Joshua Bird

Distributed Visual Simultaneous Localization and Mapping

Computer Science Tripos – Part II
Queens' College

April 6, 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 6, 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 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 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:

[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

- 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 performace 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[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[2].

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.

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

[2] `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 $\{\texttt{agent}_n \mid n \in \{1, 2\}\}$, each agent has a camera which publishes to the $\texttt{/agent}_n\texttt{/camera}$ topic. The $\texttt{SLAM\_Processor}_n$ node subscribes to the $\texttt{/agent}_n\texttt{/camera}$ topic, and performs simultaneous localization and mapping using the image stream. The $\texttt{SLAM\_Processor}_n$ node then publishes to the $\texttt{/agent}_n\texttt{/new\_map\_data}$ 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)[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.

[1] 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

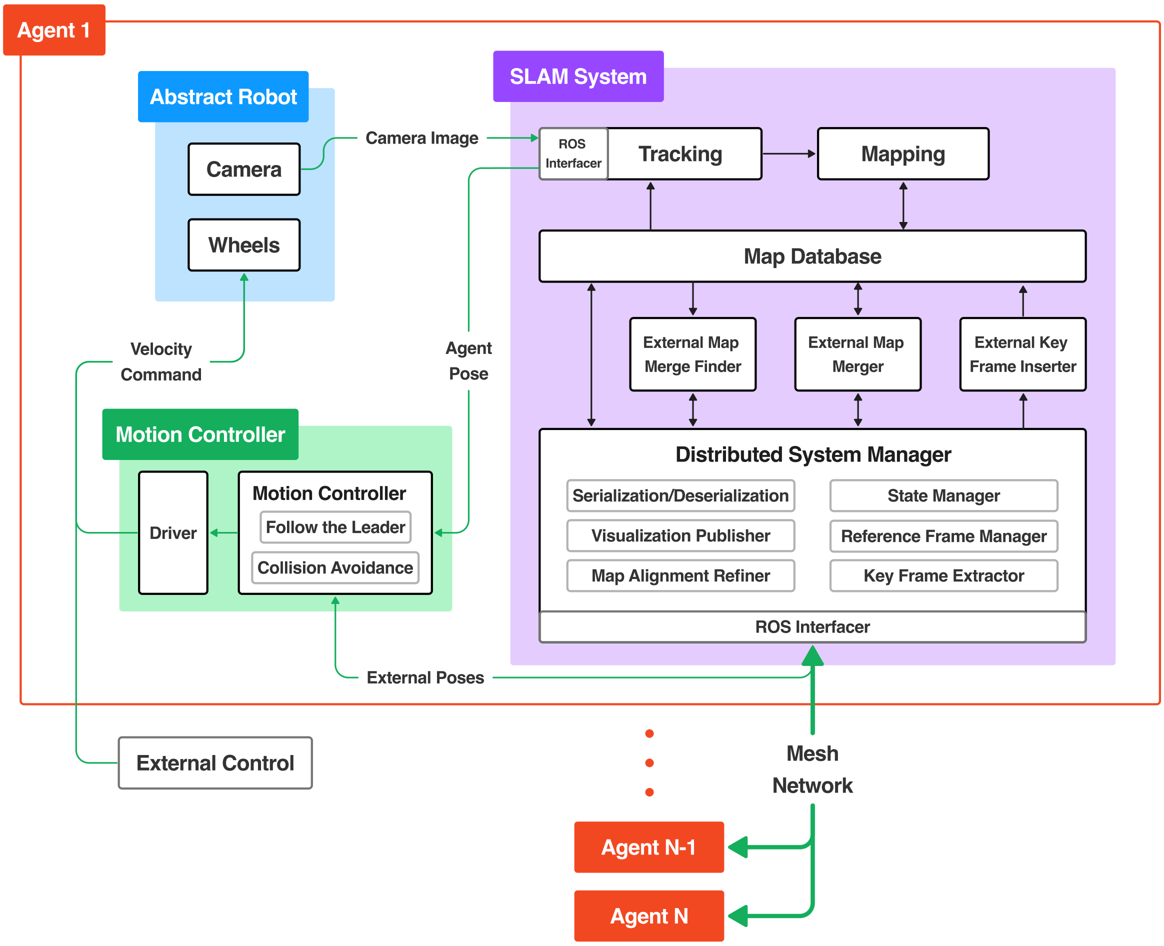

Figure 3.1: Agent diagram. Green arrows represent messages sent over ROS topics, while black arrows are internal communications within a node.

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 that 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 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 layer I have built on top in the interest of space.

TODO: Add info on map datastructure

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 corectness 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. In the *Generalizing to $N \geq 3$ Agent Systems* Section, we sill show how this easily generalizes to a system with an arbitrary number of agents when augmented by some simple rules.

3.2.1.1 State Manager

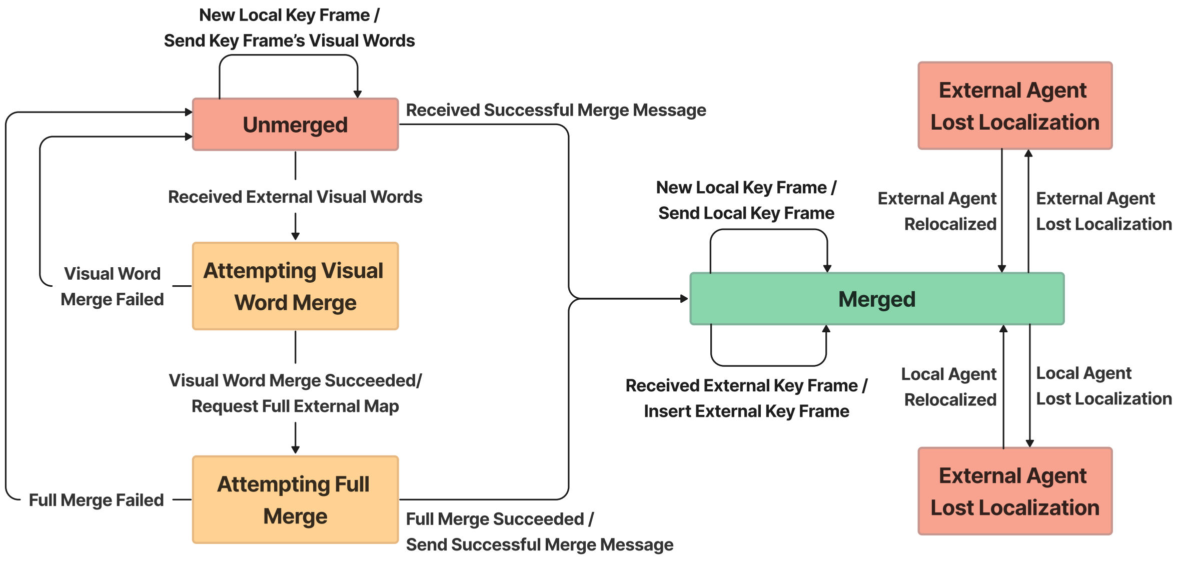

Figure 3.2: SLAM system state machine for a single peer.

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

```
$(externalMergeScore,\ bestMatchKeyFrame) \leftarrow CalculateMergeScore(E)$
$I \leftarrow ComputeVisualWords(bestMatchKeyFrame)$
$(internalMergeScore,\ _{)} \leftarrow CalculateMergeScore(I)$
$success \leftarrow externalMergeScore \geq internalMergeScore$
```

3.2.1.3 External Map Merger

Performing full map merges is perhaps the most improtant 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 final version of the external map merger works as follows:

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

2. **Confirm that a map merge is possible using the full map data.**
   The visual word map merge finder works well considering its very low communication overhead, however we need to confirm that a map merge is actually possible using the full map data from both agents.

   TODO

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

3. **Find the translation $T_{loc \to ext}$ that aligns the local map to the external map.**
   TODO

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

5. **Move a subset of the external map's key frames and map points into our local map**
   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.

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

7. **Run bundle adjustment to optimize map point and key frame positions.**

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

9. **Broadcast successful merge message.**
   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.

By the end of the merge process, the local agent will change its coordinate frame to that of the external agent.

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 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. More details of the deserialization process are given in subsection 3.2.3.
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.[1]
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. More details of the deserialization process are given in subsection 3.2.3.
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

[1] 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.

3.2.1.6 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 use the external map points when running the 7 point algorithm (TODO: ref section?) to generate new local key frames.

The *Tracking* and *Mapping* modules, which are responsible for localizing the robot and generating new key frames, only interact with the *Map Database*. Our external key frame inserter does all the work of properly reconecting external map points and merging duplicate map points, leaving the *Map Database* datastructure looking as if all the map points and key frames were generated locally. This essentially abstracts the *Tracking* and *Mapping* modules away from the distributed aspects of the system, meaning they fully use the external map points when localizing the agent and generating new key frames.

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 $A = \{agent_1, agent_2, ..., agent_N\}$, where $agnet_i$ is the agent with ID = i. Within this system we maintain a state $S_i = state_{n-m}$ for every agent pair $(agnet_n, agnet_m)$, 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 $agent_m$'s children. 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 $G = \{\{agent_1\}, \{agnet_2\}, ..., \{agent_N\}\}$

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 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 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.

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 key frames, it will send the visual words of all key frames generated by agents within its group. Keep in mind that it will only be sending these visual words to other group leaders. This introduces no additional intra-group communication, as all agents within a group already send each other all 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, the agent with the larger ID $(agnet_m)$ 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 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. Eventually, once all agents are merged together, they will all be in $agent_0$'s coordinate frame.

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.

TODO: fix obviously

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

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

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

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 [8] Serialization C++ library since it supports standard library collections and other common classes.

As discussed in section 3.2, maps contain key frames 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 custom class, describing which class 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 key frame'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 key frame k and its map points M_k to an external agent, due to the many-to-many relationship between key frames 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.

We manage these connections by giving every key frame 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. The beauty of UUIDs is that they do not require a centralized server to assign a unique ID to every object. Instead, we can generate the unique IDs without any communication with our peers[1].

TODO: add figure of this map fragment serialization and deserialization

[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.

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 which I have not had the space to discuss in this dissertation.

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 out 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 and with a 180° rotational offset. This ensures that the leader and follower have no visual overlap, 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 derived from [9] 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 $\boldsymbol{x}$ as the control input $\boldsymbol{v_{cmd}}$ applied to its current pose $\boldsymbol{p}$:

$$\boldsymbol{x}(t) = \boldsymbol{p} + \int_0^t \boldsymbol{v_{cmd}}\, dt \tag{3.3}$$

We can then solve the following minimization problem to find an optimal $\boldsymbol{v_{cmd}}$:

$$\begin{aligned} &\min_{\boldsymbol{v_{cmd}}} \int_{t=0}^{T} J_x(\boldsymbol{x}(t), \boldsymbol{x_{ref}}(t)) + J_s(\boldsymbol{x}(t)) + J_d(\boldsymbol{x}(t))\ dt \\ &\text{subject to } \boldsymbol{v_{min}} \leq \boldsymbol{v_{cmd}} \leq \boldsymbol{v_{max}} \end{aligned} \tag{3.4}$$

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:

$$J_x(\boldsymbol{x}, \boldsymbol{x_{ref}}) = \|\boldsymbol{x} - \boldsymbol{x_{ref}}\|^2 \tag{3.5}$$

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

$$J_s(\boldsymbol{x}) = \sum_{i=1}^{N_{static}} \frac{s_s * Q_s}{1 + \exp(d_i^{static}/s_s)} \tag{3.6}$$

$$J_d(\boldsymbol{x}) = \sum_{i=1}^{N_{dynamic}} \frac{s_d * Q_d}{1 + \exp(d_i^{dynamic}/s_d)} \tag{3.7}$$

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 far the agent stays away from static obstacles compared to dynamic ones.

Additionally, we define s_s and s_d as a scale normalizing parameter that ensures 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 ensures 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 3.8. 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.

$$x_{min} = \ln\left(\frac{\sqrt{Q_s^2 - 4Q_s} + Q_s}{2} - 1\right) \tag{3.8}$$

The key benefit of these cost functions is that the optimal distance to obstacles of r is invariant to parameters Q_s, Q_d, and r. This is in contrast to [9] which required you to retune the parameters κ and Q upon changing agent radius r, greatly slowing down development.

TODO: add graph for cost functions, maybe a 2d surface showing the cost function

3.3.2.2 Implementation Details

Solving the integral in Equation 3.4 is computationally inefficient, therefore we split our calculations into discrete time steps. Specifically, the time horizon is set as $T = TODO$ with $TODOms$ timesteps.

The Sequential Least Squares Programming minimization method is used due to its robustness and ability to constrain the optimization space. We set parameters $Q_s = TODO$ and $Q_d = TODO$ to make the agent more adverse to approaching dynamic obstacles compared to static ones, as there is more uncertainty in the dynamic obstacle's location. To calculate the future locations of dynamic obstacles $x_i^{dynamic}$ 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. Therefore, 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. 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.

TODO: ACTUALLY EXPLAIN WHAT THIS INTERFACE DOES

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

The Webots Simulation Software Section in the Preparation Chapter explains the motivation behind using a simulator. Essentially, leveraging simulations enabled much faster iteration cycles and the ability to test in a variety of environments. This section will focus on the implementation details of the simulation software.

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 and the data was streamed to the Ubuntu VM through a websocket bridge developed by Webots.

I then developed a ROS node which exposes the Webots cameras and agent controls as ROS topics, fo for my SLAM system to connect to.

3.7 Real World Implementation

After months of development within a simulator, my distributed SLAM system was able to seamlessly be deployed in 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 2 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 controlled by a NVidia Jetson Orin. 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 for 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, dockerizing my project means that I can build it once and share that built docker image with all the robots. 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 were provided by the Prorok Lab's OptiTrack Motion Capture System, which advertises accuracies of $\leq$0.3mm and polling rates of 180hz [10]. These 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 External Camera Tracking

In addition to tracking the RoboMasters, we also track an external camera as it captures the experiment. Knowing the external camera's trajectory allows us to project 3D visualization data onto the recorded video as the camera moves around in real time, giving an augmented reality experience. This can be used to visualize the SLAM system's key frame and map point locations in the real world, or to overlay the SLAM system's predicted trajectory onto the video to see how it 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. We therefore use the Kabsch-Umeyama algorithm described in subsection 2.6.1 to align the trajectories captured by the motion capture system and SLAM system after enough data has been collected to create a successful alignment. Finally, use Foxglove studio to draw our 3D markers and project them on top of the moving camera's video stream.

TODO: talk about raspberry pi setup TODO: talk about the novelty of this sytem? is it novel? why is it useful?

3.8 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.

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

[9] 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`.

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

Appendix A

⟨APPENDIX A NAME⟩

Appendix A...

Appendix B

⟨APPENDIX B NAME⟩

Appendix B...

Appendix C

Proposal

Proposal...