

Joshua Bird

# Distributed Visual Simultaneous Localization and Mapping



Computer Science Tripos – Part II  
Queens' College

April 4, 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 4, 2024

# Proforma

Candidate number:  $\langle$ CANDIDATE NUMBER $\rangle$   
Project Title: **Distributed Visual Simultaneous Localization and Mapping**  
Examination: **Computer Science Tripos – Part II, 2024**  
Word Count:  $\langle$ WORD COUNT $\rangle$ <sup>1</sup>  
Code Line Count:  $\langle$ LINE COUNT $\rangle$ <sup>2</sup>  
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.

---

<sup>1</sup>This word count was computed using `texcount`.

<sup>2</sup>This code line count was computed using `clloc` (excluding autogenerated test output).

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Relevant Work . . . . .	1
1.2	Motivation . . . . .	1
1.3	Project Overview . . . . .	2
<b>2</b>	<b>Preparation</b>	<b>3</b>
2.1	Starting Point . . . . .	3
2.2	Visual SLAM Background . . . . .	4
2.2.1	Problem Statement . . . . .	4
2.2.2	Visual Odometry . . . . .	4
2.2.3	Feature Descriptors . . . . .	4
2.2.4	Loop Closure . . . . .	4
2.3	ORB-SLAM 3 . . . . .	4
2.4	Development Tools & Frameworks . . . . .	4
2.4.1	Webots Simulation Software . . . . .	4
2.4.2	Robot Operating System 2 Communication Middleware . . . . .	5
2.4.3	Testing Infrastructure . . . . .	6
2.4.4	Docker Containers . . . . .	6
2.4.5	Visualization Tools . . . . .	6
2.5	Datasets . . . . .	6
2.6	Algorithms . . . . .	6
2.6.1	Kabsch-Umeyama Algorithm . . . . .	6
2.6.2	Visual Bag of Words . . . . .	6
2.6.3	RANSAC . . . . .	6
2.7	Requirements Analysis . . . . .	6
2.7.1	Development Model . . . . .	6
2.7.2	Version Control and Testing . . . . .	6

<b>3</b>	<b>Implementation</b>	<b>7</b>
3.1	Agent Architectural Overview . . . . .	7
3.2	SLAM System . . . . .	8
3.2.1	Decentralized System Manager . . . . .	8
3.2.1.1	State Manager . . . . .	8
3.2.1.2	External Map Merge Finder . . . . .	9
3.2.1.3	External Map Merger . . . . .	9
3.2.1.4	Keeping Maps Merged . . . . .	10
3.2.1.5	Local Key Frame Inserter . . . . .	10
3.2.1.6	External Key Frame Inserter . . . . .	10
3.2.2	Generalizing to $N \geq 3$ Agent Systems . . . . .	11
3.2.2.1	Map Alignment Refiner . . . . .	12
3.2.2.2	Reference Frame Manager . . . . .	13
3.2.2.3	Visualization Publisher . . . . .	13
3.2.2.4	Losing Localization . . . . .	13
3.2.3	Map Serialization and Deserialization . . . . .	13
3.3	Motion Controller . . . . .	13
3.3.1	Follow The Leader . . . . .	13
3.3.2	Multi-Agent Collision Avoidance . . . . .	13
3.4	Central Management Interface . . . . .	13
3.5	Custom Evaluation Suite – Multi-Agent EVO . . . . .	13
3.6	Simulation Environment . . . . .	14
3.7	Real World Implementation . . . . .	14
3.7.1	Repository Overview . . . . .	14
<b>4</b>	<b>Evaluation</b>	<b>15</b>
4.1	Review of Success Criteria . . . . .	15
4.2	Benchmarking . . . . .	15
4.3	Comparison to Related Work . . . . .	15
4.4	Real World Experiments . . . . .	15

**5 Conclusions** **16**

5.1 Lessons learned . . . . . 16

5.2 Future Work . . . . . 16

**Bibliography** **17**

**A <APPENDIX A NAME>** **18**

**B <APPENDIX B NAME>** **19**

**C Proposal** **20**

# 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 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<sup>1</sup>, Boston Dynamic's GraphNav system used on their robot Spot<sup>2</sup>, and several models of DJI quadcopters<sup>3</sup> making this a particularly practical field of research.

### 1.1 Relevant Work

While there have been many advanced implementations of visual SLAM over the last decade, very few of them have focused on distributed multi-agent systems, instead focusing on single-agent or centralized multi-agent systems. ORB-SLAM3 [1] is perhaps the most popular single-agent SLAM system, and often ranks at the top of benchmarks in a variety of environments [2], however it is fundamentally a single-agent system with no considerations in place for collaboration among peers. Out of the few multi-agent systems that do exist, the majority of them require a centralized server to manage communications between the agents and perform map merging, such as CCM-SLAM(2019) [3] and COVINS(2021) [4] among many others.

todo: add comparison table?

### 1.2 Motivation

Multi-robot systems are only becoming more common as automation in numerous fields continues to grow, such as self-driving cars, drone swarms and warehouse robots. These systems require the agents to understand the world around them, as know each other's locations within that world for collision avoidance. This task is often achieved with technologies such as GPS or motion capture setups, however we can not assume that all environments will have access to these systems. A few emerging examples include:

---

<sup>1</sup><https://developers.google.com/ar/develop/fundamentals>

<sup>2</sup><https://support.bostondynamics.com/s/article/GraphNav-Technical-Summary>

<sup>3</sup>DOI: 10.1016/j.vrih.2019.09.002

- 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 is able to assist, as it enables us to build a map of an unknown environment and keep agents aware of their relative poses. However, as noted in section 1.1, the majority of existing multi-agent visual SLAM implementations are centralized systems, requiring the agents to maintain reliable communications with the central server in order to operate. This is somewhat counterintuitive, as environments which don't have access to GPS or motion capture systems are more than likely to also 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 which 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 with one another.

It is easy to see the broad reaching uses cases this opens up. 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 only know their peer's locations when they are within communication range, but this is sufficient if we only need the relative position for collision avoidance (as is common in multi-robot systems).

## 1.3 Project Overview

In this project, I:

1. Design and implement a distributed multi-agent visual SLAM system that 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. Create a simulation environment for testing and evaluating my system locally.
3. Deploy my system on physical robots, demonstrating the practical use cases of this system and benchmarking real world performance.
4. Develop *Multi-Agent EVO* – an evaluation library for multi-agent SLAM systems based on *EVO* [5].
5. Evaluate the performance of my system on standardized datasets, **demonstrating that its performance is superior to comparable state-of-the-art systems.**



# Chapter 2

## Preparation

### 2.1 Starting Point

As noted in the *Relevant Work* section, 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 thinking 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 a novel and under-researched aspect of the field.

I chose ORB-SLAM 3 as the single agent SLAM system to base my system on top of, as its researchers released code alongside their paper. I primarily utilized the system’s visual odometry (VO) front end and helper functions from the backend to perform operations such as bundle adjustment.

However, as I will expand on in the *ORB-SLAM 3* section, a significant amount of time and effort was still required to understand its extremely large undocumented codebase, especially since an almost complete understanding of its inner workings were needed to both extract and inject map information from the system, as it was never designed to work in a multi-agent context. In retrospect, using an existing single-agent SLAM system as a foundation was not as much of a time-saver as I had initially hoped.

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 is accurate and performant enough to have real-world use cases.

At the time of submitting my project proposal, I had forked the ORB-SLAM3 [1] git repository<sup>1</sup> 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<sup>2</sup>.

I had no prior experience working with SLAM systems, but I had researched 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.

---

<sup>1</sup>[https://github.com/UZ-SLAMLab/ORB\\_SLAM3](https://github.com/UZ-SLAMLab/ORB_SLAM3)

<sup>2</sup>[https://github.com/jyjblrd/part\\_II\\_project](https://github.com/jyjblrd/part_II_project) TODO: Redact

## 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 [6] and research papers [7] have discussed in depth, which I will attempt to summarize here.

### 2.2.1 Problem Statement

### 2.2.2 Visual Odometry

### 2.2.3 Feature Descriptors

### 2.2.4 Loop Closure

## 2.3 ORB-SLAM 3

ORB-SLAM 3 is a cutting-edge single-agent SLAM system, often ranking at the top of visual SLAM comparison papers. As noted in *Starting Point* section, I used ORB-SLAM as my starting point, primarily for its mature visual odometry (VO) front end and backend helper functions.

## 2.4 Development Tools & Frameworks

From the start, I knew that a well-structured development plan would be essential to the successful implementation of this project. An entire suite of infrastructure had to be implemented to aid the development of my distributed SLAM system, including:

- Simulation software
- Saving and loading test cases
- Motion control systems
- Evaluation libraries

### 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, allowing me to easily test my system in various environments and scenarios before committing to deploying it on physical robots. Additionally, it allowed me to record numerous test cases which I used as regression tests and benchmarks for my system throughout development.

TODO: ...

## 2.4.2 Robot Operating System 2 Communication Middleware

Robot Operating System (ROS) 2 is the glue holding everything together, allowing independent software processes and hardware to communicate through an abstracted 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 on 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.

This is best illustrated with an example. Below is a toy distributed SLAM system. Given agents  $\{\text{agent}_n \mid n \in \{1, 2\}\}$ , each agent has a camera which publishes to the `/agentn/camera` topic. The `SLAM_Processorn` node subscribes to the `/agentn/camera` topic, and performs simultaneous localization and mapping using the image stream. The `SLAM_Processorn` node then publishes to the `/agentn/new_map_data` topic, which the other agent can subscribe to and use to improve their local map.

todo: add diagram

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

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 turns my project from simply being a nice codebase to something that anyone can take and run on their own robots.

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

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Data\\_Distribution\\_Service](https://en.wikipedia.org/wiki/Data_Distribution_Service)

### **2.4.3 Testing Infrastructure**

### **2.4.4 Docker Containers**

### **2.4.5 Visualization Tools**

## **2.5 Datasets**

## **2.6 Algorithms**

### **2.6.1 Kabsch-Umeyama Algorithm**

### **2.6.2 Visual Bag of Words**

### **2.6.3 RANSAC**

## **2.7 Requirements Analysis**

### **2.7.1 Development Model**

### **2.7.2 Version Control and Testing**

# Chapter 3

## Implementation

### 3.1 Agent Architectural Overview

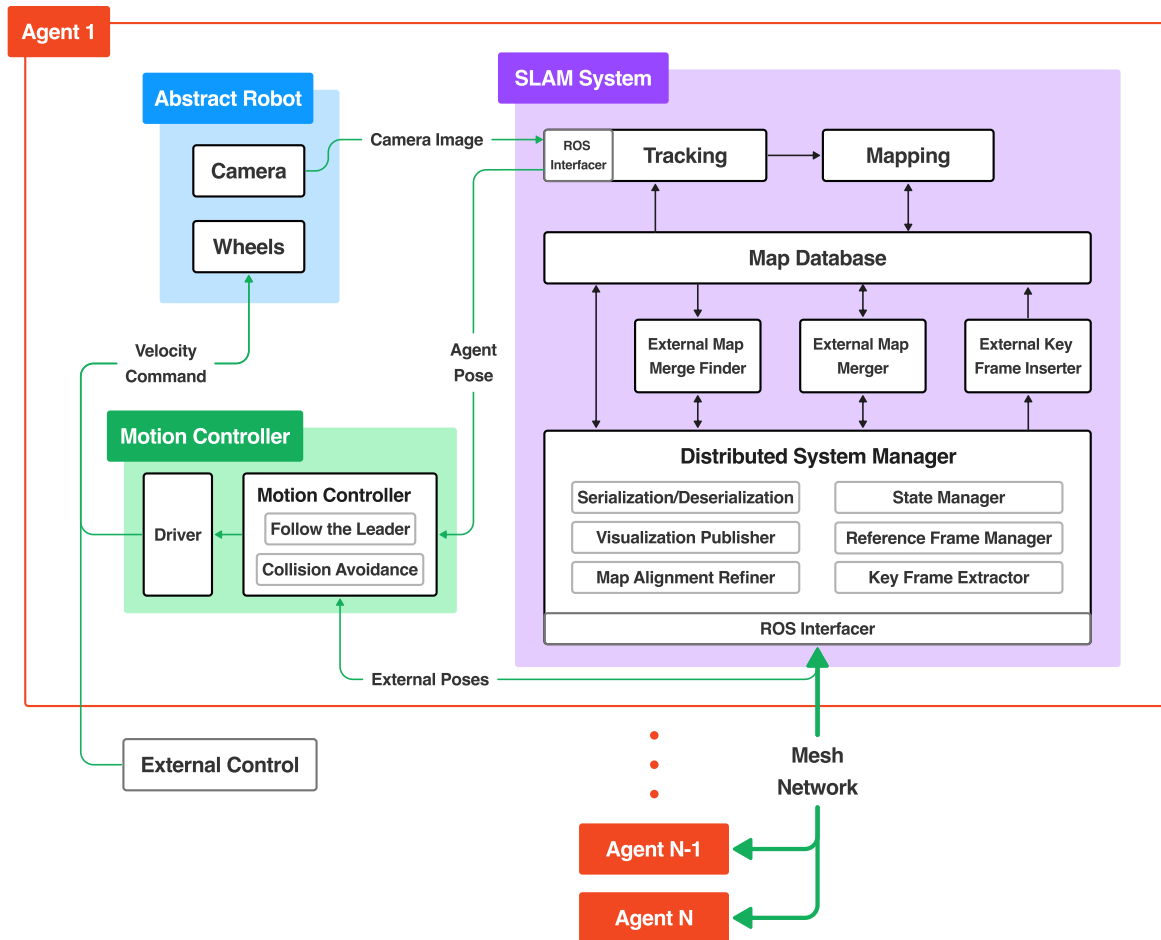


Figure 3.1: Agent diagram. Green arrows represent messages sent over ROS topics, while black arrows are internal communications within a node.

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 which provides an interface to the robot’s hardware, as explained in subsection 2.4.2. 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 which performs the **Tracking** and **Mapping** tasks. While substantial modifications were made to the base single-agent system, I will generally focus on the decentralized layer I have built on top in the interest of space.

### 3.2.1 Decentralized System Manager

Decentralized collaborative SLAM systems have significantly more complexity than a single-agent or even centralized collaborative system, 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 robustness and correctness of the system, which I have implemented in the **Decentralized System Manager** component.

For the sake of simplicity, the following explanations will explore the interactions between just two agents: a *local* and *external* agent. However, this framework easily generalizes to a system with an arbitrary number of agents when augmented by some simple rules set out in the *Generalizing to  $N \geq 3$  Agent Systems* Section.

#### 3.2.1.1 State Manager

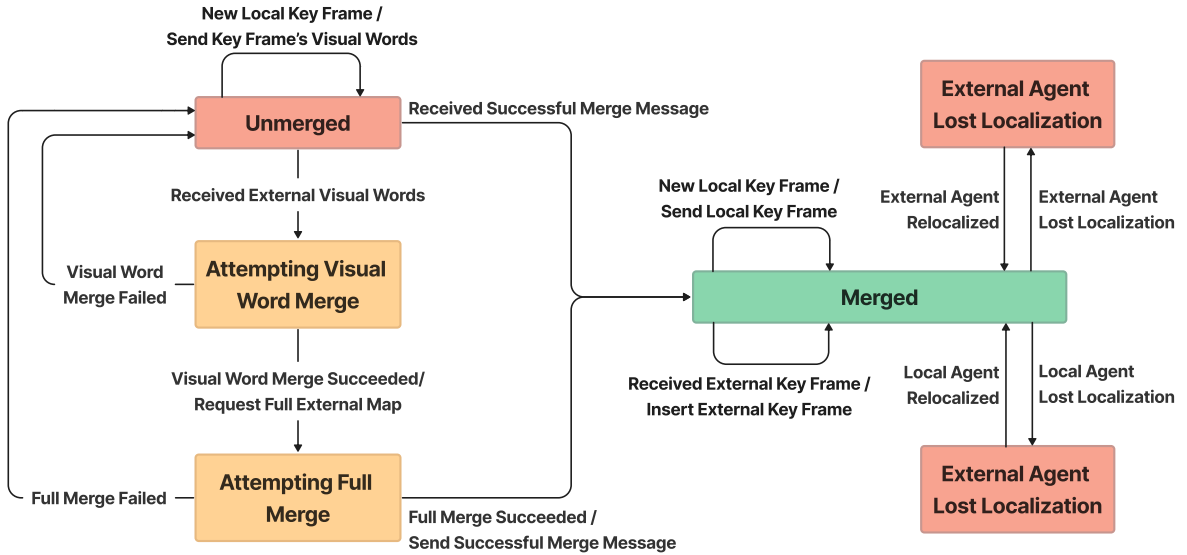


Figure 3.2: SLAM system state machine for a single peer.

Each agent’s **Decentralized System Manager** maintains a state machine for every peer in the system, shown in Figure 3.2. All peers are initially in the **unmerged** state, meaning that they are in different coordinate frames and can not collaboratively build a map together. As the local agent starts to explore the same places as its peers, we are able to recognize the visual overlaps and merge 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.

### 3.2.1.2 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 clearly 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 before attempting a full map merge. 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 key frames, we calculate the visual bag of words seen by that key frame and send them to our peers over the `/new_key_frame_bows` ROS topic. These visual words are significantly smaller than sending over the complete map and enable agents to detect if there is a significant amount of visual overlap between its local map and the external agent's map. This is computed by Algorithm 1.

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 so the agents can begin collaborating and determine relative positioning. TODO: generate stats on this?

TODO: generate stats on how much this cuts down on bandwidth and computation

---

**Algorithm 1** Map merge finder using visual words. TODO: improve

---

**Input:**  $E$ : Set containing external key frame's visual words

**Output:** *success*: Boolean value signaling if a merge is possible based on visual words

- 1:  $(externalMergeScore, bestMatchKeyFrame) \leftarrow CalculateMergeScore(E)$
  - 2:  $I \leftarrow ComputeVisualWords(bestMatchKeyFrame)$
  - 3:  $(internalMergeScore, ) \leftarrow CalculateMergeScore(I)$
  - 4:  $success \leftarrow externalMergeScore \geq internalMergeScore$
- 

### 3.2.1.3 External Map Merger

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 through the `/get_current_map` ROS topic. Once the map is received, we deserialize it and place it into our *map database* data structure.

From this point, we need to confirm that a map merge is possible using the full map data.

TODO

Given the external key frame  $k_0$  whose visual words triggered this map merge attempt, we extract a *local window*  $K$  of key frames connected to  $k_0$  in the covisibility graph.

If the full map merge is ultimately successful, we broadcast a `/successfully_merged` message to tell the external agent that we have successfully merged their map. The external agent will then move to the `merged` state and both agents will begin sharing key frames with each other.

It is important to note that this system requires only one agent to calculate the map merge, significantly reducing the computational overhead of map merging, especially in systems with many agents (further explained in subsection 3.2.2).

### 3.2.1.4 Keeping Maps Merged

### 3.2.1.5 Local Key Frame Inserter

As mentioned in the External Key Frame 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

### 3.2.1.6 External Key Frame Inserter

Once the local and external agents have merged their maps and share the same coordinate frame, they can begin sharing key frames with each other.

Each agent maintains a set of sent key frames  $K_{sent}$  and map points  $M_{sent}$ . The set of unsent key frames and map points are therefore represented as  $K_{unsent} = K/K_{sent}$  and  $M_{unsent} = M/M_{sent}$  respectively. Once  $\#K_{unsent}$  exceeds a certain threshold, we serialize  $K_{unsent}$  and  $M_{unsent}$  and send them to the external agent. Finally, we add  $K_{unsent}$  to  $K_{sent}$  and  $M_{unsent}$  to  $M_{sent}$ .

Upon receiving the serialized key frames 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 by the *external key frame inserter*.

The external key frame 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:

1. **Pop external key frame  $k_{ext}$  from front of queue.**
2. **Relink  $k_{ext}$  with covisible key frames and observed map points in the local map.**  
 $k_{ext}$  contains references to its covisible 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.
3. **Move  $k_{ext}$  and its new 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.<sup>1</sup>
4. **Relink  $M_{ext}$  with key frames in the local map which observe it.**  
 $M_{ext}$  contains references to key frames which observe it which have already been sent or were generated by another agent. We search our local map for key frames that match these references, reconnecting them.

---

<sup>1</sup>I had previously used a *key frame 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 key frames from properly integrating with the local map. However, experimental testing showed that this method instead caused the local and external maps to frequently diverge.



### 5. Merge $M_{ext}$ with map points in the local map.

$M_{ext}$  will already be correctly linked to observing key frames in the local map, 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 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.

TODO: add more details? idk

### 6. Perform a local bundle adjustment around $k_{ext}$ .

Since we have merged map points in the previous step, we need to perform bundle adjustment to tweak the key frame and map point locations to minimize reprojection error. This helps create a more accurate map, minimizing tracking error. We also only need to perform this bundle adjustment locally, greatly reducing the computational cost.

TODO: add diagram

## 3.2.2 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  $A = \{agent_1, agent_2, \dots, agent_N\}$ , where  $agent_i$  is the agent with and  $agentId = i$ . 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  $agentId$ .

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

**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 agents 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 is much smaller than the total number of agents.

Additionally, having all merge operations performed by the group leader prevents potential race conditions introduced by communication latency, removing unnecessary complexity.

As discussed in subsubsection 3.2.1.2 and subsubsection 3.2.1.3, 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 key frames, it will instead send the visual words of all key frames generated by agents within its group. This introduces no additional intra-group communication, as all agents within a group already send each other new key frames.

Moving on to (2), once a pair of group leaders ( $agnet_n, agnet_m$ ) (with  $n < m$ ) have successfully merged their maps,  $agnet_m$  with the larger agentId will change its coordinate frame to that of  $agent_n$ .  $agent_m$  will then send the transform from  $agent_m$  to  $agent_n$ 's coordinate frame 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 using Equation 3.1, and agents begin exchanging key frames to update each other's maps.

TODO: fix obviously

$$i \in G_n, j \in G_m, state_{i-j} \in S. state_{i-j} = merged \quad (3.1)$$

$$\text{where } G_n \in G \text{ and } G_m \in G \text{ are } agent_n \text{ and } agent_m \text{'s groups respectively.} \quad (3.2)$$

This is perhaps best represented in visual form. Take the simple example where: ...

### 3.2.2.1 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 TODO: define. This is largely a side effect of our aggressive early merge strategy which 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. After enough changes have been made to our local map, we perform the following steps:

1. **Request map point locations from the lead agent.**

This is defined as the set  $TaggedMP_{ext}$  where  $TaggedMP_{ext_i} = (uuid, (x, y, z))$

2. **Extract local map point locations.**

This is defined as the set  $TaggedMP_{local}$  where  $TaggedMP_{local_i} = (uuid, (x, y, z))$

3. **Use the Kabsch-Umeyama and RANSAC algorithms to find transform  $T$  which best aligns  $TaggedMP_{ext}$  to  $TaggedMP_{local}$ .** The Kabsch-Umeyama algorithm finds the SIM(3) transformation  $T$  from  $TaggedMP_{ext}$  to  $TaggedMP_{local}$ , minimizing the RMSE. However, our input data has a large number of outliers so we use RANSAC on top 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  $T$  to our local map.**

#### 3.2.2.2 Reference Frame Manager

#### 3.2.2.3 Visualization Publisher

#### 3.2.2.4 Losing Localization

### 3.2.3 Map Serialization and Deserialization

## 3.3 Motion Controller

#### 3.3.1 Follow The Leader

#### 3.3.2 Multi-Agent Collision Avoidance

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

- Manually control the agent's poses.
- Record trajectories generated by the SLAM system as well as ground truth data for later evaluation.
- Record camera data and play it back for testing and benchmarking purposes.

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 in this project. TODO: move to somewhere more relevant?

This project aims to create more than just a research project which 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.

## 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. (TODO: say that no multi agent slam paper released code used for evaluation) Therefore, I have developed an open-source multi-agent SLAM evaluation tool: *Multi Agent EVO*, based on the popular single agent SLAM evaluation tool *EVO* [5].

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 will be 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 on how long it takes for agents to successfully merge maps, the accuracy of relative pose estimation, and much more.

TODO: add graph illustrating this coordinate frame stuff TODO: perhaps list out all capabilities added

## **3.6 Simulation Environment**

## **3.7 Real World Implementation**

### **3.7.1 Repository Overview**

# Chapter 4

## Evaluation

### 4.1 Review of Success Criteria

### 4.2 Benchmarking

### 4.3 Comparison to Related Work

### 4.4 Real World Experiments

# Chapter 5

## Conclusions

### 5.1 Lessons learned

### 5.2 Future Work

# 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] 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.
- [4] Patrik Schmuck et al. *COVINS: Visual-Inertial SLAM for Centralized Collaboration*. 2021. arXiv: 2108.05756 [cs.RO].
- [5] Michael Grupp. *evo: Python package for the evaluation of odometry and SLAM*. <https://github.com/MichaelGrupp/evo>. 2017.
- [6] Xiang Gao and Tao Zhang. *Introduction to visual SLAM: from theory to practice*. Springer Nature, 2021.
- [7] Hugh Durrant-Whyte and Tim Bailey. “Simultaneous localization and mapping: part I”. In: *IEEE robotics & automation magazine* 13.2 (2006), pp. 99–110.

# Appendix A

⟨APPENDIX A NAME⟩

Appendix A...



# Appendix B

⟨APPENDIX B NAME⟩

Appendix B...

# Appendix C

## Proposal

Proposal...