaster Platform

I chose to deploy my system on the Cambridge RoboMaster, which is an omnidirectional robot platform with a NVidia Jetson Orin for compute. The only sensor used on the RoboMaster was the integrated Raspberry Pi HQ Camera, which provides a 1920x1080 image at 15hz.

This was an obvious platform to use as ROS drivers for the camera and wheels had already been developed and a ground-based platform allows for easier testing.

As an aside, **I was an author of the paper *The Cambridge RoboMaster: An Agile Multi-Robot Research Platform***, which has been submitted to the 17th International Symposium on Distributed Autonomous Robotic Systems. My distributed SLAM system running the multi-agent collision avoidance motion controller was included as one of the experiments analyzed in the paper.

3.7.2 Deploying with Docker

My SLAM system is deployed on the RoboMasters in Docker containers. This allows my code to be compartmentalized from other projects being developed on the RoboMasters, as they are actively used by numerous researchers in the Prorok Lab. Additionally, I have set up a continuous integration process on GitHub to cross-compile a Docker container for the RoboMasters on every code push, which can then be pulled to the RoboMasters. This is a very useful feature, as building my full codebase can take upwards of 20 minutes on the NVidia Jetson Orins.

3.7.3 OptiTrack Motion Capture System

The ground truth for my real-world experiments are provided by the Prorok Lab's OptiTrack Motion Capture System, which advertises accuracies of $\leq$0.3mm at rates of 180hz [34]. The motion capture system publishes the real-time poses of the tracked objects to a ROS topic which can be recorded for later analysis.

3.7.4 Raspberry Pi Video Publisher

This dissertation also presents the Raspberry Pi Video Publisher, an independent piece of infrastructure which I have developed both the hardware and software for. This platform publishes real-time camera data to a ROS topic and can be tracked by the lab's motion capture system. The two primary use cases I have used this platform for are (1) dataset generation and (2) real-time AR visualization of 3D data (subsection 3.7.5).

As explained in the Continuous Integration / Continuous Deployment section, the software for this platform is entirely dockerized and a CI/CD pipeline has been implemented to automatically build and deploy the software to all Raspberry Pi video publishers when new code is pushed to the GitHub repository. This is particularly useful, as it is designed as a plug-and-play system that starts streaming video data as soon as it is turned on.

The hardware for this platform consists of a Raspberry Pi 4b, Raspberry Pi Camera v2, 3D printed frame, and motion capture markers. The 3D-printed frame securely mounts the components together, allowing the system to be carried around or mounted to a tripod.

3.7.5 Augmented Reality Visualization

The Raspberry Pi Video Publisher platform is tracked by the motion capture system, allowing us to project 3D visualization data onto the captured video. This can be used to visualize the SLAM system's keyframe and map point locations in the real world and to view how the SLAM system's predicted trajectory aligns with reality.

To overlay our SLAM system's data on the video, we must first align the SLAM system and motion capture systems' coordinate frames. This needs to be done on every run, as monocular SLAM systems have an arbitrary scale. Therefore, we use the Kabsch-Umeyama Algorithm to align the trajectories captured by the motion capture system and SLAM system after enough

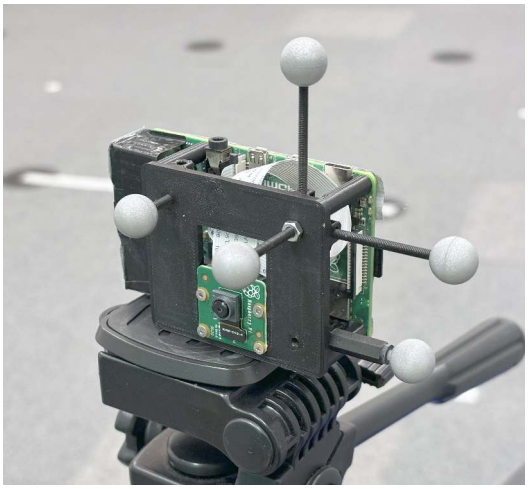


Figure 3.13: Custom-built Raspberry Pi Video Publisher mounted on a tripod.

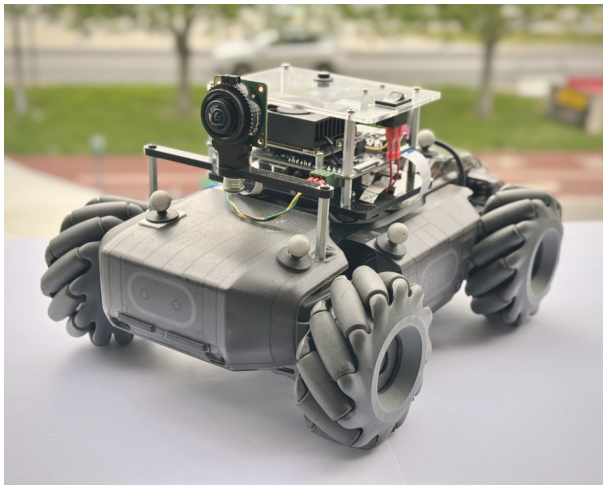


Figure 3.14: Cambridge RoboMaster platform with NVidia Jetson Orin and Raspberry Pi HQ Camera.

data has been collected to create a successful alignment. We can then use Foxglove Studio to draw our 3D markers and project them on top of the tracked camera's video stream, a snapshot of which is displayed in Figure 3.15.

To my knowledge, this system is the first of its kind to be used in such a project. Not only is it visually impressive, but it also can be used for further analysis and understanding. For example, the camera can be used to understand which features the SLAM system is tracking and how well their predicted location aligns with reality. Additionally, it can be used to analyze the SLAM system's predicted trajectory – this visualization helped me identify that my collision avoidance demo was being severely impacted by latency spikes which I proceeded to fix.

Figure 3.15: Features tracked by the RoboMaster overlayed over an "external perspective" handheld video from my Raspberry Pi Video Publisher. Note that the features are not being generated by the Raspberry Pi video itself, but instead the features generated by the RoboMaster are reprojected onto the Raspberry Pi video as it moves through space. This is made more obvious in the video demonstration of this feature: TODO add video.

This visualization makes it clear that my SLAM system is tracking the corners of boxes, letters on the Prorok Lab banner, and the floor vents.

3.8 Repository Overview

TODO: complete

This project consists of three separate repositories:

The `part_ii_project` repository contains the core SLAM system and all the infrastructure needed to run it. This includes five ROS packages: `orb_slam3_ros`, `motion_controller`, `webots_sim`, `controller`, and `interfaces`, which all follow the standard ROS package structure. `orb_slam3_ros` and `interfaces` are written in C++ and built using CMake, while the other packages are written in Python.

```
part_ii_project/
├── src/ ..... +1,000 -1,000
│   ├── orb_slam3_ros/ ..... Core SLAM package +1,000 -1,000
│   │   ├── configs/ ..... Config for different camera systems
│   │   ├── src/ ..... Source code for my SLAM system +1,351
│   │   ├── orb_slam3/ ..... Heavily modified ORB-SLAM3 codebase +1,000 -1,000
│   │   └── ...
│   ├── motion_controller/ ..... Motion controller package
│   │   ├── motion_controller/ ..... +615
│   │   │   ├── collision_avoidance.py ..... +142
│   │   │   ├── follow_the_leader.py ..... +78
│   │   │   ├── helpers/ ..... +325
│   │   │   └── ...
│   │   ├── webots_sim/ ..... +276
│   │   │   ├── launch/robot_launch.py ..... +42
│   │   │   ├── webots_sim/robot_driver.py ..... +58
│   │   │   ├── worlds/ ..... Simulation environments
│   │   │   └── ...
│   ├── controller/ ..... Centeral management interface package +442
│   ├── evaluation/ ..... Jupyter Notebooks for evaluation +10
│   ├── interfaces/ ..... Custom ROS message type definitions +10
│   └── tools/ ..... Misc. scripts +10
├── .github/workflows/config.yml ..... CI/CD configuration +73
├── Dockerfile ..... Main dockerfile for the project +20
├── nvidia_jetson.Dockerfile ..... Dockerfile for the NVidia Jetson +30
└── ...
```

The `rpi_ros2_camera_publisher` repository contains the code and CI/CD configuration for the Raspberry Pi Video Publisher.

```
rpi_ros2_camera_publisher/ ..... +190
├── src/
│   ├── send_images.py ..... +132
├── .github/workflows/config.yml ..... CI/CD configuration +36
├── Dockerfile ..... Main dockerfile for the project +10
├── docker-rpi-cam.service ..... Systemctl service file +11
└── ...
```

The `multi_agent_evo` repository is a fork of the EVO library and is used to evaluate multi-agent SLAM systems. The changes made to EVO are shown below.

```
multi_agent_evo/ ..... +317 -21
├── evo/
│   ├── core/ ..... +126 -5
│   ├── tools/ ..... +189 -16
└── ...
```

Chapter 4

Evaluation

4.1 Review of Success Criteria

My project has met all success criteria as outlined below:

1a. Multiple agents will be able to localize themselves within a world using purely visual data.
My system is the first decentralized SLAM system capable of operating using only monocular camera data. This removes the need for large and expensive stereo cameras, LiDAR, or RGBD sensors.

1b. Agents will be capable of communicating with each other to build a shared understanding of the world.
Agents use ROS to perform decentralized and reliable communication with each other and are able to merge maps and share keyframes to build a shared world and provide relative localization.

1c. Agents will be able to act independently, failing gracefully if they lose communication with their peers.
Agents fall back to performing single-agent visual SLAM when communication with their peers is degraded or lost. Once communication is regained, the agents share their unsent map data.

2. Evaluate the capabilities of the system compared to a comparable single-agent system.
This is performed in the Benchmarking section. I go beyond the original success criteria by also comparing my system to comparable multi-agent systems, demonstrating my system's superior performance.

After discussions with my supervisor, we decided to not pursue my original project extensions, as they had either implicitly been achieved during development or were no longer very relevant to my project. Extension 1 and 2 were to "enable agents to intelligently share information with their peers" and "distribute calculations among the different agents", which were already achieved by the `external keyframe inserter` and `external map merger` modules. Extension 3 was to create and test map compression algorithms, however, preliminary testing revealed that data transfer rates were already acceptable. Finally, extension 4 involved adding additional sensors as inputs to the system, however, this has already been achieved by other systems.

Although there was significant potential to further develop each of these extensions, the primary reason for our decision to pivot towards deploying my system in the real world was our belief

that it would substantially enhance its credibility. Demonstrating the ability to operate my system in real-time on a distributed system proves its applicability in real-world scenarios and shows that my system can adapt to the noisy and unpredictable conditions of the real world.

Additionally, deploying my system on real robots provided evidence of how software engineering decisions such as using ROS and Docker allowed my system to be reproducible on a variety of hardware, instead of just my laptop.

4.2 Benchmarking

This section will benchmark the performance of my system when run on industry-standard visual SLAM datasets. All evaluations are run using only monocular camera data.

The Central Management Interface is used to stream the dataset to the agents and my library *Multi-Agent EVO* is used to provide the Absolute Trajectory Error (ATE) metric, as defined by [35].

4.2.1 EuRoC Machine Hall

The EuRoC Machine Hall dataset [23] is one of the most widely used visual SLAM datasets, providing a 752x480 20fps video feed and millimeter-level ground truth data at 20hz. The camera rig is attached to a Micro Aerial Vehicle (MAV) which flies through a machine room approximately 15x15 meters in size. In line with other research, we simulate a multi-agent dataset by running the Machine Hall 01-03 scenarios in parallel on three agents.

Figure 4.1 presents the RMS absolute trajectory error and total system data transfer of my distributed system running this dataset. The median RMS ATE is 5.9cm over the 279 meter total trajectory length, demonstrating my system's very impressive accuracy. The ATEs and data transfer rates have a 1.4cm and 0.11 MB/s spread around the median respectively across 5 trials, demonstrating the repeatability and robustness of my system.

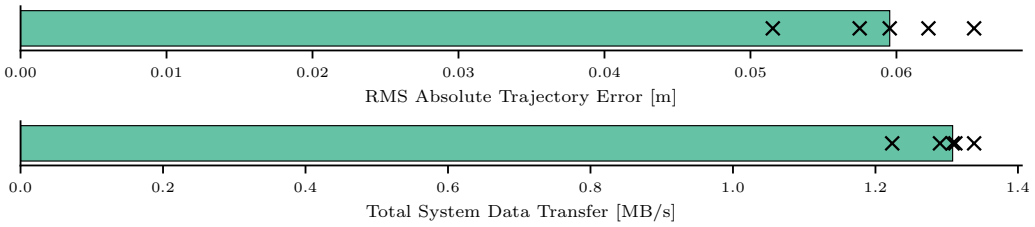


Figure 4.1: Plot the RMS ATE and Total Data Transfer rate of running the EuRoC Machine Hall 01-03 dataset 5 times on my system.

To further analyze the characteristics of my SLAM system, we focus on an individual trial. Figure 4.4 displays the full trajectory of the three agents compared to the ground truth, demonstrating my system's high global accuracy and relative positioning. Figure 4.2 plots the ATE as the trial progresses. It is clear that my system is performing SLAM as opposed to simple visual odometry as the ATE returns to the baseline at the end of the trial when the agents return back to their starting position, relocalizing itself within the previously built map.

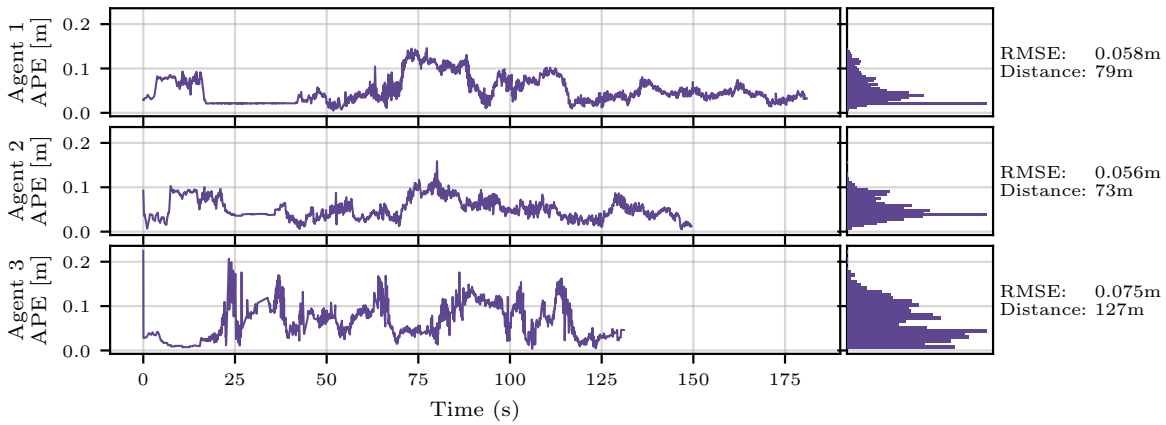


Figure 4.2: Plot of my system's absolute trajectory error (ATE) with respect to the ground truth when running the EuRoC Machine Hall 01-03 scenarios in parallel on three agents.

RMSE represents the root mean square of the ATE.

We now analyze the network usage presented in Figure 4.3. Initially, the agents send bag of word information before quickly detecting a merge opportunity. $agnet_1$ proceeds to send its full map to $agnet_2$, which can be seen in the large initial spike in network bandwidth. The agents successfully merge, and begin exchanging keyframes. The rate of keyframe data being sent rises and falls depending on how much new area the agent is exploring.

Along with sending keyframes, the agents sporadically send map points to refine their map alignment. This occurs less frequently the longer system runs due to the additive increase multiplicative decrease method used to schedule map alignments, described in the Map Alignment Refiner section. However, the size of the messages also grows over time due to the larger number of tracked map points.

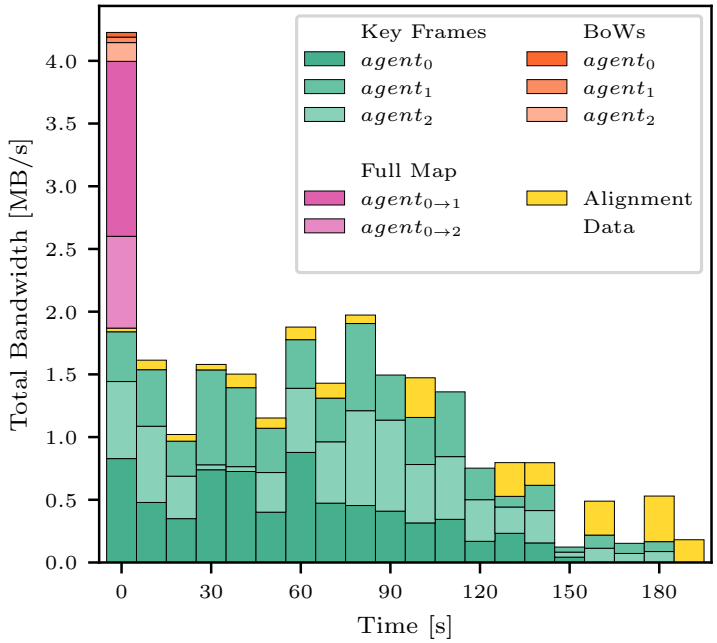
The data exchanged between agents largely consists of new keyframes, and further work should be dedicated to optimizing the serialized representation of keyframes.

TODO: add more figures of the visualizations of the maps built by the system by the end of the sessions

4.2.2 TUM-VI Rooms

The TUM visual-inertial dataset [24] consists of handheld fisheye 512x512 video with ground truth data. As before, we use only monocular visual data and combine the Room 1-3 sessions to simulate a multi-agent dataset. The "room" environment is used for this evaluation, which is a 3x3 meter motion capture lab with plain flat walls with some posters hung up. There is less texture in this environment than the machine hall, making visual-only SLAM difficult. As seen in the full trajectory estimate given in Figure 4.5, parts of the room are revisited dozens of times by different agents, allowing us to evaluate our system's ability to relocalize agents within previously mapped environments.

Figure 4.7 shows that the RMS ATE is very tightly clustered around the median of 6.95cm, with one outlier caused by an agent in that trial having a slightly unoptimal map merge. The tight clustering is most likely due to the small environment with areas being revisited many times, allowing the shared map to converge on a very similar solution each time. The lower



(a) Total system data over time, segregated by message type

		KB	Avg. KB/s
Key Frames	$agent_0$	69,971	351.9
	$agent_1$	63,908	321.4
	$agent_2$	65,164	327.7
BoWs	$agent_0$	371	1.9
	$agent_1$	437	2.2
	$agent_2$	1,496	7.5
Full Map	$agent_{0\to1}$	13,953	70.2
	$agent_{0\to2}$	7,319	36.8
Alignment Data		22,560	113.6
Total Data		**245,218**	**1,233.1**

(b) Total system data by message type

	Sent		Received	
	KB	Avg. KB/s	KB	Avg. KB/s
$agent_0$	**114,585**	**576.2**	**131,005**	**658.8**
$agent_1$	**64,345**	**323.6**	**173,554**	**872.8**
$agent_2$	**66,660**	**335.2**	**164,606**	**827.7**

(c) Data by agent

Figure 4.3: Bandwidth used by the EuRoC Machine Hall 01-03 scenarios running on my SLAM system.

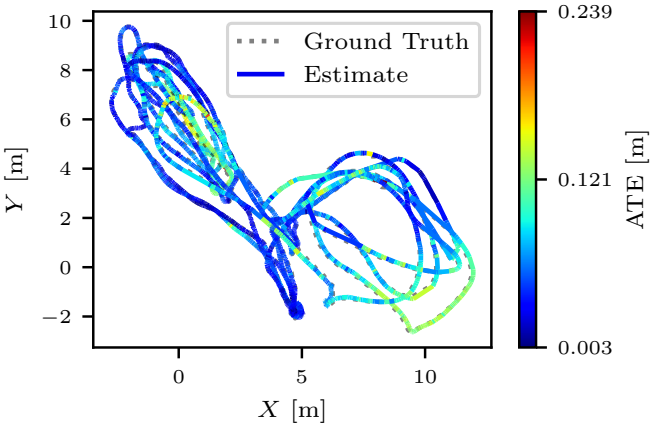


Figure 4.4: EuRoC Machine Hall 01-03 estimated trajectory and ground truth. TODO: split up by agent somehow?

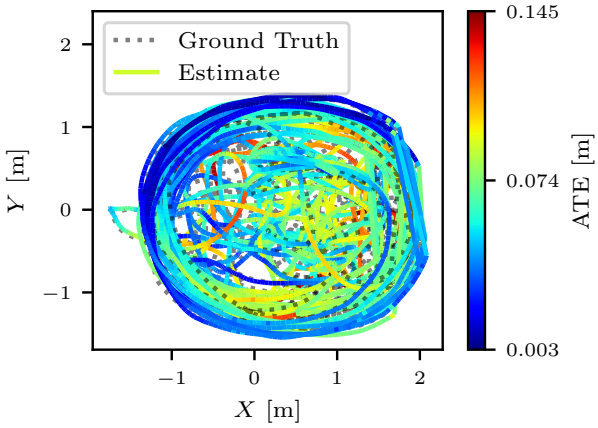


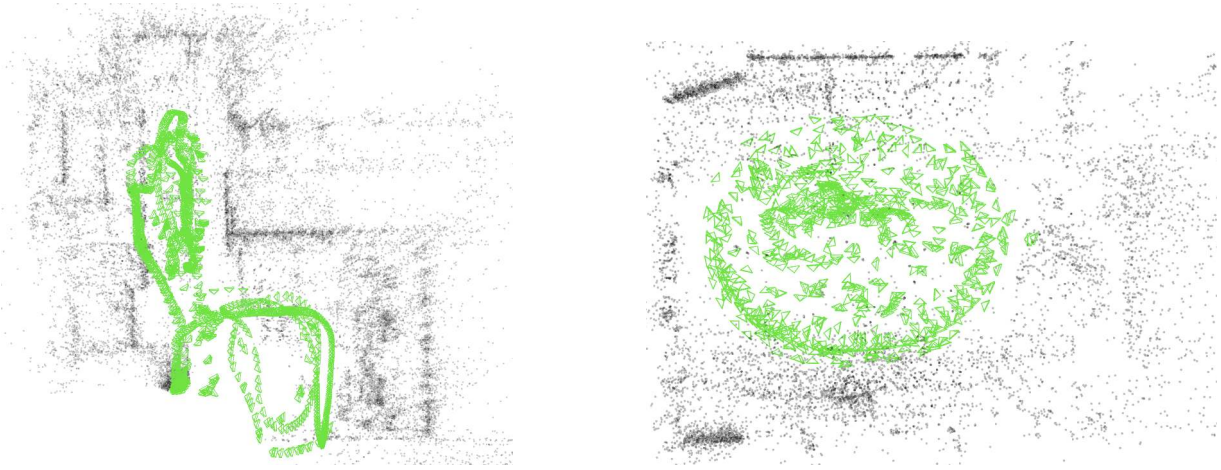
Figure 4.5: TUM-VI Rooms 01-03 estimated trajectory and ground truth.

median data transfer rate of 0.84 MB/s is also due to the small environment, resulting in the agents having to share less information.

Once again, we focus on an individual trial to further evaluate my system. Figure 4.8 shows that there is no long-term error built up, demonstrating that my system is successfully localizing agents within the shared map and performing long-term map point association.

Figure 4.9 presents the data transfer breakdown of a single trial, and yields similar conclusions to that of the EuRoC Machine Hall dataset. The only difference to note is the smaller key frame and alignment data transfer rate due to the smaller environment.

Too many graphs? Kinda seems a bit repetitive



(a) EuRoC Machine Hall 01-03 **(b)** TUM-VI Rooms 1-3

Figure 4.6: Sparse maps built by my distributed SLAM system running industry-standard datasets.

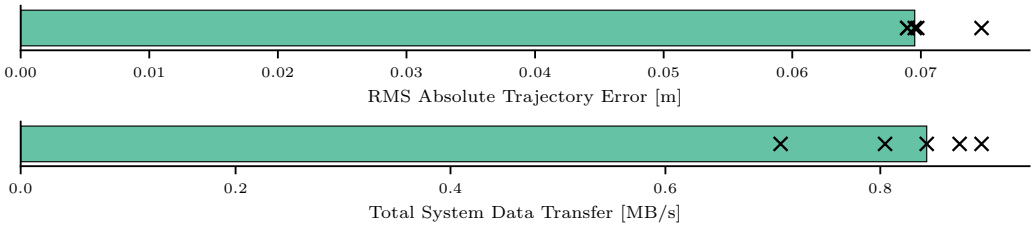


Figure 4.7: Plot the RMS ATE and Total Data Transfer rate of running the EuRoC Machine Hall 01-03 dataset 5 times on my system.

4.3 Comparison to Related Work

As discussed in the Relevant Work section, my system is the first decentralized monocular vision SLAM system available. This makes direct comparisons with other multi-agent SLAM systems difficult, as they rely on more accurate but bulky sensors such as stereo cameras and LiDAR. However, this section still attempts to compare my system to the latest research in the field to give context to its performance relative to other state-of-the-art systems.

We are only able to compare results using the EuRoC dataset as it is the only public dataset evaluated by CCM-SLAM. Ideally, I would run more datasets on CCM-SLAM system, however, I encountered many issues when trying to build the codebase published alongside their paper. This reinforces the well-known reproducibility issues within the field of visual SLAM [2], which I have attempted to mitigate in my own project by publishing pre-compiled docker containers alongside my code. **My SLAM system is not specifically tweaked to perform well in the EuRoC dataset and we see similarly low RMS ATE values in both the TUM Rooms dataset and real-world evaluations.**

4.3.1 CCM-SLAM

Centralized Collaborative Monocular SLAM (CCM-SLAM) (2019) from the ETH Zurich Vision for Robotics Lab [11] is the most comparable multi-agent SLAM system in recent literature and is well-cited within the field. Their approach also uses purely monocular vision with ORB-SLAM to perform the front-end tracking and local mapping stages of the SLAM pipeline, with a

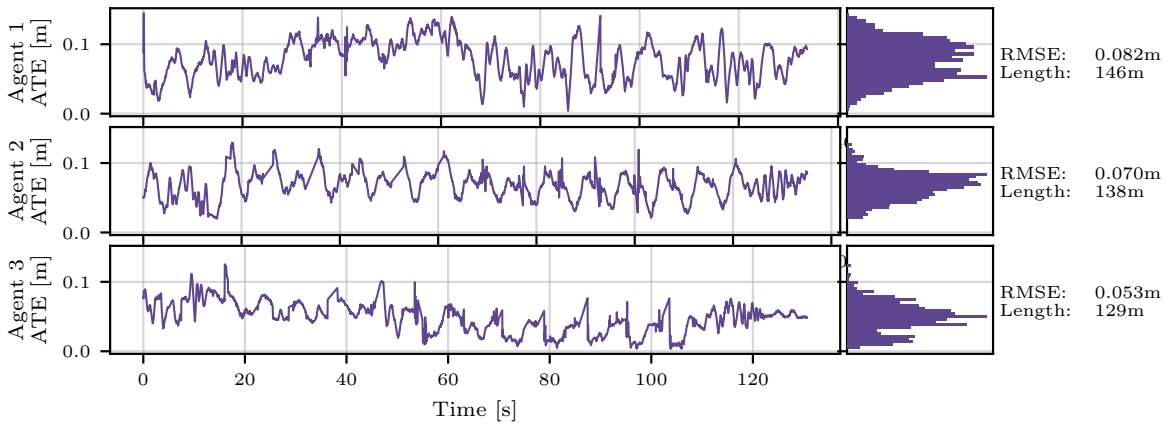
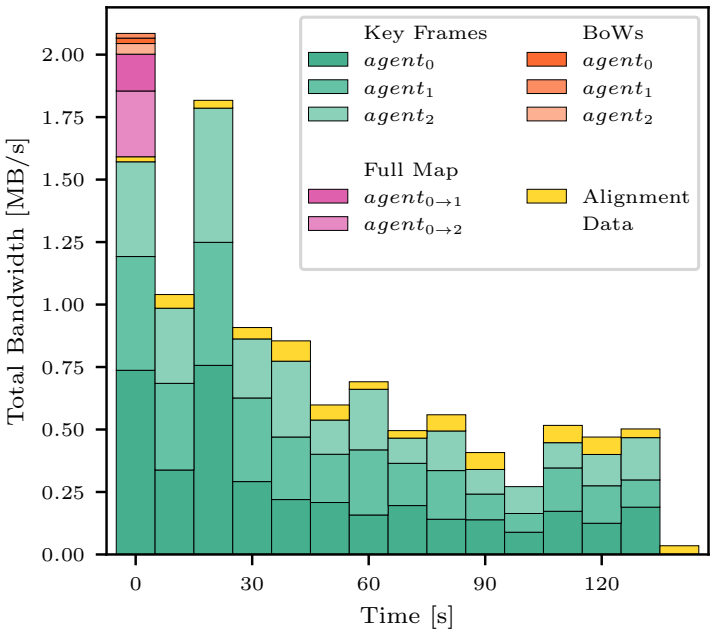


Figure 4.8: Plot of my system's average trajectory error (ATE) with respect to the ground truth when running the TUM-VI Room 1-3 scenarios in parallel on three agents.



(a) Total system data over time

		KB	Avg. KB/s
Key Frames	$agent_0$	37,599	264.9
	$agent_1$	33,066	233.0
	$agent_2$	29,963	211.1
BoWs	$agent_0$	217	1.5
	$agent_1$	188	1.3
	$agent_2$	426	3.0
Full Map	$agent_{0\to1}$	1,472	10.4
	$agent_{0\to2}$	2,631	18.5
Alignment Data		6,947	49.0
Total Data		**112,511**	**792.8**

(b) Total system data by message type, segregated by message type

	Sent		Received	
	KB	Avg. KB/s	KB	Avg. KB/s
$agent_0$	**49,083**	**345.9**	**63,644**	**448.5**
$agent_1$	**33,254**	**234.3**	**76,624**	**539.9**
$agent_2$	**30,389**	**214.1**	**80,649**	**568.3**

(c) Data by agent

Figure 4.9: Bandwidth used by the TUM-VI Rooms 01-03 scenarios running on my SLAM system.

custom backend to integrate data from multiple agents into the shared map. To my knowledge, it is the most recent monocular vision-capable multi-agent SLAM system.

The results of CCM-SLAM and my system running the EuRoC Machine Hall 01-03 dataset are presented in Figure 4.10. **It is important to note that CCM-SLAM is a centralized system, significantly simplifying their multi-agent SLAM problem compared to my distributed system.** Nevertheless, my system is able to outperform CCM-SLAM in terms of RMS ATE, demonstrating its competitiveness. My system's total data transferred is 32% higher than CCM-SLAM, however, this could easily be reduced as I have made no attempt to optimize the data transfer efficiency of my system.

4.3.2 VINS-Mono Multisession SLAM

VINS-Mono (2018) [36] is a popular open-source single-agent monocular vision SLAM system. While it is a single-agent system, it has multisession capabilities. This allows multiple datasets to be played consecutively, with each session building upon the map built by the previous sessions (eg. session 2 is initialized in the map generated after running session 1). This is an easier problem than multi-agent SLAM, where the sessions are run in parallel and the agents have to share their maps in real time. As presented in Figure 4.10, my system still outperforms VINS-Mono multisession.

	My system	CCM-SLAM[1]	VINS-Mono[2]
RMS ATE [m][3]	**0.059**	0.077	0.074

Figure 4.10: Comparison of my SLAM system to comparable state-of-the-art multi-agent systems running the EuRoC Machine Hall 01-03 dataset with only monocular images. doesnt feel like a very strong comparison since we can only compare with one dataset :(

[1]CCM-SLAM is a centralized system, as opposed to my decentralized system.
[2]VINS-Mono is a single-agent SLAM system run in multisession mode.
[3]RMS ATE is taken as the median of 5 trials. Results of external systems are taken from [11].

4.3.3 Comparison to Single-Agent SLAM Systems

Along with providing relative positioning, we would hypothesize a multi-agent SLAM system to have higher accuracy than a comparable single-agent system. However, the reality is far more complex than this simple assumption. Figure 4.11 presents my multi-agent system's errors compared to ORB-SLAM3 running the datasets individually. ORB-SLAM3 is used to perform the tracking and local mapping in my multi-agent system and is widely regarded as the most accurate single-agent SLAM system currently available so there is no surprise that it performs very well.

An observation is that most dataset sessions have only a marginal improvement in error over the single-agent system, which I hypothesize is the result of the varying ability among agents to utilize the shared map. For example, the TUM-VI Room dataset has very similar performance in both the single-agent and multi-agent systems since the environment is very small and areas of the map are revisited dozens of times in one session. This reduces the benefit of a multi-agent system as a single-agent system already observes more than enough data to build an accurate map of the world, making the extra data provided by the additional agents in a multi-agent system redundant.

We see a significant improvement in error for the EuRoC Machine Hall 02 session, likely due to the 02 agent closely following behind the 01 agent for the majority of the session. The Machine Hall 03 session spends most of its time exploring a separate part of the environment from the other sessions, so a similar performance is to be expected. Surprisingly, the Machine Hall 01 session's error is not reduced – this could perhaps be attributed to it "forging the way" for the 02 session most of the time. improve explanations?? they're a bit handwavy?

On a broader note, many researchers believe single-agent SLAM to be a solved problem. This is reflected by the results seen here, where the single-agent system is able to give incredibly low

	My system[1]	ORB-SLAM3[2]
EuRoC MH 01	0.051	**0.049**
EuRoC MH 02	**0.048**	0.099
EuRoC MH 03	**0.058**	0.062
TUM-VI Room 1	0.072	**0.071**
TUM-VI Room 2	**0.048**	0.050
TUM-VI Room 3	**0.041**	0.042

Figure 4.11: Comparison of individual agent errors from my multi-agent SLAM system and the single-agent ORB-SLAM system. All results are the median of 5 runs.

[1]Trajectories are aligned with the ground truth on a per-agent basis so we can compare each individual agent's performance in isolation.
[2]Results are generated by running ORB-SLAM3 on my device with its default settings for the EuRoC dataset.

trajectory errors that even multi-agent systems with far more data struggle to improve upon. **This is not something specific to my system – many other multi-agent systems do not have significantly higher accuracies than single-agent systems.**

Therefore, I believe that the primary benefits of a multi-agent system are in its ability to provide relative positioning for path planning and collision avoidance and to build a shared map of the environment which is useful in applications such as search and rescue or cave mapping.

TODO visualizes the collaborative shared map built by my system compared to the individual maps from the single-agent ORB-SLAM3. There is clear deep integration between the different agents' map data in my collaborative map, demonstrating that the agents are truly collaborating and welding their maps together. The multi-agent system is also able to build a full map of the environment, showing the advantage of collaborative SLAM systems.

4.4 Real World Experiments

Deploying my system on physical robots demonstrates its ability to be run in real-time within the computational, bandwidth, and latency constraints of real-world systems. Furthermore, it demonstrates the robustness of my development framework which allowed seamless migration from local testing using simulations to a real distributed system using live camera feeds with no changes to my code base.

The details of the real-world setup are given in the Real World Implementation section. In summary, the Cambridge RoboMasters platform was used within the Prorok Lab motion capture lab.

4.4.1 Multi-Agent Collision Avoidance

In this experiment, we showcase the NMPC collision avoidance motion controller working in conjunction with the SLAM system to avoid collisions with both static and dynamic obstacles. Real-time relative position data from the SLAM system is fed to the NMPC collision avoidance

motion controller which controls the robot's velocity accordingly. The SLAM system is running locally on each Cambridge RoboMaster, with communication facilitated by a router in the lab.

Figure 4.12 tests the system in an intersection environment, where two robots (traveling along the horizontal and vertical axis) would normally collide. The shared map generated by the two agents allows them to localize each other even when their views do not overlap and they can not see the other agent, showcasing the versatility and wide-ranging use cases of my distributed SLAM system. Additionally, this demonstrates the real-time capabilities of my SLAM system, allowing the robots to react fast enough to avoid collisions in dynamic environments.

Out of the four consecutive trials run in this environment, there were zero collisions between the two agents, and Figure 4.13 shows that the distance between agents never went below the collision threshold of 0.55 meters. Figure 4.14 shows the ATE of the 4 trials, with a RMS ATE of only 0.074 meters over the 50.6-meter long trajectory.

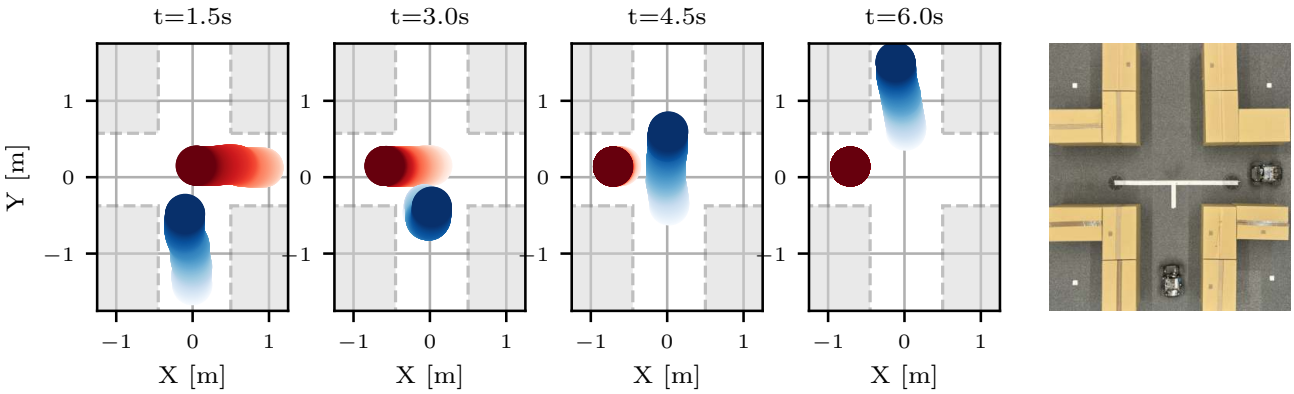


Figure 4.12: Demonstration of multi-agent collision avoidance, facilitated by my distributed SLAM system running locally on the Cambridge RoboMasters. Two robots are set 90° to each other in an intersection environment, with no direct view of the other robot and little visual overlap (right). The agent travelling along the Y axis is given a goal pose on the other side of the intersection, and successfully avoids a collision when the agent travelling along the X axis is pushed through the intersection. The trajectories generated by the SLAM system are presented on the left charts.

add "evaluation" of raspberry pi augmented reality visualization? just attach a video?

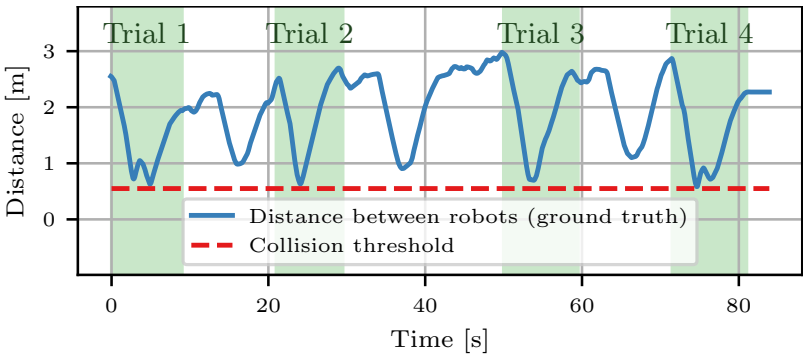


Figure 4.13: Plot of ground truth distance between the two robots throughout all four collision avoidance trials conducted in the intersection environment. The dips between trials are the robots' positions being reset. Figure 4.12 visualizes Trial 4 in more detail.

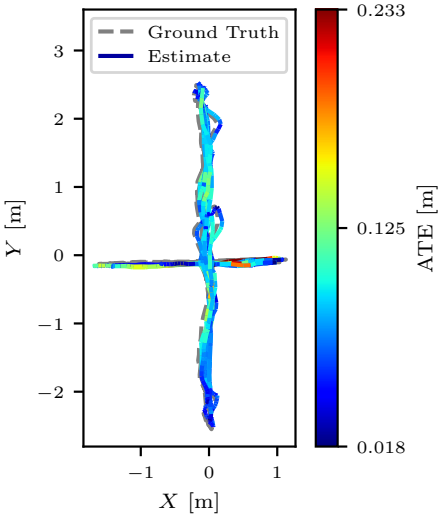


Figure 4.14: Collision avoidance estimated trajectory and ground truth. The RMS ATE is 0.074 meters.

Chapter 5

Conclusions

5.1 Future Work

TODO – any suggestions?

5.2 Lessons Learned

The most important lesson learnt from this project is that cutting edge research does not equate to well written or documented code. I committed to using the single agent ORB-SLAM3 system as the foundation for my project early on, however I quickly discovered that it was extremely hard to build upon their codebase. Large chunks of code were commented out without explination, everything was defined as instance variables making dataflow hard to decypher, and many function were named, for example, `localMapping()`, `localMapping2()`, with different parts of the code arbitrarliy choosing any one of the functions to use. In addition, there is simply a large amount of inherit complexity within the codebase as it is arguably the most advanced single-agent SLAM system avaiable, having been in development for over 6 years, and runs on 4 seperate threads.

ORB-SLAM also did not compile out-of-the-box, requiring me to make modifications to both the makefiles and codebase. Due to the above complexities as well as it being my first time using C++, it took almost month to get ORB-SLAM running, and many more before I began making good progress on the distributed aspect of the project – far longer than I had expected. In summary, I should not have assumed that research code would be easy to compile and build upon, even if it was one of the most well cited and performant systems in the field.

5.3 Reflection

Four years ago when I was appling to Cambridge, I wrote in my personal statement that I was facinated by computer vision – specifically visual SLAM. I find it rather fitting that I now conclude my degree by developing a novel distributed visual SLAM system that performs better than comparable state of the art systems – something I could have only dreamt of when writing my personal statement all those years ago.

Overall, I am very satisfied with the results achieved by my system. I knew from the beginning that I did not want to spend a year creating something worse than what was already out there, and I am proud to have acheived that.

Bibliography

[1] Carlos Campos et al. "ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial and Multi-Map SLAM". In: *IEEE Transactions on Robotics* 37.6 (2021), pp. 1874–1890.

[2] Dinar Sharafutdinov et al. "Comparison of modern open-source visual SLAM approaches". In: *CoRR* abs/2108.01654 (2021). arXiv: 2108.01654. URL: https://arxiv.org/abs/2108.01654.

[3] Xiang Gao and Tao Zhang. *Introduction to visual SLAM: from theory to practice*. Springer Nature, 2021.

[4] Xiang Gao et al. *14 Lectures on Visual SLAM: From Theory to Practice*. Publishing House of Electronics Industry, 2017.

[5] Hugh Durrant-Whyte and Tim Bailey. "Simultaneous localization and mapping: part I". In: *IEEE robotics & automation magazine* 13.2 (2006), pp. 99–110.

[6] Takafumi Taketomi, Hideaki Uchiyama, and Sei Ikeda. "Visual SLAM algorithms: A survey from 2010 to 2016". In: *IPSJ transactions on computer vision and applications* 9 (2017), pp. 1–11.

[7] Khalid Yousif, Alireza Bab-Hadiashar, and Reza Hoseinnezhad. "An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics". In: *Intelligent Industrial Systems* 1 (Nov. 2015). DOI: 10.1007/s40903-015-0032-7.

[8] Andreas Geiger et al. "Vision meets Robotics: The KITTI Dataset". In: *International Journal of Robotics Research (IJRR)* (2013).

[9] Ethan Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[10] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, ISBN: 0521540518, 2004.

[11] Patrik Schmuck and Margarita Chli. "CCM-SLAM: Robust and efficient centralized collaborative monocular simultaneous localization and mapping for robotic teams". In: *Journal of Field Robotics* 36.4 (2019), pp. 763–781.

[12] Patrik Schmuck et al. *COVINS: Visual-Inertial SLAM for Centralized Collaboration*. 2021. arXiv: 2108.05756 [cs.RO].

[13] Xin Zhou et al. "Swarm of micro flying robots in the wild". In: *Science Robotics* 7.66 (2022), eabm5954. DOI: 10.1126/scirobotics.abm5954. eprint: https://www.science.org/doi/pdf/10.1126/scirobotics.abm5954. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm5954.

[14] Suvam Patra et al. "EGO-SLAM: A Robust Monocular SLAM for Egocentric Videos". In: *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2019, pp. 31–40. DOI: 10.1109/WACV.2019.00011.

[15] Pengxiang Zhu et al. "Distributed Visual-Inertial Cooperative Localization". In: *CoRR* abs/2103.12770 (2021). arXiv: 2103.12770. URL: https://arxiv.org/abs/2103.12770.

[16] Hao Xu et al. "D^2SLAM: Decentralized and Distributed Collaborative Visual-inertial SLAM System for Aerial Swarm". In: *arXiv preprint arXiv:2211.01538* (2022).

[17] Yulun Tian et al. "Kimera-Multi: Robust, Distributed, Dense Metric-Semantic SLAM for Multi-Robot Systems". In: *IEEE Transactions on Robotics* 38.4 (2022), pp. 2022–2038. DOI: 10.1109/TRO.2021.3137751.

[18] Pierre-Yves Lajoie and Giovanni Beltrame. "Swarm-SLAM: Sparse Decentralized Collaborative Simultaneous Localization and Mapping Framework for Multi-Robot Systems". In: *IEEE Robotics and Automation Letters* 9.1 (2024), pp. 475–482. DOI: 10.1109/LRA.2023.3333742.

[19] P. Lajoie et al. "DOOR-SLAM: Distributed, Online, and Outlier Resilient SLAM for Robotic Teams". In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 1656–1663.

[20] Pierre-Yves Lajoie and Giovanni Beltrame. "Swarm-SLAM: Sparse Decentralized Collaborative Simultaneous Localization and Mapping Framework for Multi-Robot Systems". In: *IEEE Robotics and Automation Letters* 9.1 (Jan. 2024), pp. 475–482. ISSN: 2377-3774. DOI: 10.1109/lra.2023.3333742. URL: http://dx.doi.org/10.1109/LRA.2023.3333742.

[21] Zhimin Peng et al. "ARock: An Algorithmic Framework for Asynchronous Parallel Coordinate Updates". In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016), A2851–A2879. ISSN: 1095-7197. DOI: 10.1137/15m1024950. URL: http://dx.doi.org/10.1137/15M1024950.

[22] Siddharth Choudhary et al. "Distributed Mapping with Privacy and Communication Constraints: Lightweight Algorithms and Object-based Models". In: *CoRR* abs/1702.03435 (2017). arXiv: 1702.03435. URL: http://arxiv.org/abs/1702.03435.

[23] Michael Burri et al. "The EuRoC micro aerial vehicle datasets". In: *The International Journal of Robotics Research* 35.10 (2016), pp. 1157–1163.

[24] David Schubert et al. "The TUM VI Benchmark for Evaluating Visual-Inertial Odometry". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 1680–1687. DOI: 10.1109/IROS.2018.8593419.

[25] B. W. Boehm. "A spiral model of software development and enhancement". In: *Computer* 21.5 (1988), pp. 61–72. DOI: 10.1109/2.59.

[26] *GNU General Public License*. Version 3. Free Software Foundation, June 29, 2007. URL: %5Curl%7Bhttp://www.gnu.org/licenses/gpl.html%7D.

[27] *The MIT License*. URL: %5Curl%7Bhttps://opensource.org/license/mit%7D.

[28] Dorian Gálvez-López and J. D. Tardós. "Bags of Binary Words for Fast Place Recognition in Image Sequences". In: *IEEE Transactions on Robotics* 28.5 (Oct. 2012), pp. 1188–1197. ISSN: 1552-3098. DOI: 10.1109/TRO.2012.2197158.

[29] *Boost C++ Libraries*. https://www.boost.org/.

[30] Hyeong Ryeol Kam et al. "RViz: a toolkit for real domain data visualization". In: *Telecommun. Syst.* 60.2 (Oct. 2015), pp. 337–345. ISSN: 1018-4864. DOI: 10.1007/s11235-015-0034-5. URL: https://doi.org/10.1007/s11235-015-0034-5.

[31] *Foxglove Studio*. https://foxglove.dev/.

[32] Mina Kamel et al. "Nonlinear Model Predictive Control for Multi-Micro Aerial Vehicle Robust Collision Avoidance". In: *CoRR* abs/1703.01164 (2017). arXiv: 1703.01164. URL: http://arxiv.org/abs/1703.01164.

[33] Michael Grupp. *evo: Python package for the evaluation of odometry and SLAM.* https://github.com/MichaelGrupp/evo. 2017.

[34] *OptiTrack for Robotics*. https://optitrack.com/applications/robotics/. Accessed: 2024-04-05.

[35] Jürgen Sturm et al. "A benchmark for the evaluation of RGB-D SLAM systems". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 573–580. DOI: 10.1109/IROS.2012.6385773.

[36] Tong Qin, Peiliang Li, and Shaojie Shen. "VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator". In: *IEEE Transactions on Robotics* 34.4 (2018), pp. 1004–1020. DOI: 10.1109/TRO.2018.2853729.

Appendix A

⟨APPENDIX A NAME⟩

Appendix A...

Appendix B

⟨APPENDIX B NAME⟩

Appendix B...

Appendix C

Proposal

Proposal...