8-2018

Low-Cost Multiple-MAV SLAM Using Open Source Software

Andrew Hollenbach
anh7216@rit.edu

Low-Cost Multiple-MAV SLAM Using Open Source Software

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Zack Butler, Supervisor

Dr. Minseok Kwon, Reader

Dr. Joseph Geigel, Observer

Low-Cost Multiple-MAV SLAM Using Open Source Software

by

Andrew Hollenbach, B.S.

THESIS

Presented to the Faculty of the Golisano College of Computer and Information Sciences
Rochester Institute of Technology
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science in Computer Science, Department of Computer Science

Rochester Institute of Technology

August 2018

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Zack Butler, for his mentorship, guidance, and patience throughout the process. You are a wonderful professor and mentor, and I appreciate all of the work you put into making this thesis a success.

I would also like to thank the rest of my comittee, Dr. Minseok Kwon and Dr. Joseph Geigel for their time and support. I would also like to thank Christina Rohr, who is a shining light in the CS department and goes above and beyond for all of her students. It has been a pleasure to work with all four of you throughout my time at RIT.

To my mom and dad—thank you for everything and more. You are patient and understanding, even when it tests your beliefs. You are always willing to make the drive to Rochester (or anywhere) to see me and support me, even for just a few hours. To Rebecca—you are the inspiration for much of what I do. I appreciate you so much, even from afar. I love you all!

To Serena, thank you for all of your enduring support, care, and insight from all too far away.

Thank you to Joey and Vivian—you have been there for me countless times in countless ways.

Finally, thank you to the many other family members, friends, and

coworkers who have helped me make this a reality. It is with all of your help that I am proud to achieve my Master's.

Abstract

Low-Cost Multiple-MAV SLAM Using Open Source Software

Andrew Hollenbach, M.S.
Rochester Institute of Technology, 2018

Supervisor: Dr. Zack Butler

We demonstrate a multiple micro aerial vehicle (MAV) system capable of supporting autonomous exploration and navigation in unknown environments using only a sensor commonly found in low-cost, commercially available MAVs—a front-facing monocular camera. We adapt a popular open source monocular SLAM library, ORB-SLAM, to support multiple inputs and present a system capable of effective cross-map alignment that can be theoretically generalized for use with other monocular SLAM libraries. Using our system, a single central ground control station is capable of supporting up to five MAVs simultaneously without a loss in mapping quality as compared to single-MAV ORB-SLAM. We conduct testing using both benchmark datasets and real-world trials to demonstrate the capability and real-time effectiveness.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Simultaneous localization and mapping (SLAM) allows robots to safely navigate and explore unknown environments. Teams of robots can perform this task more quickly, although as outlined by *Saeedi et al*, multiple-robot SLAM provides its own unique challenges. A few key issues include uncertainty of relative robot poses, map updating, and communication, but several others are mentioned as well [43].

Availability of capable, low-cost micro aerial vehicles (MAVs) continues to rise, lowering the barrier of entry to own not one, but multiple MAVs. MAVs provide an added benefit over their ground-based counterparts in that they can avoid most ground obstacles with ease and are fairly impervious to drastic terrain change. However, low-cost MAVs have a limited payload and battery life, and as such, many commercially available platforms ship with very few sensors—typically an Inertial Measurement Unit (IMU) and/or a monocular camera. Popular research with these MAVs is often in the field of visual SLAM, and several visual SLAM algorithms have been recently published and open sourced [17, 20, 38]. These algorithms will be further discussed in the related work section.

This thesis will present a system applying open source state-of-the-art monocular SLAM algorithms to a team of commercially available MAVs in GPS-denied scenarios. The problem is twofold, addressing the issues of SLAM across multiple robots, as well as dealing with the restrictions of low-cost MAVs in order to do so.

1.1 Motivation

We explore the potential of UAV-based SLAM in particular due to the increased flexibility provided by aerial vehicles in comparison to their ground-based counterparts. Aerial vehicles are generally impervious to terrain issues, making them ideal candidates for traversing areas with dangerous and dynamic terrain. Furthermore, while ground vehicles typically provide sensing from a fixed viewpoint along a two-dimensional plane, aerial vehicles can provide a wide variety of viewpoints due to their ability to seamlessly move in all three dimensions with ease.

Control and navigation of singular UAVs has been researched with many positive results, but an area less explored is unleashing the potential of coordinating multiple UAVs to work together to conduct SLAM. The use of multiple UAVs can provide redundancy in the event of UAV failure, and can typically cover an area more quickly, a valuable tool in search and rescue and many other situations. We have seen previous work in the area demonstrate groups of MAVs with downward-facing cameras [19], which is a slightly different problem as compared to using front-facing cameras, primarily since the

visual odometry algorithm does not have to be as robust to rotation issues. With the recent release of ORB-SLAM [38], we are presented with a visual SLAM algorithm capable of maintaining tracking while handling the rotations seen in quadrotors, but also lightweight enough to handle several UAVs, even when running from a single base machine.

Finally, we note the rising popularity of UAVs with front-facing monocular cameras. Although often found in toy and hobby shops, these UAVs can be harnessed for research, and the commercial availability and extremely low cost allows for systems with large quantities of cheap UAVs, with capable UAVs coming in under $50 USD, although sacrificing some sensing power and UAV quality. As an effect of this, sensed maps with low-cost MAVs using just a monocular camera may be less accurate, particularly when it comes to scale, as compared to maps generated from depth sensors. However, these low-cost systems can provide benefits over a small number of costly UAVs, which may contain better sensors but are limited in number and costlier to replace. By pushing towards systems that require a lower barrier to entry in terms of sensors and computing power, we open up potential uses in education as well as for situations where map precision can be sacrificed for large benefits in redundancy and speed. Our selection in SLAM algorithm was made with this in mind.

1.2 Problem Statement

The objective of this thesis is to develop a multiple MAV system capable of supporting autonomous exploration and navigation in unknown environments using only a sensor commonly found in low-cost, commercially available MAVs—a front-facing camera. Through modification of a popular open-source SLAM library to support multiple inputs, we aim to leverage the speed increases of multiple MAVs to produce maps comparable in quality to a single MAV SLAM system in significantly less overall time.

Chapter 2

Background

2.1 SLAM and Autonomous Navigation

Simultaneous localization and mapping (SLAM) is a core problem in mobile robotics that, when coupled with a path planning technique, enables robots to safely navigate and explore unknown environments autonomously. In SLAM, the robot is presented with a set of sensor readings, and then uses this, coupled with previous sensor readings, in order to not only create a map, but to localize on that map as well, simultaneously. This map can be sparse or dense, and the field has developed over the years, being able to achieve efficient loop closure (recognizing when the robot is revisiting an area it has already been), relocalization, and more.

SLAM comes in many forms and has been used to solve many problems. At the simplest, SLAM can be applied to a webcam moving around in a person's hand, and at its most complex, it can be performed by fusing together a wide and varying array of sensors, strapped to a massive flying unmanned aerial vehicle (UAV). While SLAM does not inherently imply robotics, the two are strongly correlated, and due to the content of this thesis, it will often be referred to in the context of robotics.

2.1.1 What is SLAM?

Imagine a robot lands from outer space and awakens with no bearings and no map. The robot begins to explore, noting its position relative to things it sees around it, and it builds out a map based on what it sees. This is the essence of SLAM—creating a map of everything the robot senses while simultaneously positioning the robot on the aforementioned map. Of course, our MAVs are not from outer space, but they face the same challenge as our outer space robot.

Now imagine sometimes the robot messes up exact details and may have to go back and edit its map based on further observation. Sometimes while wandering around it runs into places it has already been and realizes the map was a little off. Sometimes the robot gets lost, but eventually recognizes something from its map and everything is right in the world again. These actions are bundle adjustment, loop closure, and relocalization, respectively...and are all crucial pieces to aid the SLAM task and assist with autonomously navigating environments.

Researchers have been creatively exploring ways for the outer space robot (and his many forms of friends!) to more effectively and efficiently perform the SLAM task for more than 30 years. This thesis will explore a small branch of this field, monocular SLAM, which we describe in the sections following.

2.1.2 The Core Components of SLAM

Generally speaking, there are three core components to SLAM that help in defining the system as a whole, and is handy when comparing different SLAM offerings. Various techniques may specify specific sensors used, which can vary depending on the environment of the robot, the size, and so on. Examples of sensors include range sensors, such as LiDAR or SONAR, positioning sensors such as a GPS module or a magnetometer, visual sensors such as monocular or stereo cameras, and more. A good summary of these sensors and the three core elements of a SLAM system can be found in Figure 2.1.

SLAM systems will also vary in the method in which they process the sensor data. In the data processing step, sensor data is fused if necessary, and uncertainty is also handled. This process converts the raw sensed environment into something that the robot can work with in order to create a map and localize.

Lastly, SLAM systems will present some form of map representation of the area. Once again, this can vary greatly, and is dependent on the restrictions or desired outcomes you may have. Maps can be sparse or dense, and anywhere in between, and can be represented by a grid, connectivity graphs, and more.

SLAM systems are deeply varied and the limitations and attributes of the environment and proposed systems play a key role in deciding the SLAM approach for a specific task.

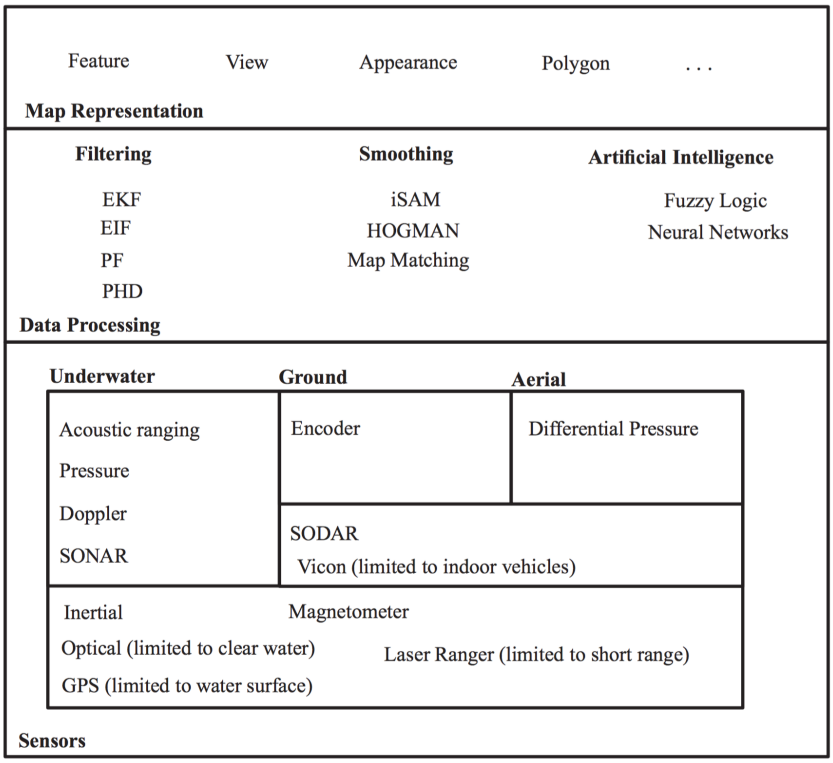


Figure 2.1: The three core components of SLAM. [43]

2.1.3 Visual SLAM

Visual SLAM (also known as vSLAM or VSLAM) is the specific problem of achieving SLAM through a visual exteroceptive input (i.e. a series of images produced from a camera). This is a particularly fascinating branch of the SLAM problem, as the minimum exteroceptive sensor requirement for successful visual SLAM is extremely cheap and lightweight—a single monocular camera. However, it is also often associated with systems that use stereo cameras and RGB-D cameras (such as the Kinect sensor) as input.

Visual SLAM emerged and is viewed as the real-time equivalent of a technique known as structure from motion (SfM). SLAM using a monocular camera was demonstrated in pivotal 2003 paper presenting a system known as MonoSLAM [10]. A paper coining the term Visual SLAM was published in 2005 and presented the fusion of a single camera and odometry readings from a ground robot to perform real-time SLAM [26]. The authors cited the rise in popularity and availability of cameras as a key driving factor, and the same still holds true recent innovations in visual SLAM are now driven by the availability of even smaller and more compact cameras, especially on smartphones and small UAVs, and sometimes, the combination of the two, both in academia [30] and the commercial sector [3]. It is also worth noting that stereo cameras appear to be surging in popularity in commercial smartphones and will possibly be a focus of upcoming papers as availability increases.

2.1.4 Monocular SLAM

As alluded to before, visual SLAM performed with a single monocular camera is known as monocular SLAM, with the first demonstration of autonomous MAV flight being demonstrated in [7]. Using a set of 2D images to estimate 3D space introduces a host of problems that are not nearly as prevalent when using stereo or RGB-D sensors, which provide a much better sense of the third dimension. Some challenges include non-smooth camera movement and blurry images, moving objects in a scene, and visually repetitive imagery [21].

Many developments in monocular SLAM and visual SLAM in general use a keyframe-based approach. In keyframe-based SLAM, a subset of frames (keyframes) are extracted to be used for SLAM calculations. Typically some sort of measure is employed to ensure that keyframes are describing unique areas in order to reduce overlap of frames or are particularly high quality frames (containing a large number of features, for example), which ultimately reduces map size and improves computation time.

2.1.5 Feature-Based vs. Direct and Sparse vs. Dense

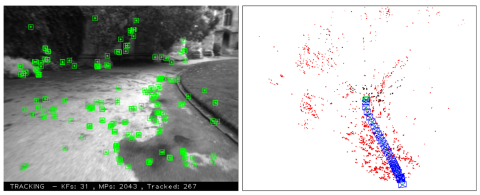
Traditionally, visual SLAM has produced sparse maps using feature-based techniques. However, with the advent of more powerful processors, recent years have seen the rise of dense monocular SLAM systems working directly with the image in localization and mapping, rather than first extracting features, as well as semi-direct approaches, which present a compromise

between the feature-based and direct.

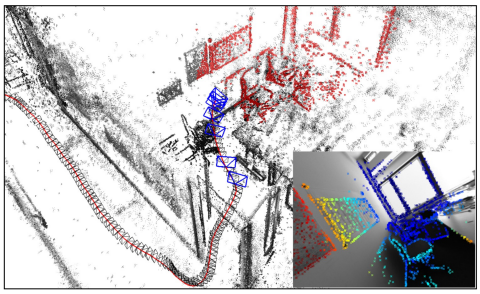
A quick comparison of the feature-based versus direct systems reveals that feature-based methods are faster than their direct counterparts and tend to be more flexible, but direct methods typically allow for a denser reconstruction and retain more complete information for decision-making [15]. It is important to clarify that a feature-based method can generate a dense map, and a direct method can generate a sparse map—feature-based does not equate to sparse just as direct does not equate to dense. No single approach works best in all situations, and you will notice below that a healthy variety of approaches are used in SLAM systems. Examples of each combination can be seen in Figure 2.2.

While sparse vs. dense is fairly straightforward and covers how much information is stored and how, feature-based and direct approaches are drastically different, and can affect the entire SLAM pipeline.

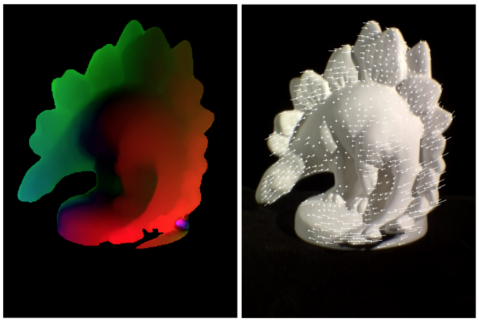
Feature-based monocular SLAM was demonstrated in 2003 in the form of MonoSLAM [10]. The next major jump in feature-based monocular SLAM came in 2007, in a paper where Klein and Murray present a system called PTAM (Parallel Tracking and Mapping) [27]. Possibly the biggest piece to come out of this paper was the parallelization of the tracking and mapping threads of SLAM, vastly increasing efficiency. PTAM has continued to be a leading technique in the field and is widely used as a benchmark for system comparisons, although it was originally designed for small environments. Direct approaches began emerging at the turn of the decade, with one of the most



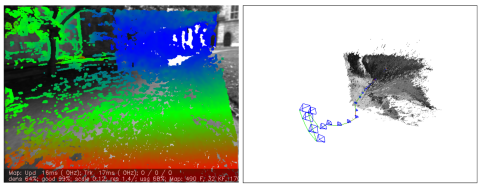
(a) Sparse, Feature-based [38]



(b) Sparse, Direct [16]



(c) Dense, Feature-based [47]



(d) [Semi-]Dense, Direct [17]

Figure 2.2: The various combinations of feature-based vs. direct and sparse vs. dense methods. Sparse, feature-based methods are have a long and popular history, with dense, direct approaches also garnering interest, but work has been done in the unlikely combination spaces of the sparse/direct and dense/feature-based methods as well.

notable being DTAM (Dense Tracking and Mapping) [40]. GPU improvements have made direct methods much more popular in recent years, although much work is yet to be done. In 2013, a novel approach known as SVO (Semi-dense Visual Odometry) [20], a semi-direct approach, was published, and in 2014, another direct method, LSD-SLAM was introduced, which focused on large-scale application, although works at smaller scales [17]. Lastly, one of the most recent developments has been a feature-based system known as ORB-SLAM, shown to be much faster and more resistant to viewpoint change than its PTAM predecessor [38].

2.1.6 Downward- vs Front-Facing Cameras

One distinction seen in approaches to monocular SLAM is the use of downward-facing cameras and front-facing cameras. As the name implies, a downward-facing camera has a perspective perpendicular to the ground plane. Outside of SLAM applications, there are many uses for a downward-facing camera, including aerial mapping, agricultural monitoring, and more. One thing to note is that by nature, maps generated using a downward-facing cameras are typically 2.5D due to the nature of their use/the overall flatness of the ground plane. On the opposite side, it is more straightforward to capture full 3D maps using front-facing cameras. Furthermore, front-facing cameras are more effective in indoor environments and situations in which the UAV is flying amongst obstacles, such as walls or trees. However, a front-facing camera will produce a less-detailed understanding of the ground below if it contains

uneven surfaces, so the ideal setup will vary and the appropriate configuration must be selected for the job.

2.1.7 Loop Closure and Relocalization

There are two other components to the SLAM ecosystem that need to be accounted for. SLAM systems are subject to drift, which increases over time, so longer operation typically means larger error. Furthermore, robots can lose localization and must find its location relative to the map it has been generating, or an existing map, potentially generated by another robot. To counter these issues, two tasks, loop closure and relocalization, can be performed.

Loop closure is used to detect when a robot has returned to an area that has already been mapped. Primarily this serves to correct error that has accumulated since the area was last visited. When a landmark or set of landmarks are identified at a rate above a certain threshold, the system will mark a loop closure. Previous measurements are then reconciled to fit within the new understanding of the robot's position.

Relocalization is the process of finding the robot's position on an existing map. Sudden viewpoint changes makes SLAM with MAVs particularly prone to loss of localization, and localization in monocular SLAM in general suffers in areas without textures and with occlusions that may occur. One simple, yet common approach to relocalization is backtracking to the point of last known localization.

2.1.8 Map Representation

At the center of a SLAM algorithm is the map in which it stores the interpreted data that it has collected. This representation of the area comes in many forms, falling into two key subsets, metric maps and topological maps, and increasingly more common—a hybrid between the two.

Metric maps, perhaps the more common of the two in the robotics community, describe the world using coordinate space, typically as 2D or 3D point clouds or occupancy grids. Point clouds are sets of points in metric space typically representing instances of obstacle detection derived from input sensors. Popular far beyond robotics, point clouds can be used for surface reconstruction, object mesh generation, and more. Raw point clouds can also be used for robotic navigation, but another very common setup is to feed sensor data into occupancy grids. Occupancy grids [13] are probabilistic grids of cells representing discrete areas in space, aiming to answer the question of 'is this cell of the grid occupied or free for navigation?' These grids are comprised of cells representing a physical area, each in charge of tracking the probability of the cell being occupied or free based on the input observations from sensors. Grids have been found to be very useful in path planning, providing a good sense of what areas can or cannot be explored, but typically do not scale nicely with outdoor environments, as one would have make the tradeoff between grid resolution (smaller resolution means better information for path planning) and computational complexity (too small of a resolution will increase the path planning space significantly). Interesting work furthering occupancy grids has

been the adaptation of quadtrees and octrees for the use of occupancy grids, thus reducing some of size issues that arise with a constant grid resolution. One key potential downfall of metric maps is the assumption of both an accurate pose reading and sensor readings, two values which tend to be big points of uncertainty particularly in visual SLAM.

Topological maps are useful for understanding the connectivity and relationships of an area, often seen in the form of connectivity graphs containing edges representing connections and their respective magnitudes. Topological maps are much more lightweight than their metric counterparts, as they store a much smaller set of data, which can be very advantageous in regards to efficient interactions with the map—key functionality such as path planning and loop closure detection can be quite quick, since the search complexity is reduced to a simple graph walk. Although topological maps are not dependent on knowing exact robot pose, this comes with a flip side—lack of granularity may be an issue, especially in having sufficient information for a robot to navigate the immediate area and dealing with obstacles.

In a hybrid of the two, one can leverage the advantage of the local granularity of metric maps, while also reaping the benefits of a global topology for its efficient and speedy lookups. All three of the SLAM systems that we investigate further in Section 2.1.9, make use of the advantages offered by a hybrid system.

2.1.9 SLAM Libraries

Three of the most recent open-sourced advancements in visual SLAM for single-entity systems are ORB-SLAM [38], SVO [20], and LSD-SLAM [17]. As previously mentioned, we aim to conceptually develop a framework generic enough to plug-and-play SLAM algorithms, and as such we evaluated several, both by studying and through some basic real-world trials with single camera systems.

2.1.9.1 SVO

SVO (Semi-direct Visual Odometry) provides a blend between traditional approaches and the more recent trend of direct methods, to great success in terms of speed. SVO only extracts features at keyframes, but is still able to perform motion estimation in the frames in between through a direct comparison. Additionally, it is able to filter out point outliers. Critically, it does not support loop closure out of the box. The original work was conducted on a MAV with a downward-facing camera, but we experimented using the front-facing camera of a Parrot AR.Drone 2.0, which runs at 30fps, a potential bottleneck in a system that runs around 300fps on a laptop. Unfortunately, we achieved less than stellar results in our short experimentation, although we are not sure this is entirely due to slow frame rate from the camera. Although never explicitly stated by the authors, we believe that SVO is likely more effective on downward-facing cameras, as opposed to front-facing, due to the motion estimation process not being robust to angular changes that are

more likely to occur in a front-facing camera.

2.1.9.2 LSD-SLAM

LSD-SLAM (Large-Scale Direct SLAM) is a direct monocular SLAM approach that generates a semi-dense map. LSD-SLAM attempts to extract a subset of the surface detected by reconstructing all points detected around distinctive areas; this is done through direct image pixels, rather than through feature detection. In our experimentation of applying LSD-SLAM to the Parrot, we had accurate results, but the frame rate was less than optimal. Although the algorithm is capable of running in real time, we found it to be consistently running at less than 30fps on our mid-level laptop, which would be an issue processing more than one stream in real time. You can see LSD-SLAM in action in Figure 2.3.

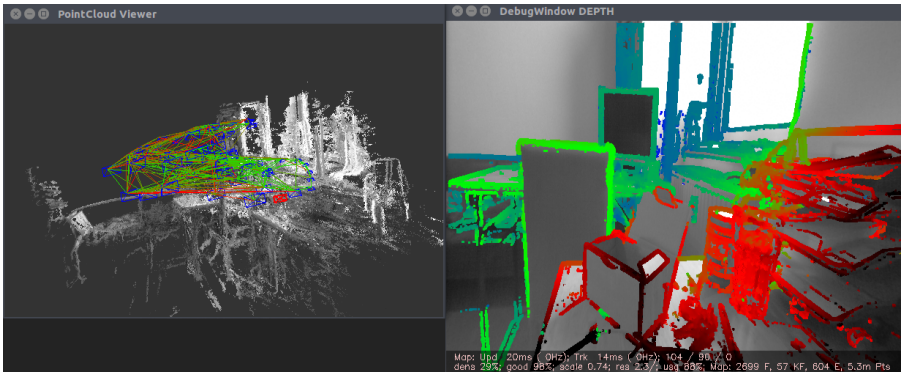


Figure 2.3: A sample LSD-SLAM run. Note the high level of detail in the reconstruction, which is useful for fine-grain navigation problems.

2.1.9.3 ORB-SLAM

ORB-SLAM is the most recent of the trio, and most notably is centered around ORB (Oriented FAST and Rotated BRIEF) features. These features are used in all SLAM tasks—mapping, localization, loop closure, and relocalization, making it very spatially and computationally very efficient, and are able to run loop closure detection on each keyframe with limited performance degradation even for large loops. In our experimentation with the Parrot, ORB-SLAM ran very efficiently and effectively given the camera of the Parrot. We found this to be the best option for us. Later testing showed that multiple streams were easily handled by the single ground control station, and map merging can be efficiently calculated using ORB-SLAM's sparse map. An example of ORB-SLAM and its sparse map can be seen in Figure 2.4.

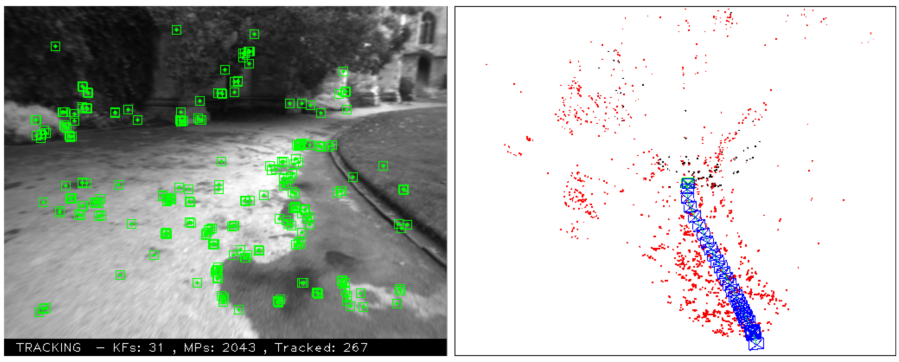


Figure 2.4: A sample ORB-SLAM run. Note the ORB feature extraction and sparse map.

2.1.9.4 Honorable Mention: DSO

Lastly, a recent development at the beginning of our work, DSO (Direct Sparse Odometry) [16], was just published, which is different from the three aforementioned methods in that it is both direct and sparse. The results are promising, but the source code was not released until after work was underway. Upon preliminary review, it seems as though this may be a good candidate if our work was to expand to using different SLAM algorithms.

2.2 MAVs

An unmanned aerial vehicle (UAV) is a term for any aerial vehicle without a human pilot on the aircraft. UAVs come in many shapes and sizes, from plane-like jet-propelled systems to tiny multi-rotor systems that can fit in the palm of a hand. Over the years, advances in miniaturization of sensors and systems have allowed for ever smaller UAVs.

A micro aerial vehicle, also known as a MAV or μUAV, is a particular class of UAVs. Although the classification varies depending on the source, it is generally a term given to fairly small UAVs with a size of 0.1-0.5m in length and 0.1-0.5 kg in mass [34]. The micro denotation is often used as a blanket term for small UAVs, although the nano and even pico prefixes have been used to represent UAVs smaller than the micro classification, and the mini prefix has been used for UAVs slightly larger.

Common UAV wing structures include flapping wing craft (i.e. bird-like

wings), fixed wing craft (i.e. airplanes), and rotary wing craft (i.e. helicopters). Rotary wing craft are notable for their ability to hold position and make concise pose adjustments in crowded scenarios such as indoor flight, whereas fixed wing craft boast significantly longer flight time. Unfortunately, flapping wing and fixed wing UAVs have not seen nearly the growth in popularity as their rotary wing counterparts.

Our particular focus will be with quadrotors, a form of rotary wing craft with four rotors, which has exploded in popularity in recent years. One primary advantage of quadrotors (and other multi-rotors as well) is a simplified path planning system. Although in theory, a quadrotor must handle 6 degrees of freedom (DOF)—$(x, y, z, \phi, \theta, \psi)$, with acceptable error, the problem can be reduced to 4DOF (x, y, z, ψ) [23].

2.2.1 Control Paradigms

As mentioned, UAVs have no human pilot on-board to provide control. However, the term does not imply autonomous flight a human-controlled RC helicopter is an example of a UAV. The control of a UAV can be delegated to an on-board system, or to an off-board system such as a computer or a remote control. One common scenario is a mixture of the two, where basic flight mechanics and stabilization are performed on-board, but complex computations for autonomous flight are performed off-board on a ground control station (GCS). Many MAVs are capable of self-autonomy with fully-functional on-board processors. However, these systems are inherently larger and require

more power in order to support the processing. This will be further discussed in 2.3.1.

In autonomous flight, there are two key aspects of controlling a quadrotor, split into lower-level and higher-level control. At the lower level, basic flight stability is achieved through a very quick (roughly 100–1000Hz) closed-loop control that adjusts attitude. Nearly all commercially available quadrotors provide a basic form of attitude stabilization such as this. More complex systems can use an inertial measurement unit (IMU)—typically comprised of gyroscopes, accelerometers, and possibly magnetometers—which provides a good odometry estimation, position hold via GPS, and further refined stabilization and position hold in GPS-denied areas through visual odometry techniques. At the higher level, the aforementioned issues are abstracted and stability is assumed for the most part, allowing for a focus on more macro tasks, such as relative movement, path planning, SLAM, and inter-UAV interaction. This thesis will focus primarily on the higher-level, although considerations must be made for the lower-level.

2.2.2 MAVs in Research

MAVs have been used both manually and autonomously in a broad variety of applications, and the list is constantly growing as new algorithms are developed for these. A few applications of MAVs include filming and aerial photography, search and rescue tasks, and maintenance investigations. This flexible nature has been noticed by the robotics community and MAVs have

been used in robotics research for many years, with increasing frequency driven by increased popularity and availability. Research is conducted in a variety of environments (indoor and outdoor, known and unknown, obstacle-free and obstacle-dense, etc), and may or may not include work with physical UAVs (some work is done strictly or primarily in simulation).

2.2.3 Motion Capture Systems

Much of the work done on multi-robot and in particular multi-UAV systems is powered by a motion-capture system, such as the Vicon Motion Capture System. Using the Vicon system as an example, it is capable of running at 375Hz and providing pose error of less than 50μm, which is extraordinarily precise relative to the errors that a strictly on-board system would be able to achieve [34]. While motion capture systems provide excellent and accurate positioning information and allow for more complex algorithms to be performed, they are location limited (the UAVs are restricted to a room or subset of rooms) and require previous setup. Furthermore, these systems are extremely costly. Nonetheless, systems such as these have pioneered considerable work on path planning and execution, as well as numerous other tasks, and are by all means worthy of mention. One example of work that leverages a motion capture system to perform more advanced actions can be found in [33].

2.2.4 Commonly Used MAVs

There are many existing MAV platforms that have been used in UAV research, each with their own merits. To provide a frame of context, we will investigate several systems and their possible uses, although some systems may not fit within the needs of this thesis.

Ascending Technologies (AscTec) UAVs are widely used in UAV research due to their mature product and commercial support. Notable users include UPenn's Vijay Kumar lab, JPL [8] and more. These UAVs are extremely extensible and allow a fine-grain control of the system [34]. Additionally, base flight time is more than 20 minutes. However, their UAVs are available through consultation only and do not fit our criteria of being low-cost, with the cheapest solutions available pricing in at several thousand dollars each. The two most commonly used systems for vision research include the Pelican, which is capable of carrying and powering a Kinect sensor system, and the Hummingbird, which is a slightly smaller UAV.

The Parrot AR.Drone series first emerged in 2010 and was later refined in 2012, and has been extremely popular amongst MAV researchers due its low cost ($200 USD at time of writing) and availability—it can be found in many toy and department stores, as well as online. Several libraries have been developed around its SDK, including an interface through ROS (Robot Operating System) and Paparazzi, two popular robot middlewares, as well as through many different programming languages. The flight time is shorter than the AscTec offerings, coming in somewhere between 10 and 15 minutes,

and the MAV features front-facing and downward-facing cameras as well as an ultrasonic height sensor. Communication is via a direct WiFi connection between the AR.Drone and the controller, although solutions exist for connecting the AR.Drone to a central WiFi, which allows for control of multiple MAVs simultaneously.

The Crazyflie 2.0 is another MAV used in research. Classified by some as a nano-UAV (a degree smaller than a MAV), the Crazyflie is a fully open-source option, as compared to the two aforementioned options. However, the Crazyflie does not ship natively with a camera, although the addition is possible as shown in [11], although this results in a flight time of less than five minutes—not ideal for SLAM research.

This is by no means a comprehensive list, but represents a handful of common MAVs used in visual SLAM research.

2.3 Multi-UAV SLAM

Visual SLAM has matured greatly in the past few years with the development of many new techniques. While single-camera/single-robot SLAM is far from full maturity, there is another field that has not received as much focus: collaborative visual SLAM, in particular as applied to UAVs. There are several challenges introduced by the nature of this problem, including the communication, sharing, distribution, and processing of data.

2.3.1 Data Communication

One key issue to address in multi-robot systems is communication. A quick primer was presented in 2.2.1, where it was mentioned that UAV to GCS communication can come in many forms, and the UAV to GCS communication methodology will depend on the aim of the work and on the environment. Furthermore, in multi-UAV systems, inter-UAV communication can be used to extend network reach. In the event that a UAV is out of range of the GCS, communication could be relayed through an intermediary UAV.

Additionally, bandwidth of the network must be considered. Key communication payloads range from raw measurements (from an IMU and/or camera) to calculated maps/poses. Determining the form in which data is transmitted is some function of the task and capabilities of the system. In visual SLAM, a video stream, even after compression, can take a considerable amount of bandwidth. By processing the video on-board, bandwidth is generally reduced (although this may not be the case in dense mapping), but this requires a powerful processor on-board the system, consuming weight and energy that could otherwise be used to limit the size and increase the flight time of a quadrotor, and additionally raising the cost of the UAV. Instead, when processing the data off-board, more advanced computers and GPUs can be used to swiftly perform these calculations, but then the UAV is presented with commands in response to data collected many milliseconds previously, which can cause issues such as oscillation [18] Investigation into the benefits and drawbacks to on-board and off-board systems can be found in [12] and

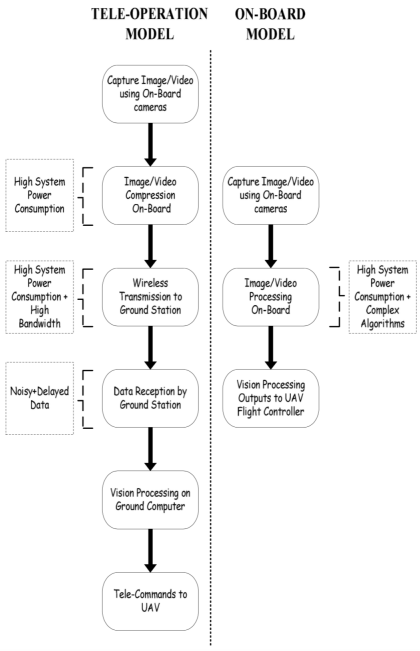


Figure 2.5: A standard vision processing model. [12]

are also represent in Figure 2.5.

One of the hottest areas in commercial UAVs right now is first person view (FPV) UAV flying and racing. In FPV flying, live video (and potentially other telemetry and system data) from the UAV is streamed to a user's control station, which is increasingly a smartphone device via Wi-Fi for the casual user, due to the pervasiveness of smartphones today. Antennas from a broad range of the spectrum are used however, if UAV to computer communication is desired using a non-Wi-Fi supported band, an antenna needs to be used for the computer (example: crazyflie). The erupting popularity of FPV flying is a driving factor behind performing visual SLAM on UAVs, as it increases the availability and decreases price for camera-enabled UAVs.

Most commercially available UAVs are focused on the UAV and GCS communication experience, providing a direct link between the two, making inter-UAV communication difficult out of the box, but work has been done in connecting such systems to a single Wi-Fi network, or through other methods, depending on the hardware communication method chosen.

2.3.2 Decision Centralization

Centralization (or lack thereof) in multi-robot systems is also very important consideration tying in with communication. The centralization of a system is referring to the location in which the core of the decision-making processing is performed. A centralized system relies on a specific entity, whether that be a UAV or a GCS, to process incoming data from the other UAVs in the system and make decisions such as task allotment or establishing goal points in exploration. To the opposite end of the spectrum, a decentralized system performs task computations on multiple or all UAVs. Although centralized systems reduce complexity, decentralized systems are more resistant to failure and can be leveraged to accomplish tasks like explore outside of communication range of a centralized system. A key benefit of using a centralized GCS is that UAVs in the system will require less computational power, as computations can be done off-board. Of particular interest, Figure 2.6 presents an outline for a multi-entity SLAM system. Depending on where the lines of communication are drawn, however, can yield drastically different network structures and subsequent effects.

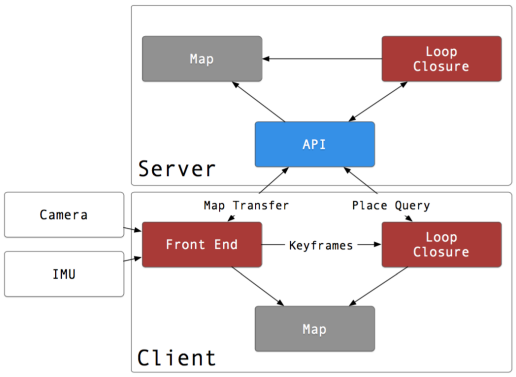


Figure 2.6: An example distributed SLAM design. In this example, the client is considered a UAV with on-board processing. Visual odometry, mapping, and loop closure are performed on-board, but the server (the GCS) also processes loop closures and merges maps. [35] It's important to note, however, that it is trivial to add a layer of communication between the inputs (camera/IMU to the right) and the processing client, creating an off-board architecture.

2.3.3 Robot Middleware

In order to communicate between UAVs, a robot middleware system may be deployed. Robot middleware provides some form of standardized communication between UAVs, a GCS, and any other devices in the system. There are several populare middlewares available, one of the most popular being ROS (Robot Operating System). Nearly all of the aforementioned open source visual SLAM systems are written to be compatible with ROS out of the box, making it a prime choice for use.

Another system geared specifically towards UAVs is the open source UAV software Paparazzi. A system of connecting multiple commercially available MAVs capable of autonomous flight using Paparazzi is proposed in [41]. However, the focus is primarily with the assumption of GPS and does not

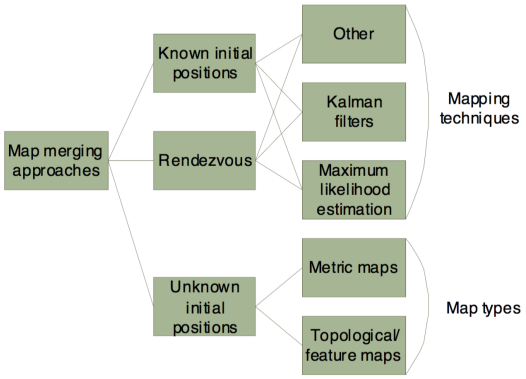


Figure 2.7: Map merging approaches. [4]

conduct any visual SLAM. Furthermore, Paparazzi is centered around the concept that each UAV contains a Paparazzi-compatible board, and while it is still possible to interface with UAVs that are not immediately compatible, it adds a layer of obscurity.

2.3.4 Map Merging Techniques

In single-robot SLAM, a robot navigates and maps an area, performing loop closure when needed and so on, but for multi-robot SLAM, some alterations are made to that cycle. Particularly, in multi-robot SLAM, generated maps are consolidated as frequently as possible, and this may occur either in a centralized or decentralized fashion, but ultimately the robot should perform navigation and path planning on as complete of a map as possible. The main goal of map merging is to apply transformations to the local maps of each robot, keeping in mind that relative starting positions may be known or unknown. A good summary of the merging space can be found in [4], and a

good visual is provided 2.7.

If starting positions are known, map merging can be simplified, as the transformation matrix between any given robots is initially known. However, drift can occur, meaning that a system must be robust to somewhat unknown relative positions anyway. Another case is that a robot comes across another robot in the process of mapping, known as a rendezvous. From this, a robot has a solid reference from which to adjust its relative position. Note however, that visual rendezvous, especially with slightly-less-than-ideally stable MAVs can be a difficult task, and relative position even when another MAV is detected, can be difficult to precisely determine. In exploration of an unknown environment, robots may not have a sense of their relative position until visiting an area that another robot has already visited. When a robot senses a landmark that it believes to be one found in another robot's map, we are provided with a reference in which to apply a transformation. A series of re-recognized landmarks can provide a fairly accurate transformation in order to merge maps. However, this is dependent on the local maps of the robots being shared in real time, as well as landmark comparison being performed in real time. This more or less describes the submap matching problem, in which pieces of one map are compared to pieces of another, and the transformation produced is the one in which reduces error the most.

2.3.5 Collaborative Visual SLAM Using UAVs

In recent years, a few works have been presented featuring collaborative visual SLAM using UAVs. In 2010, a feature-based system was presented using EKF and nonlinear H∞ filters, but was only performed in simulation [39]. In 2013, a team of 2–3 MAVs conduct collaborative visual SLAM using actual MAVs. Using a a keyframe feature-based approach, the authors are able to successfully conduct mapping of a moderately-sized outdoor area and achieve very good results. The proposed method uses downward facing cameras for mapping and localization, and in this work, the authors opted to use AscTec MAVs for testing [19].

2.4 System Requirements and Challenges

The goal for the hardware and software of the system is for both to be straightforward and available enough such that an ordinary individual with limited technical knowledge can create his/her own copy of the system relatively quickly and at a low-cost to said individual, making the system accessible to researchers and educators alike. To frame this more specifically, we aim to present a minimum multi-UAV system (two UAVs) for US $500 or less, and thus, the cost of an individual UAV in the system should be less than $250. Furthermore, it should require very little or no assembly, and very little software configuration, especially regarding the ability to create a network of UAVs. Furthermore, the emphasis of this thesis is on open source SLAM algorithms, such that the individual is able to download and modify their own

version and contribute back to the community. Lastly, motion capture systems are often used in UAV testbeds in order to receive accurate global positioning of each of the UAVs in the system, but our main focus is SLAM in unknown environments, and as such, we will not be incorporating a motion capture system.

By establishing these requirements and the reasoning behind them, we establish a multi-UAV system capable of performing complex tasks to be used in a variety of applications (search and rescue, teaching, etc.) with a very low barrier to entry.

Chapter 3

System Design

As previously mentioned, the objective of this thesis is to develop a multi-MAV system capable of supporting autonomous exploration and navigation. In this chapter we will present our overall system architecture and will delve into the design for each of the subsystems, as well explore the drivers and design decisions behind these subsystems.

3.1 Overall System Architecture

At a very primitive level, the system architecture consists of six main components:

1. System actors, which are either UAVs, simulated UAVs, or video streams
2. A coordination server for reading inputs and producing ORB-SLAM outputs
3. A modified version of the ORB_SLAM2 library
4. A custom Octomap server that reads ORB-SLAM outputs and turns them into actionable maps

5. A control server that accepts maps and coordinates both individual and group movement

6. Middleware and communication infrastructure

Throughout this chapter, the composite of these subsystems will be referred to as *the system*, which represents the operation in its entirety. This high-level design is visualized in Figure 3.1. Each section of this chapter will dive deeper into the individual components, shedding more light on how the system interacts.

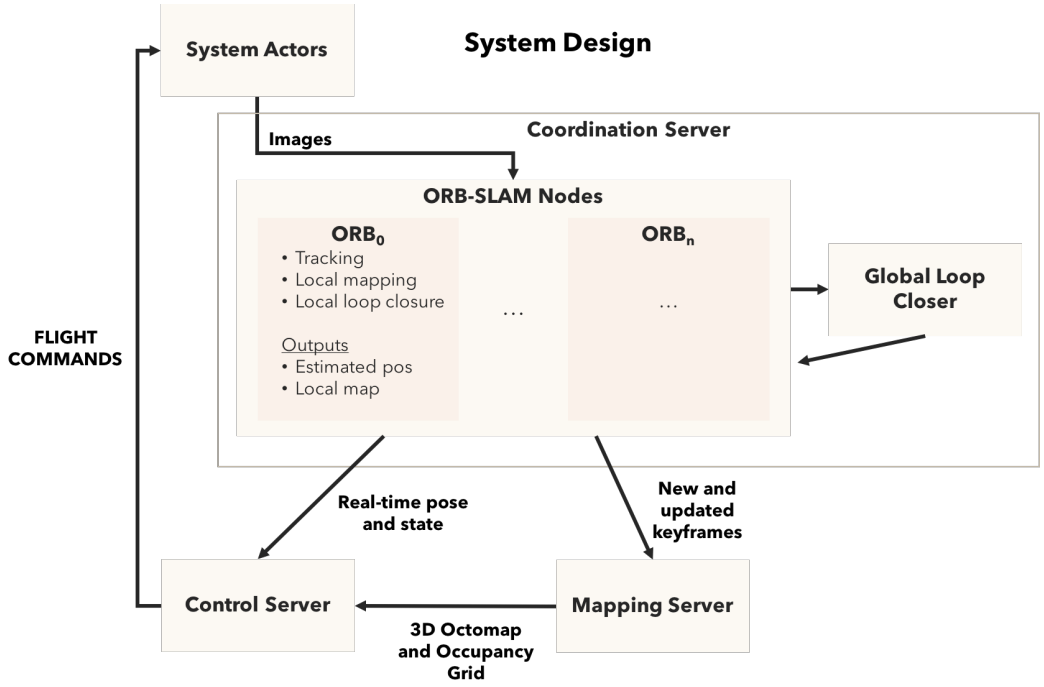


Figure 3.1: High-level System Architecture.

3.2 Middleware and Communication Infrastructure

We will begin our deep dive into the inner workings of the system with one of our most central pieces—the middleware and communication infrastructure. Ultimately, we elected to use ROS as a middleware for communication between the different components. Although ROS is focused more generally on robotics, competitors such as Paparazzi are not nearly as popular—all the SLAM systems we evaluated supported ROS out of the box. At a basic level, ROS acts as a data broker between our UAVs and our ground station, transforming transmitted video feeds and relaying commands to the UAVs. ROS promotes modular design, where different nodes complete different tasks. Thus, we have developed or incorporated several nodes that comprise our system that each serve specific tasks—each of which will be introduced in sections below.

In regards to group communication, we have designed a centralized system, where each UAV speaks to a central ground station, which creates and stores maps for each UAV and sends commands to the UAV. We recognize that a distributed approach is more robust to connectivity issues, but it requires increased processing power on the UAVs themselves, which does not align with our goal of using low-cost and small MAVs. Our architecture brings us back to Figure 2.6, but in a world where the server and client are colocated at the GCS, and the inputs and control responses are sent over the wire, rather than calculated onboard the UAV.

3.2.1 A Primer on ROS

Structurally, ROS allows us to connect a set of heterogeneous nodes in a lowly-coupled fashion, harnessing its robust publish/subscribe (pubsub) messaging system. You can have one node for each physical machine, or more likely—several nodes per machine, each handling a discrete subsystem. A simple remote-control robot could easily be composed of a joystick node in charge of reading raw device inputs, a control node that accepts inputs and produces instructions for the robot, a node on the robot converting raw camera images to ros messages, and a node for processing telemetry and logs and feeding them to the user. All of these are managed centrally by a master service that deals with coordination and ensuring messages are distributed around.

The pubsub messaging—arguably the core of ROS component design and a great differentiator—both enables and encourages systems to be built very modular and language-agnostic (for example, our system is a mixture of C++ modules for the ORB_SLAM2 and raw image handling components and Python modules for some of the control system pieces). Messaging is done via ROS messages, a format defined for inter-node communication. ROS messages are defined using a simple description language that allows specification of sets of field names and types in *msg* files. These files support the basic types you might expect (bools, ints, timestamps, etc.), as well as other messages. ROS also has well-established packages with many common message types used in robotics—for example, the geometry_msgs package contains messages for a 2D

or 3D point, quaternions, and twist (linear and angular velocities combined), among others. Other packages exist for common sensor data, navigation data, and more. The result is a rich ecosystem of fully interoperable open source libraries that build on each other.

These messages are published by nodes to topics—a channel for messages of the same time to be organized around. Topics use a slash notation for grouping related topics, for example a */camera/image_raw* topic transporting *sensor_msgs/Image* messages and */camera/camera_info* topic transporting corresponding *sensor_msgs/CameraInfo* messages, describing data from the same camera.

Published messages are pushed out onto topics to which any ROS node can subscribe. On the consumer side, messages are then queued to be processed predominately asynchronously, although ROS also supports synchronous *services* as well.

This flexible approach to publishing and consuming means that you can get very interesting publishers and consumers. A consumer can be wired up to record any/all topic data from a live session and then replay that data back exactly as it was experienced the first time for repeatability—these are called ROS bags. When replayed, these ROS bags become the publishers, and the system you are testing will never know the difference. This functionality is the reason we can transparently swap out real sensor data (monocular camera feeds) from actual drones with recordings from previous runs, and through some conversion, we can even play videos through ROS bags that were never

connected to a ROS system.

Further material on ROS available on the wiki page for the project, located at `http://wiki.ros.org/`.

We will further explore how we fully utilize these features in our system design in the subsequent sections. ROS is a tremendously powerful tool, and the wealth of libraries available made piecing components together very simple, from hardware-specific drivers to image processing tools to our SLAM library to simulators to visualization systems to mapping systems.

3.2.2 Decentralized Design

Initially, an attempt was made to develop a predominantly decentralized system. In this design, there would be a one-to-one mapping between input video streams and discrete ROS processing nodes. This is good in theory, but all communication would have to be routed through ROS, which generates a massive amount of traffic, rather than the simpler inter-thread communication which has much less network overhead. The advantage with the distributed design is that processing power for any individual node does not need to be that great, and to scale, one simply needs to add more machines to handle the additional load. However, this design fails to meets the needs of bundle adjustment and other similar techniques, which are especially prevalent and important in monocular SLAM, where scale is estimated and accuracy tends to dip below their depth-based SLAM counterparts. In monocular SLAM, these adjustments happen often and effect a large number of keyframes, although

both of the aforementioned occur with different frequencies and magnitudes vary from implementation to implementation.

Our work with ORB-SLAM found that although it uses an extremely sparse representation, it is not practical to be constantly transmitting all of the information necessary to conduct efficient and effective SLAM across multiple physical machines. In this light, we chose the aforementioned more centralized system as described in our overall system architecture. We found that a sufficiently powerful and multi-threaded machine (our primary GCS was a mid-level 2014 laptop) could handle several UAVs in real-time with ease. Note that this causes an issue of scalability—the bottleneck of our system is in the single ground control station in charge of performing SLAM on the incoming video streams, even if all other processes (visualization, control, etc) are offloaded to other machines. The communication handoffs will be further discussed in their respective components.

3.3 System Actors

In our work, three forms of actors were used for development and testing, a hardware platform, a simulation platform, and played-back datasets. For the vast majority of the functionality in our proposed SLAM system is that our actors represent some form of platform that publishes a video stream to a ROS topic, which then can be grabbed by our coordination server. Further along in this thesis, we will also discuss actors accepting inputs, which only applies to our live hardware and simulation platforms, which listen to

specific ROS topics for ROS Twist messages. The actors as they interact with our overall system is presented in Figure 3.2.

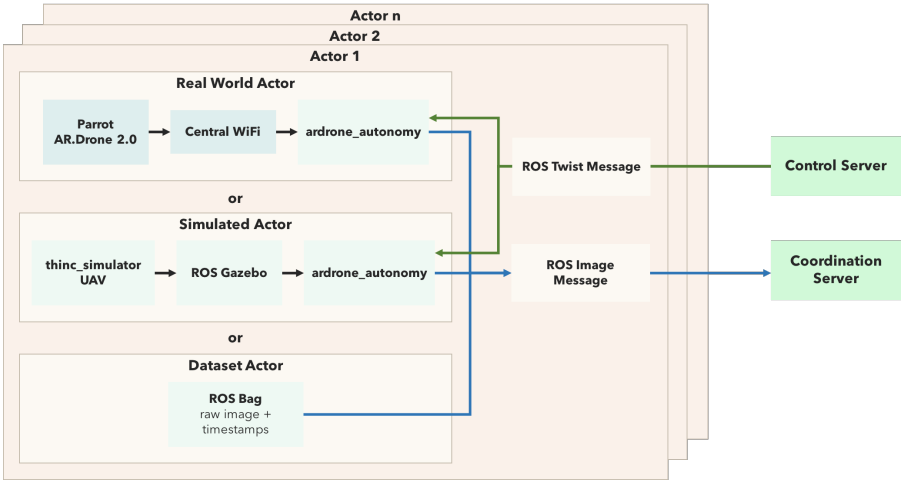


Figure 3.2: This diagram shows the many inputs feeding into the coordination server for processing and re-distribution to corresponding ORB-SLAM nodes. Each actor should be designed to emit ROS Image messages in order to provide a common format for the coordination server to understand.

3.3.1 Hardware Platform

We selected the Parrot AR.Drone 2.0 as the target platform for hardware testing. It has been chosen for its stability and widespread adoption with a relatively low price point. However, it should be noted that the software and theory should be transferable to any set of connected MAVs with on-board cameras and IMUs, although with varying degrees of simplicity.

In order to communicate, the UAVs are slightly modified to connect to the same WiFi network. As with most common commercial WiFi UAVs available, the Parrot UAVs are set up to generate a WiFi network to which the

controller then connects. However, this can be altered using openly available code [5] such that the UAV connects to a central WiFi network—the same network to which our ground control station is connected. Our testing was performed at RIT, where a central WiFi network is pervasive throughout any building one might be in, which solves the issue of a minimal range when dealing with a single WiFi access point.

These UAVs do come with several sensors outside of the necessary front-facing camera, worth noting as cheaper options may not have them—notably an IMU, altimeter, and downward-facing camera for position hold. These inputs were processed onboard by the UAV in its high-speed inner control loop, and admittedly, remote flight is difficult to accomplish without this assistance. Indeed, even at the nearly the cheapest tier (i.e. the $35 Cheerson CX-10WD), some sort of baseline inner control loop processing and even altitude hold is being performed, so this is not inherently a breaking assumption to make. Another potential advantage of choosing a larger UAV such as the AR.Drone 2.0 is that by being a physically larger UAV, it provides some stability in flight patterns as well—smaller UAVs are more responsive, and thus more difficult to control and also are more likely to produce blurrier video streams as a consequence.

To capture and bring the feeds from our UAVs into the ROS ecosystem, we use a ROS driver for the AR.Drone 2.0, ardrone_autonomy [6]. This leverages the official Parrot SDK to process the UAV data and then broadcasts it all out to ROS topics for use in our system. These topics include IMU data,

either the front- or downward-facing camera feed (only one can be emitted at a time), as well as estimated navigation data. The complete set and format of data produced can be found on the ardrone_autonomy website.

For our work, we use the front-facing camera exclusively (although the AR.Drone 2.0 uses the downward-facing implicitly for things like position hold). The image stream is a grayscale 360p (640x360) feed. Although technically an HD (720p) stream is available from the MAV, this quadruples image size, slowing the whole pipeline down.

For SLAM, we also need to do image calibration, which allows the image processing to act on a more uniform and understandable image. This is done through the ROS camera_calibration package, which requires you move the camera around in front of a calibration object (i.e. a chess board), and then returns the estimated camera matrix—a set of numbers that define how the camera behaves. This matrix can then be converted to various calibration formats used by different image processing/SLAM systems. A great resource for understanding the camera matrix is a Kyle Simek's blog [44]. Our camera matrix values can be found in the code linked in Appendix 1.

3.3.2 Simulation Platform

Of course, a top priority for us while testing and developing is not to destroy these beautiful UAVs that we have selected, so we must adopt some sort of simulation testbed. For this, we are using Gazebo as our simulated 3D environment, topped off with thinc_simulator [29], an update and adaptation

of the tum_simulator [25], a package used for simulating the AR.Drone 2.0 in Gazebo. With this, we are able to simulate multiple UAVs emulating our real-world Parrots, flying around in a semi-realistic scene. As previously mentioned, both ROS and the AR.Drone 2.0 are very popular in academic UAV research, and as such, systems like this exist, making simulation fairly straightforward and somewhat realistic. Unfortunately, in practice, we found it difficult to find good simulation environments for realistic monocular SLAM—notably, any simulation environment we tested resulted in the system aggressively loop closing due to extreme uniformity of the input images. An example of this simulation environment can be seen in Figure 3.3.

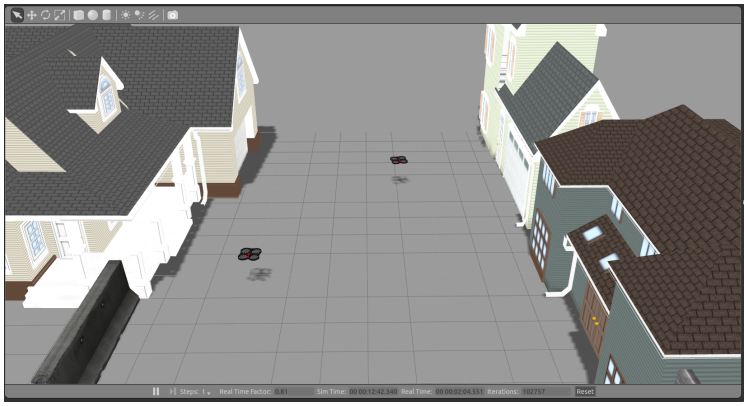


Figure 3.3: Example test environment in Gazebo, featuring two simulated AR.Drone 2.0s.

3.3.3 Datasets

Both the hardware and simulation platforms are useful in testing the system end-to-end, from UAV to input image, to SLAM, to control system. . . rinse and repeat. However, during development and for benchmarking evaluation,

it is also useful to have simple datasets containing just the visual feed, particularly when not needing to test the control aspect of the system - dataset playback is much quicker and repeatable as compared to our hardware and simulation platforms.

We used two datasets primarily during the development and evaluation of our work:

1. The EuRoC MAV Dataset [9]
2. The TUM Monocular Visual Odometry Dataset [14]
3. A custom dataset collected in the RIT Golisano building

3.3.3.1 EuRoC MAV Dataset

The EuRoC dataset is a very exciting dataset for our uses, as it contains sequences filmed by actual flying MAVs, whereas many monocular datasets are simply handheld. In total, it contains 11 sequences across three scenes, each shot at 20fps, and also includes other data such as IMU readings, ground truth pose, and ground truth scene information (for evaluation of the generated map if desired). Since it contains several groupings of sequences covering similar areas, we use this to simulate multiple UAVs flying around at the same time, although the original recordings were done in series. We focused on one scene in particular, the Machine Hall (MH) sequences, which has five sequences exploring a large indoor space with large machines inside. This proved to be the most dynamic backdrop for testing. The dataset has two "easy"

(as in, easy-to-process) sequences, labeled MH_01_easy and MH_02_easy, one "medium" difficulty sequence, MH_03_medium, and two "difficult" sequences, MH_04_difficult and MH_05_difficult. Difficulty is assigned by the authors due to things such as rapid rotations, dark scenes, and lack of texture in the scene, all of which make monocular tracking more difficult. A part of the room can be viewed in Figure 3.4 to give a sense of the scale and challenges. Between the sequences, a good portion of the machine hall is mapped and traversed, with each sequence varying from 100 to 180 seconds in duration and travel between 80 and 130 meters. Exact information can be seen in Table 3.1.

Figure 3.4: The main section of the room explored by the five EuRoC Machine Hall sequences. An upstairs area to the right of the camera that a handful of the sequences explore is not visible from this view. [9]

Name	Length	Duration	Avg. Vel	Angular Vel.	Note
MH_01_easy	80.6m	182s	0.44ms^{-1}	0.22rad s^{-1}	Good texture;bright scene
MH_02_easy	73.5m	150s	0.49ms^{-1}	0.21rad s^{-1}	Good texture;bright scene
MH_03_medium	130.9m	132s	0.99ms^{-1}	0.29rad s^{-1}	Fast motion;bright scene
MH_04_difficult	91.7m	99s	0.93ms^{-1}	0.24rad s^{-1}	Fast motion;dark scene
MH_05_difficult	97.6m	111s	0.88ms^{-1}	0.21rad s^{-1}	Fast motion;dark scene

Table 3.1: The machine hall sequence metadata, copied from the original paper. Further information can be found in the original publication. [9]

3.3.3.2 TUM Monocular Visual Odometry Dataset

This dataset contains 50 handheld, black and white video sequences of varying length and environments. It is capable of calculating overall system accuracy by focusing on errors at loop closure time (i.e. drift). While the dataset does not contain ground truth pose to frame mappings, making it not very usable for the evaluation aspect of our work, it played a vital role in the development process, as it supplies groupings of sequences in a variety of environments, both indoors and out. One set of sequences that we frequently tested with were sequences 30 through 32, in which the camera explores an outdoor courtyard of moderate scale and loops back on itself. Other groups of sequences were experimented with, but with less success.

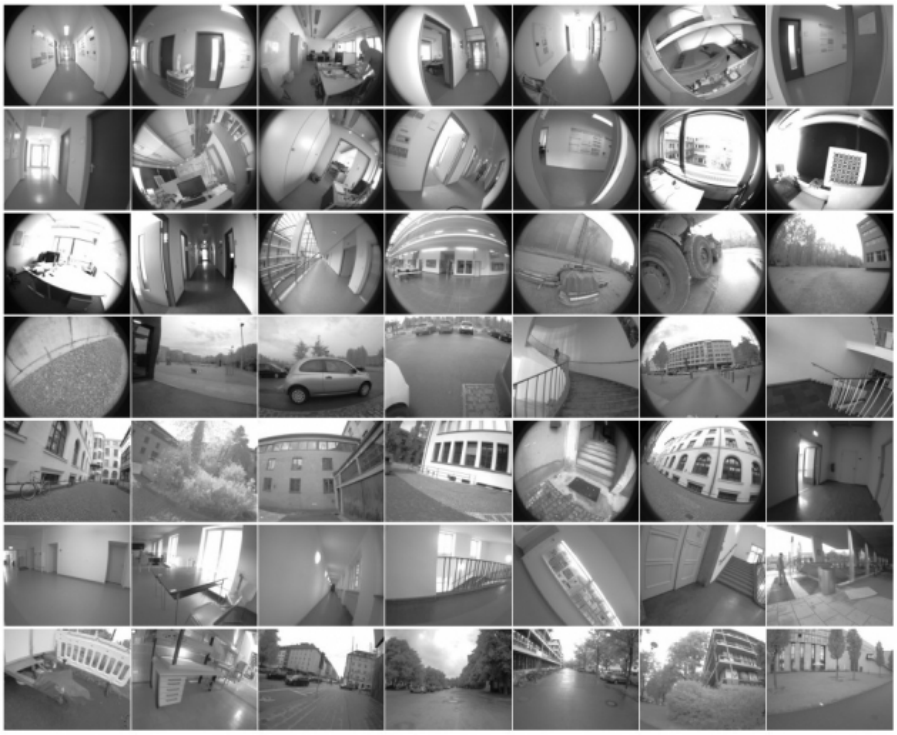


Figure 3.5: Thumbnails of the TUM monocular dataset sequences. Thumbnails are in sequential order, so sequences 30 through 32 start at the second thumbnail in the fifth row. [14]

3.3.3.3 RIT Dataset

Our collected dataset consist of roughly 20 ROS bags of various attempts to fly around the entirety of the third floor of the RIT Golisano building, where most of our real-world testing was ultimately performed. Three primary approaches were taken when collecting these bags.

1. Attempted autonomous flights in which the full system was being tested,
2. Manual flights, and

3. By running around, pushing a hovering UAV through the halls in attempts to simulate what actual flight looks like

The primary purpose behind all three of these approaches is to provide a dataset somewhat looking like our actual testing environment, while also presenting the challenge of a camera mounted to an actual, in-flight UAV, which may be shakier and blurrier than a camera mounted to a cart and walked around. Although the ideal is for all flight ROS bags to be of type one, our system was not consistent enough to provide a large enough dataset. The second preference would be of type two, where at least the UAV is responding to flight inputs. However, this is susceptible to poor piloting given the confined space of office hallways, and it was also difficult to get large enough datasets this way. Although the third option is perhaps a bit laughable in theory, it was a necessity, as long-duration flights covering the entire floor plan required the stability (of the system) provided by this technique.

The RIT dataset is used to demonstrate and examine our hardware proof of concept, and perhaps its most valuable asset is that it provides a more realistic set of sequences—including artifacts and motion blur, intermittent video connectivity loss, and a lower quality and framerate camera than provided by the aforementioned datasets. These type of data quality issues are at the core of adapting monocular SLAM systems for lower quality UAVs, which simply do not have the high-quality cameras that are used in benchmark datasets.

Figure 3.6 provides a floor plan and a few sample images from the data collected. Some interesting challenges notable to visual SLAM and this environment in general should be called out. First, note that rooms with glass walls, such as the one pictured in the east hallway, can be particularly challenging to visual-based SLAM when the lights in the room are on, as seen in the photo. Another challenging aspect is repeated architecture—notice that the east and west hallways look nearly identical at first glance. Since monocular SLAM is innately uncertain and loop closure must be at least somewhat aggressive in order to be effective, repeating patterns can cause confusion. Luckily, in the real world, varying texture and layout differences provided enough context that we did not face trouble here, but this was a large problem in simulations. The image feed came through generally well lit—not too over- or underexposed. Some sequences include moments where the MAV is stuck against a wall and receives manual assistance to move away.

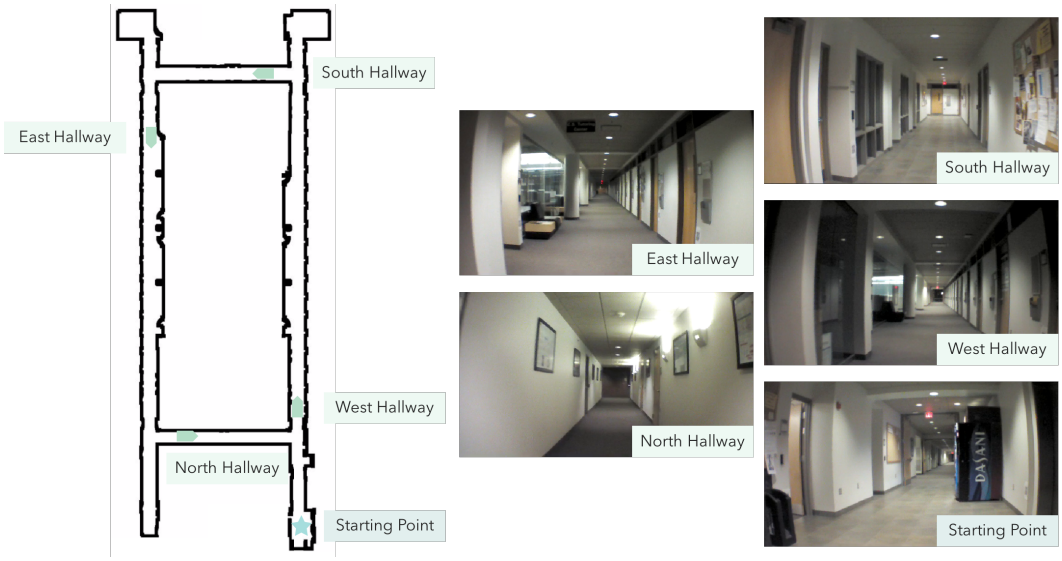


Figure 3.6: To the left, the floorplan of the area used for flight testing. To the right, sample images from actual bags. The corresponding location and direction the MAV was facing are marked on the floor plan.

We collected just over 25 sequences that ranged between 60 to 450 seconds in length, with most sequences falling in range of 100 to 200 seconds. The recorded sequences were predominately manual flights, with most of the longer sequences being "pushed" flights. The majority of autonomous flights were under 60 seconds and are not included in this figure. Sequences start from flight takeoff to land, and although tracking may have been briefly lost and relocalization necessary, the system is able to recover or maintain tracking for all sequences.

Most sequences begin at at the starting point highlighted in Figure 3.6 and attempt to head either clockwise or counter-clockwise around the hallways. Recording in both directions was done to provide the rotated (opposite) angles

of the same features. Additionally, most of these sequences were on the range of 10 meters or less in distance, traveling only a portion of one or two hallways, with a handful of sequences actually looping all the way back around to the starting point, due to the time it takes to travel the full loop—on the scale of five minutes or more.

All topic data during the flights with the AR.Drone 2.0s were collected—this includes the raw grayscale image (*sensor_msgs/Image*), the IMU data (*sensor_msgs/Imu*), AR.Drone navdata (*ardrone_autonomy/Navdata*), estimated AR.Drone odometry (*nav_msgs/Odometry*), and estimated ORB-SLAM pose (*geometry_msgs/PoseStamped*). In attempted autonomous and manual flights, control inputs were also recorded (*geometry_msgs/Twist*). The result is roughly 1GB of uncompressed data per minute of flight time.

The greatest differentiator of our sequences is also the greatest weakness. We found that the bags produced were simply not consistent or reliable enough to be used for any form of benchmark testing, and as such, we defer to the EuRoC dataset for most of our evaluation. We discuss the issue that inherently arises from using predefined sequences in 3.5.2, which is amplified when the quality of the input feed is lower quality—the system has significant difficulty recovering once localization is lost.

This dataset is not intended to be a contribution of this thesis due to the low quality and inconsistency of the results.

3.4 SLAM Library

At the core of our system is the underlying SLAM library. This work is not aiming at any particular system, but rather aims to harness the rapidly increasing set of high-quality open source SLAM options. The original goal was to make our system generic enough such that any of these options could be slotted in with limited work. Although conceptually this is nice and still holds potential for the future, in practice, the SLAM algorithms evaluated were fundamentally limited or too different in a way that this was not possible and not within the scope of this work. Furthermore, catering to the specific needs of a SLAM algorithm in order to be performant takes precedence over potential gains in algorithm flexibility—after all, it is better to have one algorithm that works well than many that do not work well at all!

3.4.1 ORB-SLAM: The System of the Hour

We chose ORB-SLAM for our purposes because it solves a few key problems we face as outlined previously in this thesis. Although the semi-dense map of LSD-SLAM could potentially provide a better understanding and map of the area which is useful when attempting to navigate through 3D space (demonstrated in Figure 2.3), it has issues with speed. Although capable of supporting a single UAV from a single computer, it would likely not be able to handle multiple on a single computer, and the map size could be problematic if attempting to distribute the computing to multiple computers. Furthermore, map size grows larger and quicker as compared to its sparse-

based competitors. On the opposite side, SVO runs extremely quickly, but in our testing and general thought is that it does not handle rotations well, making it more suitable for downward-facing cameras. Furthermore, it does not support the full suite of SLAM functionality—notably, it does not support loop detection. ORB-SLAM has the advantage of being fast enough that we can support multiple UAVs from a single computer, and the sparse map adds the potential of being able to distribute work over a network for scalability due to the limited bandwidth required.

Through the development of our system, we made a few modifications to the ORB-SLAM core code, as well as added a few custom classes for our multi-robot needs. These changes are further discussed in coordination server subsystem.

Let us dive a little deeper into the internals of ORB-SLAM, which will help us in describing our approach and alterations made.

We can see the design of ORB-SLAM in Figure 3.7. A quick walk-through on the data being used and stored: a keyframe k_i stores estimated camera pose and information, the ORB features extracted, and references to the set of map points P_i that are visible from the camera. Note that P_i can contain map points not observed initially in the processing in k_i, and map points are shared with the set of neighbor keyframes K_1 that can also see the map points. The set K_1, in combination the set of keyframes K_2 that are neighbors to K_1. K_1 and K_2 are joined to comprise the local map LM_i for k_i. The set of all keyframes is considered the map (M). Keyframe connectivity

is stored as a covisibility graph C, which is a graph with keyframes as nodes, and weighted edges representing the number of shared map points between the two keyframes. A subgraph of C, the essential graph (E) is also stored, which is a spanning tree of "important" keyframes to be quickly traversed. Keyed on visual word, the set of keyframes containing word w (K_w) is stored for each word in the database (D_w), such that it can inserted, removed, or looked up in constant time.

The processing lifecycle of an inputted frame starts with a real-time tracking step, which encompasses extracting ORB features for the frame, estimating camera pose, and determining map points p. At the end of the tracking step, the system applies a set of heuristics to determine whether the frame qualifies for becoming a keyframe k. Upon adding the keyframe to the map, the map points are associated with any other keyframes for which it is determined are able to view the map point.

In local mapping, one of the biggest differentiators of ORB-SLAM is its aggressive culling of the map. Once the new keyframe is inserted, local bundle adjustment is run based on this new observation. Bundle adjustment can result in shifted keyframes and map points, including some that are deemed no longer necessary due to being too similar (a function of shared map points seen across different keyframes). The algorithm prioritizes aggressively adding keyframes, preferring to add frames when unsure, relying on its culling to clean up any extraneous keyframes as it goes. This turns out to be extremely useful in maintaining tracking—many keyframes are created and available especially

when it matters (while the camera is actively exploring the area and trying not to lose tracking), while still remaining efficient at the global map level over the long run.

The last piece in the cycle is loop detection and closing. The keyframe is compared against all frames in the map for similarity using a visual bag of words—essentially, a barebones description of each frame in terms of notable features, with most information stripped away. Imagine describing a room as chair,desk,window,book—every frame that contains most or all of those "phrases" (in ORB-SLAM, these are ORB descriptors) becomes a candidate for closer investigation. When potential frames are identified as candidates for loop closing (enough shared vocabulary that the two frames *could* be in the same area), ORB-SLAM calculates a similarity transform (Sim3) between the two frames. If this value is within a threshold, the frames are set for loop closure. This process involves stitching the new keyframe into the map next to the keyframe it matched with, followed by a global bundle adjustment to re-align all the frames (due to the additional information gained by joining two areas of the map).

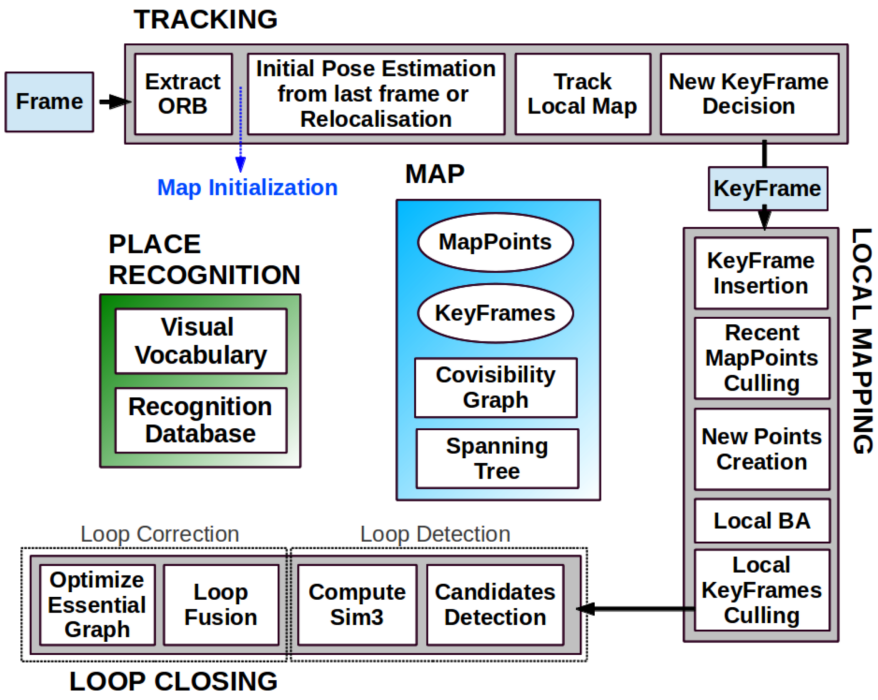


Figure 3.7: The ORB-SLAM system architecture [38].

3.5 Coordination Server

The coordination server is the backbone of the whole operation. It spawns ORB-SLAM threads for each of the UAVs, as well as conducts global loop closing across threads in a thread-safe manner. The main role of the coordination server is to take in any and all video streams provided by our UAVs (or more generically, any set of cameras) and output the current estimated camera pose, any key frame additions, updates, or deletions, and the current state of the tracker. In this section we will also discuss the modifications made to the original ORB_SLAM2 library in order to provide the

coordination server access and enable effective coordination. As previously mentioned, we intended to provide generic support to plug and play any given SLAM algorithm. Although this was eventually discarded for the aforementioned reasoning, attempts were still made to design our multi-UAV support in as minimalistic a way as possible in regards to alteration footprint.

Conceptually, the coordination server shares a lot of the same responsibilities/capabilities that the individual ORB-SLAM threads have, but it performs these responsibilities from a global viewpoint, across all threads with the increased pool of keyframes to work from. The basic design is demonstrated in Figure 3.8.

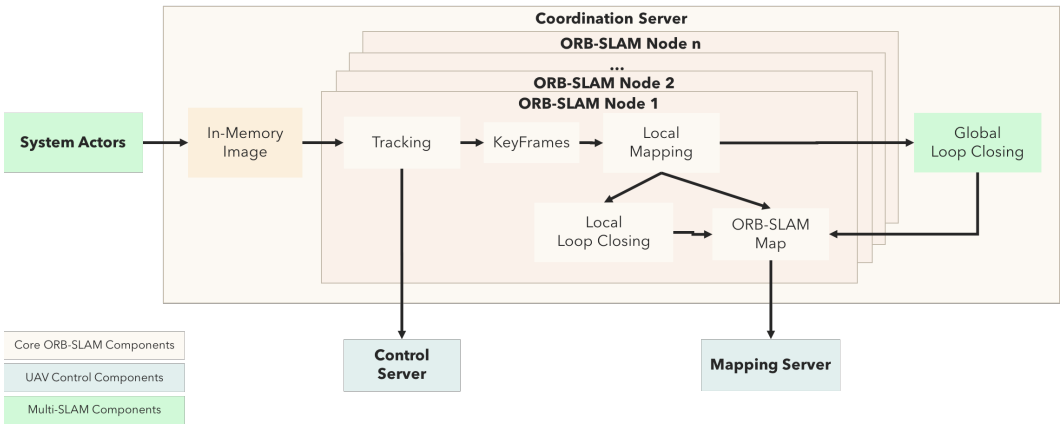


Figure 3.8: The coordination server design.

In this figure, you can see that the coordination server launches each of the ORB-SLAM clients (our terminology for an ORB-SLAM instance in charge of a single entity), as well as a global loop closer. It also acts as the conduit from the incoming ROS messages (far left) into the system. The ROS

subscriber captures incoming sensor_msgs, maps them to the corresponding client threads, and upon successful tracking, publishes the the new frame and map points generated for consumption further down the pipe (by our mapping server). More information on this can be found in the mapping server section.

Let us dive deeper and examine the global loop closer, described in more detail in Figure 3.9.

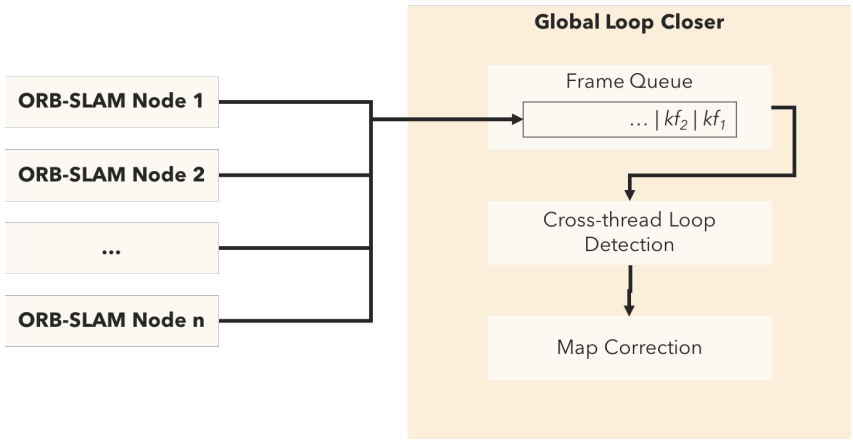


Figure 3.9: The global loop closer.

As you can see, ORB-SLAM clients queue up keyframes to be digested. These are pulled off and put into cross-thread loop detection that uses the algorithm defined in Algorithm 1.

If no loop is detected, the global detector sleeps briefly and then tries again. If a loop is detected, we begin by calculating the difference between k_i and k_n in the form of the Sim3 difference between their poses. This difference serves as the basis for adjusting map M_i to align with M_n.

Algorithm 1 Global Loop Detection for a Given Keyframe

given a keyframe k_i from ORB-SLAM node o_i,
for each other ORB-SLAM node o_n with map M_n **do**
  for each Keyframe k_n in M_n **do**
    if k_i meets similarity threshold to k_n **then**
      Mark k_i for loop closure against k_n
      Break
    end if
  end for
end for

The camera of k_i is adjusted directly according to the Sim3, and the frames in the local map LM_i are then all adjusted relative to that transform. Visible map points are also adjusted for each of these frames. This process is followed by a global bundle adjustment across all frames in M_i. Note that M_n remains unaltered. As previously mentioned, this is a design decision to reduce inter-dependency of our maps in the case where estimation was drastically wrong (we can better isolate failures).

At this point, we track the transform and store the full maps for offline pose error analysis.

With the keyframe and map points aligned, the covisibility graph and essential graphs can be adjusted and optimized for C_i and E_i.

The important generalization piece here is that we harness the algorithm's concept of a loop closure. This is normally meant to detect and correct when the input feed produces a frame that matches above a certain threshold to a keyframe previously mapped by the same feed—implying the UAV

has returned to the same location, thus creating the loop. The frames are merged if applicable, and preceding frames are corrected proportionally in order to reconcile the difference between the previously estimated location of the new frame and the location of the keyframe it matched with. In a one-input system, this difference is simply due to drift. In our multi-input system, we can instead attribute it as an observation of the translation (both pose and rotation) between two maps, allowing us to merge.

After a loop closure is detected between two sibling maps, we initiate the map correction and bundle adjustment for the originating map (the map from which the keyframe came). This is done using the calculated transformation matrix mapping the original keyframe to the keyframe with which it matched. This matrix is propagated through, adjusting keyframe poses and map points along the way, resulting in a (hopefully!) well-synced pair of maps. Loop closure and bundle adjustment are common in all the evaluated monocular SLAM algorithms, and as such, one can extrapolate how this same method could theoretically be transferred to any of these other SLAM algorithms, not just ORB-SLAM.

It is important to note that at no point are the maps belonging to each ORB-SLAM client actually merged—rather, we attempt to maintain an in-sync relationship between all maps by transforming the maps whenever they detect overlap. The reasoning behind this is predominantly due to the uncertainty around monocular SLAM in particular. Monocular SLAM lacks any real notion of ground truth or pseudo-ground truth, which means the first

estimates—or even established ones—can turn out to be drastically wrong. At this point, the map divergence might be significant and re-separating the maps would be costly.

3.5.1 ORB_SLAM2 Library Adjustments

We were not kidding when we said we tried to avoid making changes to the original library as much as possible! We list the few high level changes below:

1. A global loop closer
2. A custom viewer with support for multiple inputs
3. A ROS container class for tracking custom attributes and reporting to the global loop closer

There are several other small adjustments made in order to glue everything together, including exposing variables, assigning custom IDs, and more, but are mostly inconsequential. Furthermore, we did adopt a few community features of which we are very grateful, such as a binary BoW vocabulary [https://github.com/poine/ORB_SLAM2] to improve loading speed, as well as ROS catkinization [https://github.com/varhub/ORB_SLAM2/tree/ros-catkinization].

3.5.2 Cross-Thread Relocalization

When an ORB-SLAM node loses localization, it will initiate the relocalization protocol, scanning its keyframe database for any frames above a

certain similarity that may shed light on the UAVs location. However, depending on the datasets used and where localization is lost, it may result in ORB-SLAM losing localization for the remainder of the dataset sequence run. Cross-thread relocalization is the notion of expanding the ORB-SLAM relocalization process to incorporate frames from other threads, allowing the UAV to recover based on frames observed by another UAV in the event that it was significantly displaced from any frames it itself had generated.

We propose a simple queue system similar to the global loop closure approach. In the event that a UAV loses localization, a thread would first attempt localization on its own map. If this fails, the UAV would submit the frame to the coordination server, which would attempt to localize the frame on each other map. In the event that the coordination server returns and match and the UAV is able to relocalize, new frames could be inserted and mapping could re-commence.

However, our implementation restricts the practicality of this approach—although frames would be metrically sound (producing accurate keyframes and map points in 3D space is possible once relocalized), the topological aspect of an ORB-SLAM map (the pose graph and the associated dependencies between frames) would be broken.

We were able to prove out the initial relocalization process as described and encountered the topological map challenges. There are a great deal of interesting complexities here that we would like to further explore in further work.

As an alternative to cross-thread relocalization, loss of localization can be somewhat mitigated by backtracking and making recovery movements as described later in Section 3.7—however, dataset sequence runs do not respect the system's notion of whether it is localized or not and will continue chugging along and producing frames. One option would be to adapt the dataset sequence player (in our case, *rosbag run*) to rewind and play the past few seconds until successfully relocalized. This was not implemented and is another good candidate for future improvement.

3.6 Mapping Server

In order to convert the data published from our coordination server into something actionable, we built a custom Octomap [24] server that reads ORB-SLAM outputs and turns them into actionable maps. Octomap is a 3D occupancy grid library with ROS compatibility, leveraging the power of octrees to efficiently store occupancy probabilities.

It accepts keyframes as input, a lightweight custom ROS message containing a ROS Header, a unique frame id, an origin point of type geometry_msgs/Point32, and a list of geometry_msgs/Point32s containing the map points captured in the key frame. These are projected onto the Octomap, which is then published in its raw 3D form, as well as a downprojected 2D occupancy grid, which uses the points in the middle 50 percent of the vertical axis for determining occupancy. Grid size is dynamically determined, as the scale of the input will vary with each run due to the nature of monocular

SLAM. An example occupancy grid can be seen in Figure 3.10. The middle 50 percent of the vertical was used in order to provide a safe buffer zone for the MAVs to fly around in—it takes relatively few misinterpreted frames to mess up the proximity of something like an object on the floor.

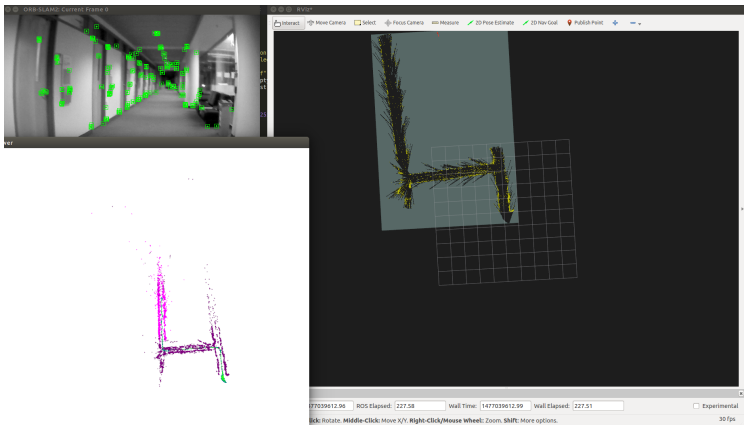


Figure 3.10: The camera view (upper left), the ORB-SLAM generated map (lower left), and the corresponding generated 2D occupancy grid for use with a control system.

3.7 Control Server

The final piece of our system is a control server, which accepts the output of the mapping server and the estimated poses of each of the UAVs and coordinates both individual and group movement. The basic control flow is demonstrated in Figure 3.11. This is implemented in Python/ROS, and interacts the same way other systems described interact—via ROS messages.

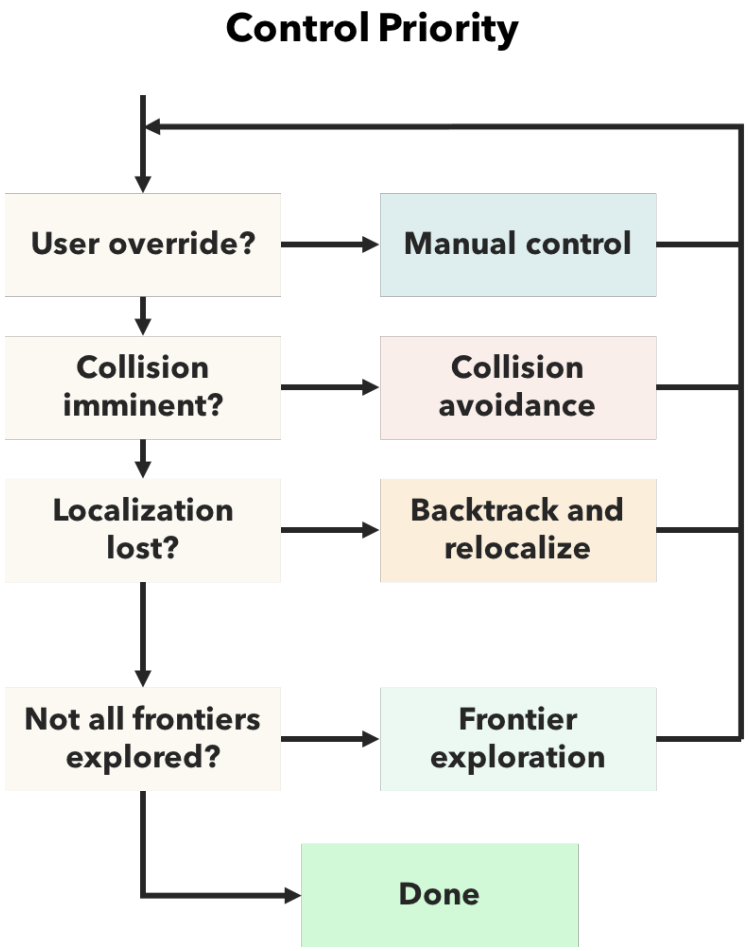


Figure 3.11: The control priority flow, influenced by [36]

3.7.1 Core Controls

At a more granular level, stabilization and move-to-location is handled by the AR.Drone 2.0 itself. The built-in control loop does leave a little to be desired. Interestingly, other work has used ORB-SLAM to successfully maintain flight by controlling the UAV directly in a tight control loop by

processing the camera and IMU [31]. We opt to use the built-in functionality, as operating a tight control loop requires significant processing in its own respect and would hurt our ability to scale.

Since we have elected to navigate based off of a 2D occupancy grid and are electing to use the built-in stabilization of the AR.Drone 2.0, our control system needs are greatly reduced—we primarily care about the four degrees of freedom ((x, y, z, ψ)) similar to that of a typical ground-restricted vehicle that revolve around movement from point to point in 2D space.

3.7.2 Safety First!

Manual override control can be achieved by running a separate ROS subsystem that could send arbitrary commands to the AR.Drone 2.0 over top of those already sent by any automated system, including takeoff, landing, and steering. We set up a direct pipe to the MAV via an *ardrone_joystick* [45] ROS node listening to raw controller controls coming from a *joy_node* [2] ROS node. The most common use case for this is to force a land, which handles the job of locking out any other movement commands automatically.

Some basic rules are added in to avoid collisions and maintain safe flight. Acceleration is eased to reduce the start/stop effects (more drastic motions result in more drastic errors). Caps are set on both linear and angular velocity to avoid overcompensation and causing damage to the MAVs.

3.7.3 Handling Relocalization

Current state of the ORB-SLAM node is consumed by the control server. In the event that localization is lost, a recovery routine is manually set, which involves make small horizontal and vertical movements without adjusting orientation. A simple backtracking algorithm is proposed—publish the last n reverse *Twist* messages (the linear and angular velocity command), with the idea that the negated *Twist*s are able to get the MAV back to a place where it can relocalize. This is a nave, but straightforward approach and does not account for the change in momentum.

3.7.4 Exploration

We propose exploration is done via a basic frontier algorithm on top of the generated 2D grid. A frontier exploration algorithm is built on finding and mapping the nearest "frontiers"—the boundary between known and unknown map space—to the robot. In the event that the robot hits a dead end, the nearest frontier becomes a different path that the robot did not take, and thus backtracking to the next viable location is handled seamlessly. Frontier exploration is useful for our needs—given sufficient time, a robot exploring frontiers will explore all possible mappable area. There is a ROS frontier_exploration library [1] which provides an out-of-the-box solution. Due to time constraints and poor performance, we did not take the time to integrate a frontier exploration algorithm. Instead, "automated" exploration was controlled by sending "goal" poses in some unknown region via the rviz UI (a visualization tool built

on top of ROS). By setting the goal pose in unknown space, we get a cheap approximation of map exploration.

3.7.5 Extras

For our "pushed" flights outlined in Section 3.3.3.3, we wanted to create flights that would be trackable by ORB-SLAM when replayed. To do so, we consumed the current system status from ROS and built a node to update the lights onboard the AR.Drone 2.0 to a particular configuration corresponding to status. When pushing the UAV around, if the status changed, we could easily reverse steps and manually perform recovery motions until the system was relocalized.

Chapter 4

Results and Discussion

In this chapter, we describe our approach to testing the presented system, which can be divided into two parts. The first part is benchmark analysis conducted using a popular monocular dataset to demonstrate the value and effectiveness of our system in more granular slices. The second part is real-world trials in which we demonstrate use of the modified library as applied to low-cost commercial MAVs. We also discuss the outcomes and lessons learned that emerged from the experimentation

4.1 Benchmark Analysis

4.1.1 Data Collection and Sources

All of our results are generated from the machine hall sequence grouping described in Section 3.3.3.1.

For benchmark results, we use results previously collected for another work [46]. The work has results for each individual sequence, averaging the accuracy from 10 runs of the unmodified library on the sequence — providing the baseline for individual performance. In this section, we refer to this as the "unmodified library" and is labeled as *single* in graphs or charts.

For our results, we simulate handling multiple MAVs providing inputs concurrently by running all the sequences in a given EuRoC grouping simultaneously using our modified ORB-SLAM. The sequences vary in length, so although all sequences are launched at the same time, they ultimately finish at different times. However, other sequences may realign and merge onto the map belonging to a completed sequence.

Trials were done additively—*nodes=2* or *multi-2* means we run sequences MH_01 and MH_02 concurrently, with one node dedicated to consuming the "inputs" for each respective sequence, *nodes=3* or *multi-3* means we use sequences MH_01-MH_03 with three nodes respectively, and so on, up to five simultaneous nodes.

While running, we store metadata and data about the state of the maps and connections between different nodes' maps. The exact data stored is described in the relevant subsequent sections. At shutdown time, the state is saved one final time to be used for final system evaluation.

At evaluation time (offline, after runs are complete), we calculate our metrics and are able to slice out results based on the number of simultaneous nodes and/or based on the specific sequence results (within the context of number of nodes that were running). This is needed to compare the performance of any one sequence when run in "single" mode (the unmodified library) as compared to "multi" mode (our modified version with other sequences running simultaneously).

One flaw of our system implementation is that non-deterministically, the mapping process on the ground control station will exit, causing the control station to fail to map. In practice, we note that an average of 20% of runs fail due to this issue. Regrettably we were unable to trace the root cause of this issue and look to resolve this in future work.

4.1.2 Keyframe Pose Error

In this experiment, we demonstrate that the modified ORB-SLAM library is capable of similar accuracy on a per-sequence basis as to that of the original ORB-SLAM library. By introducing multiple input feeds into the mapping process, we want to ensure that quality is not lost. False map merges can cause keyframe corrections that lead to less-than-optimal results. To test this, we measure the error produced in a map generated by the original single entity ORB-SLAM and compare to the error generated by our multiple entity ORB-SLAM. We use the grouping of sequences from the previously described EuRoC MAV dataset.

4.1.2.1 Determining Root Mean Square Error

Accuracy is measured as the root mean square error (RMSE) between the keyframes in the final pose graph and their corresponding ground truth poses. However, the output of SLAM systems will not have the correct trajectory alignment without some form of external sensor, which we do not accommodate for in our design, so we must do an alignment pass for evalua-

tion purposes. Furthermore, monocular SLAM systems such as ORB-SLAM need scale adjustment in order to match up with any ground truth data. To handle the post-processing alignment, we utilize a tool written by Raul Mur, evaluate-ate-scale [37]. This tool provides the necessary alignment, as well as supports plotting of the ground truth and estimated trajectories and overall RMSE values, which we use in our analysis.

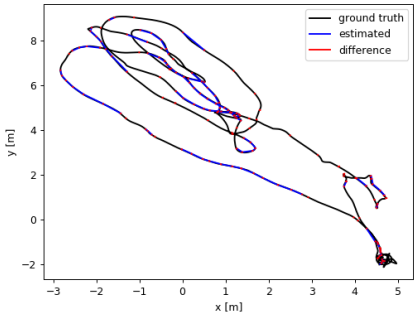
4.1.2.2 Trajectory Spot Checks

In Figure 4.1 we show example trajectories for each of the sequences tested. This is done to help provide a sense of the routes taken by each of the sequences.

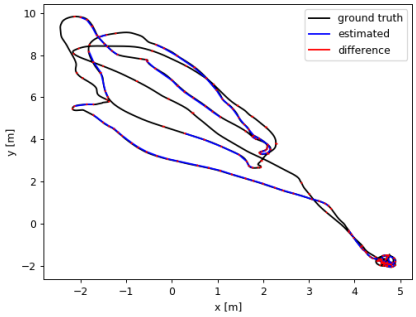
4.1.2.3 Comparison to Unmodified Library

Individually, the above figures are good for identifying problem areas and providing context on the sequences. However, in order to determine overall system health, we compare average RMSE directly between the benchmark results and our results, displayed in Figure 4.2. Here, we see the average RMSE achieved by the unmodified single entity ORB-SLAM compared side-by-side to the resulting RMSEs from a varying number of nodes running simultaneously in our modified multiple entity ORB-SLAM.

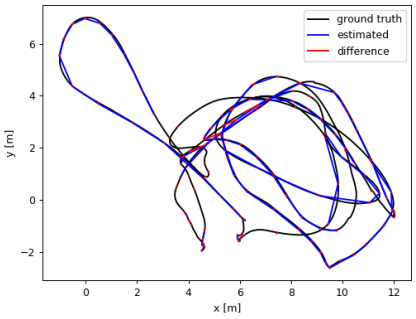
Due to our approach of running trials additively, we only see the MH_05_difficult sequence in the multi-5 results. Future analysis would be interesting if sequences were tested round-robin to get a better sense of variability as a func-



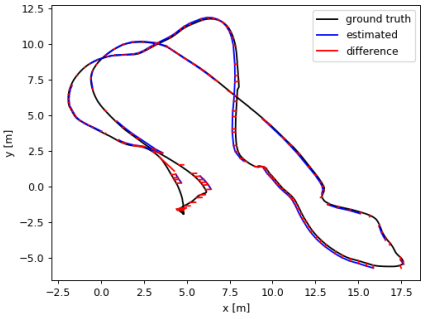
(a) MH_01_easy.



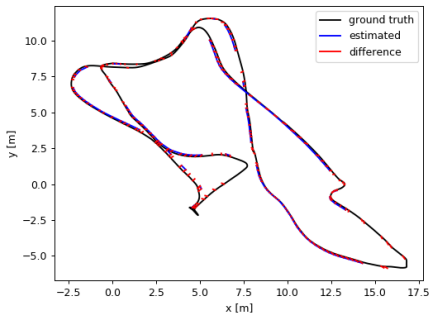
(b) MH_02_easy.



(c) MH_03_medium.



(d) MH_04_difficult.



(e) MH_05_difficult.

Figure 4.1: Sample trajectories for each of the five machine hall sequences in the EuRoC MAV dataset, used for testing.

tion of difficulty (since the higher numbered sequences are more difficult for the SLAM system to follow).

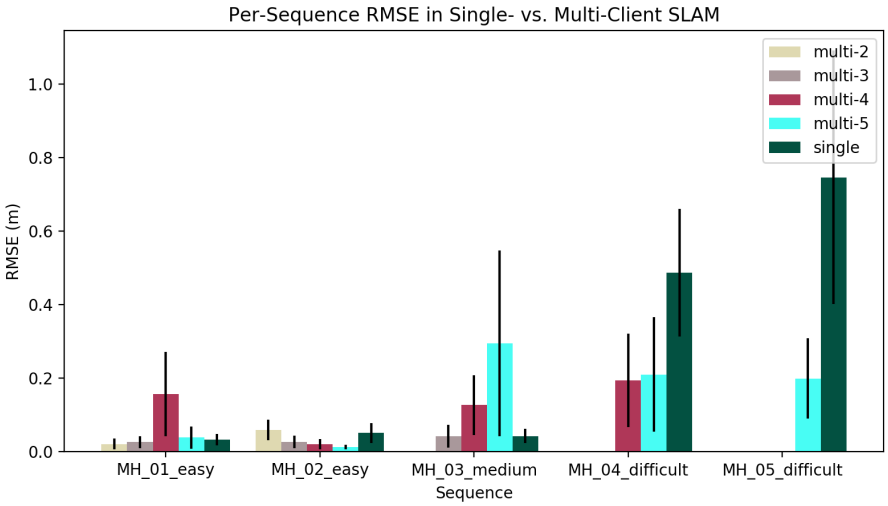


Figure 4.2: Comparison of average RMSE between the unmodified library and our modified library with a varying number of nodes. Since we ran our trials additively, MH_03_medium only appears in the multi-3/4/5 results, MH_04_difficult only appears in the multi-4/5 results, and MH_05_difficult only appears in the multi-5 results.

There is a bit to unpack from this graph. We will explore a few interesting observations, sequence by sequence.

High RMSE in multi-4 for MH_01_easy We see a slight bump in RMSE for sequence MH_01_easy in particular for when we ran with four concurrent sequences. This could allude to some sensitivity to certain sequences interacting—it is certainly possible that this sequence is corrected aggressively

against a wrong sequence. This theory is put to the test using data collected for a future section, Section 4.1.3, where we will further pursue this hypothesis.

Positive Results for Difficult Sequences; Neutral-to-Negative Results for Medium Sequences Of particular note, we see in Figure 4.2 that for the the two difficult sequences, MH_04_difficult and MH_05_difficult, we actually see significant improvements in average RMSE in our multi-slam as compared to unmodified SLAM. This is promising—the interoperation of sequence graphs appears to improve the accuracy for difficult sources. To the flip side, RMSE for the MH_03_medium appears to spike up, especially when more sequences are added in, relative to the unmodified library. This may very well represent some of the reservations we expressed with our initial hypothesis—adding additional noise to the system (in the form of other maps constantly merging with each other)—has the potential to throw off what would otherwise be a very accurate run.

4.1.3 Map Merging Volatility

We evaluate the volatility of our map merging as a proxy for our global mapping health. In an ideal world, the average keyframe adjustments should reduce over time across positional transformation, rotation, and scale. In practice, this is not always the case and is susceptible to the overlap points and quality of overlap between MAVs over time. It is also important to note that all sequences end in approximately the same physical location as they

start, allowing for loop closure testing. This results in large positional spikes towards the end of a sequence, when the camera feed shows very similar frames as to when started. This is a common trait in most data sets intended for testing loop closures.

4.1.3.1 Gathering Merge Information

To measure the merging delta over time, we log details around each map merge as they occur and the affected keyframe transformations. The set of data collected with each merge is as follows:

- *timestamp* - the timestamp when the merge was initiated
- *gba_run* - a constantly-increasing int used to uniquely identify the merge and subsequent bundle adjustment
- *source_idx* - index of the map that will be merging into the other map (frames in the map will change)
- *target_idx* - index of the map that will be merged into (frames in this map will not change)
- *parent_kf_id* - for diagnostic information, the id of the parent keyframe to track common patterns
- *adjusted_kf_id* - for diagnostic information, the id of the adjusted keyframe to track common patterns

- *r0* - quaternion angle 0 (also known as qx)
- *r1* - quaternion angle 1 (also known as qy)
- *r2* - quaternion angle 2 (also known as qz)
- *r3* - quaternion angle 3 (also known as qw)
- *tx* - euclidean transformation in the x direction
- *ty* - euclidean transformation in the y direction
- *tz* - euclidean transformation in the z direction
- *scale* - arbitrary scale variable needed for monocular SLAM

This timeseries data is collected for all merges (representing the permutation of map 0 into map 1, map 1 into map 0, map 0 into 2, etc.), and then results are aggregated across many runs to provide the resistance against randomness. Due to the data being a timeseries, aggregated across runs, as well as just high-dimensional data, it quickly becomes both difficult to reason about as well as difficult to visualize.

It is important to note that the values are deltas—changes to the given dimension at that point in time—and not actual position. High swings in deltas are expected at least once, as each map starts centered at its own set of home coordinates. When first attaching to another map, one of the maps is selected as the source of truth and the other attempts to match. Subsequent

swings may represent correcting an eager or aggressive merge. As previously mentioned, this may occur very near to the end of a sequence, when similar keyframes are brought back into view, providing an anchor for self-loop closure.

4.1.3.2 Delta Spot Check

We first get a sense of the volatility by inspecting one run, grouping deltas by sets of degrees of freedom: euclidean transform (Figure 4.3), rotation (Figure 4.4), and scale (Figure 4.5).

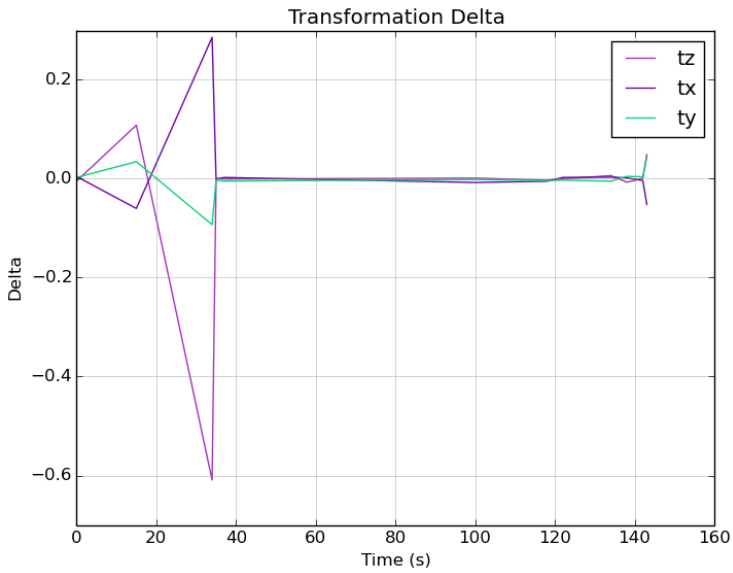


Figure 4.3: A sample transformation (XYZ) delta over time. Changes are tracked at each merge.

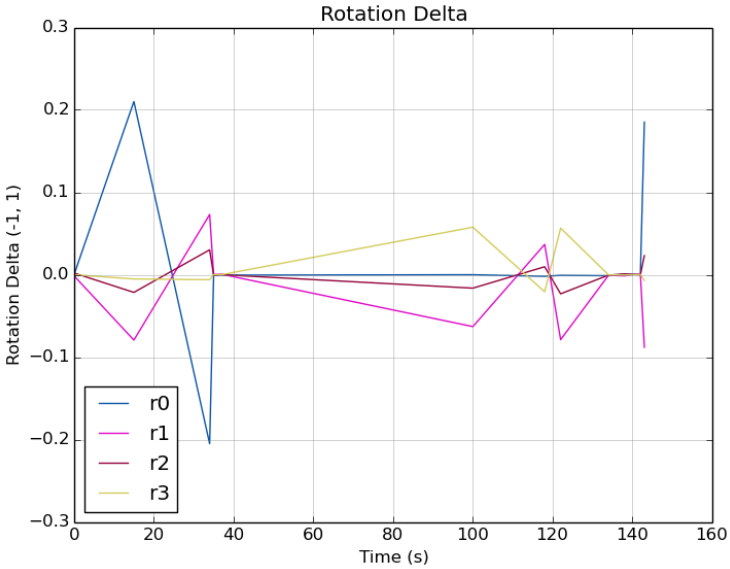


Figure 4.4: A sample rotation delta over time. Changes are tracked at each merge.

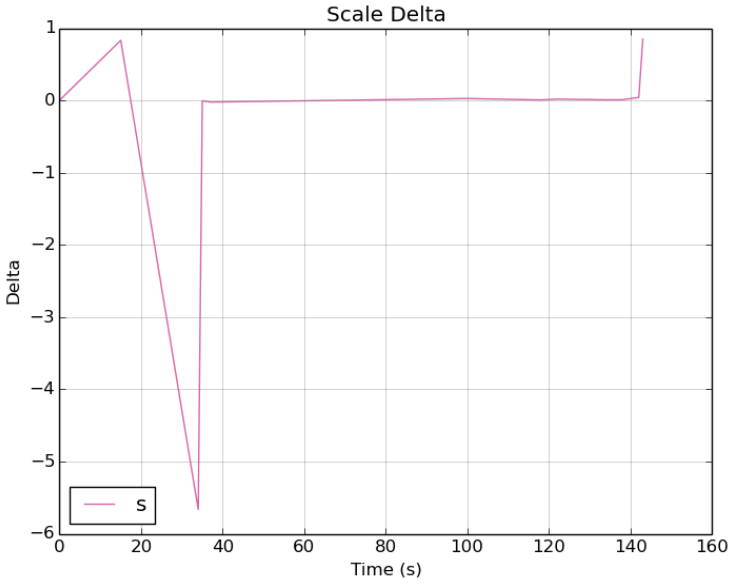


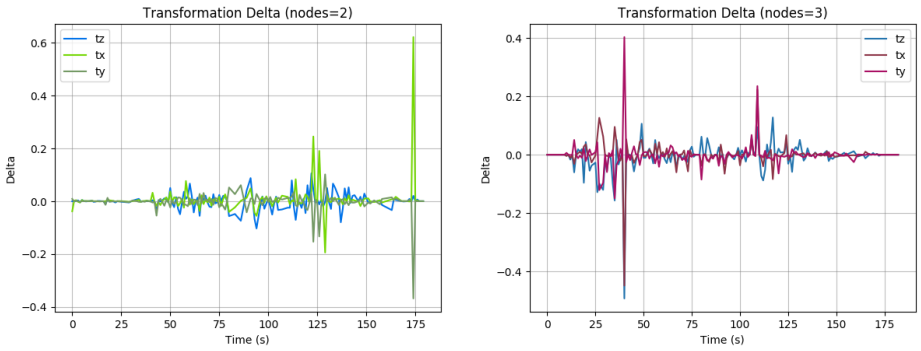
Figure 4.5: A sample scale delta over time. Changes are tracked at each merge.

Of course, this is just one run, and the variance is great from run to run.

4.1.3.3 Viewing Averaged Volatility Across Many Runs

We present charts demonstrating the merging delta over time, using all runs and averaging out values when necessary. After spot-checking in Section 4.1.3.2, we want to view aggregate behaviors to validate that volatility remains stable, indicating a generally healthy global mapping ecosystem. We start by taking average delta over time to get an idea for the shape of deltas for each degree of freedom, and viewing this across each of the run configurations (between two and five nodes simultaneously). This can be seen in Figure 4.6, Figure 4.7, and Figure 4.8.

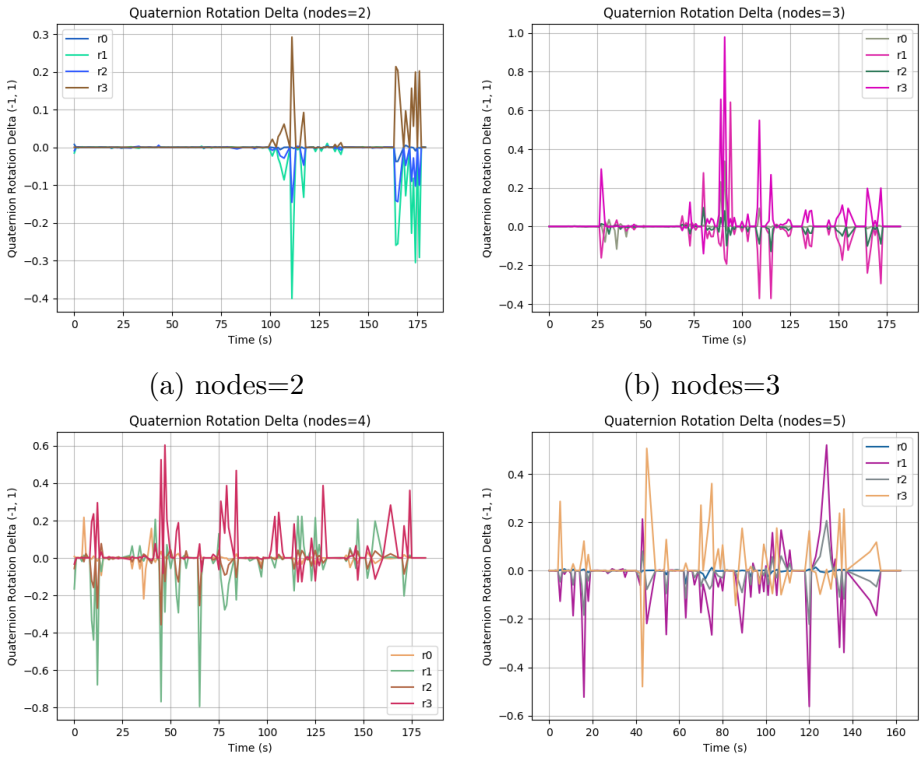
At the aggregate level, we see a few details worth a minor note, but nothing significant in either direction. This is generally not bad! If inter-map volatility does not degrade as a function of the number of nodes (and thus the amount of entropy in the system), that is a good sign. One example of the increasing entropy is seen with the rotation deltas—you can see that merges become diffused throughout the time period, rather than concentrated in the small spikes where the two nodes are able to see each other. One other callout with these graphs is around Figure 4.8—note that scale is an arbitrary value and subject to large swings and these charts are not inherently designed to be resistant to outliers. With this in mind, scale is the only one of the three DOF groups that appears to show signs of early drastic adjustment, resulting



(a) nodes=2 (b) nodes=3

(c) nodes=4 (d) nodes=5

Figure 4.6: Transformation (XYZ) deltas over time, across all runs, for various numbers of nodes. Changes are tracked at each merge.



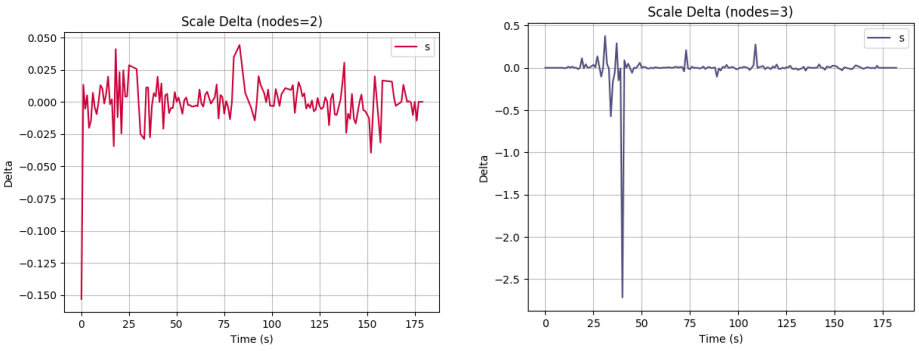
(a) nodes=2

(b) nodes=3

(c) nodes=4

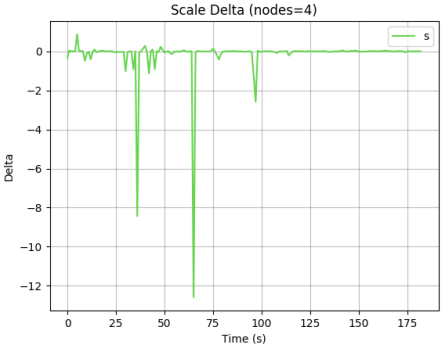
(d) nodes=5

Figure 4.7: Rotation deltas over time, across all runs, for various numbers of nodes. Changes are tracked at each merge.

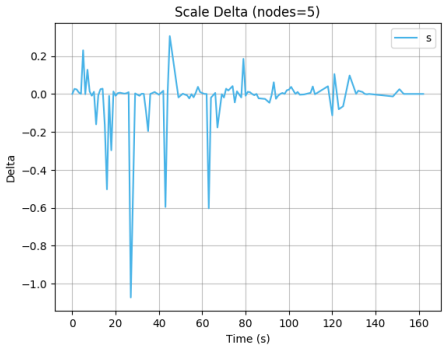


(a) nodes=2

(b) nodes=3



(c) nodes=4



(d) nodes=5

Figure 4.8: Scale deltas over time, across all runs, for various numbers of nodes. Changes are tracked at each merge.

in very little later adjustment (relatively speaking). This makes sense due to the arbitrary nature—once two maps are in sync around a notion of scale, they will not need to do much alteration to this after the fact (or at least, very little relative to any initial adjustments that have to be made).

This is just one way to slice the data. Another interesting approach would be to display each run as a separate series, with one graph per degree of freedom, rather than grouping them, which gives an idea as to the volatility from run to run on a per-degree-of-freedom basis. Yet another slice would be to view the deltas for a given degree of freedom with the number of nodes represented each as a series, which highlights the effect the number of nodes has on merging.

4.1.3.4 Using Map Volatility to Test RMSE Theories

We also have the ability to slice by the volume of merges from any one node's map onto another node's map during a run.

This data is useful for further investigation interesting behavior, such as that described in Section 4.1.2.3. To investigate, we first look at the number of merges made by a given source sequence to match a given target sequence—the sequence has identified that another sequence is similar to itself and attempts to adjust itself to match. We might see some significant spikes in volume of interaction where the MH_01 sequence as a *source* is merged into a larger number of *targets*. We can check Table 4.3, which has the average number of merges for MH_01 in the context of four simultaneous nodes. We do not see any

		Target				
		MH1	MH2	MH3	MH4	MH5
Source	MH1	0	1780	0	0	0
	MH2	1644	0	0	0	0
	MH3	0	0	0	0	0
	MH4	0	0	0	0	0
	MH5	0	0	0	0	0

Table 4.1: The average number of global bundle adjustments applied from a source sequence onto a target sequence for multi-2 runs.

particularly obvious spikes, and so ultimately this is possibly not the correct tool for explaining the aforementioned behavior, although further data slicing along these lines could also yield information—for example, it is also possible that the merges occurring are significant in nature (think big, sweeping changes to the map). This is an open question and a subject for future investigation—why *does* the four node run result in the MH_01 error to spike so much higher?

In addition to generating the merge volume table for the four node runs while attempting to address the above open question, we also generated merge volume for the other runs (Tables 4.1, 4.2, 4.4). No clear patterns emerge, although one interesting observation is that the average number of merges from MH_01 to MH_02 and vice verse is very high when those are the only two sequences, but drop significantly when additional sequences are introduced. Since merging is greedy—a new keyframe in a given map will try to latch onto the first matching keyframe from any other map, the presence of additional sequences will likely lower the number of merges between any two given maps, but the falloff is still quite drastic.

		Target				
		MH1	MH2	MH3	MH4	MH5
Source	MH1	0	461	117	2	0
	MH2	223	0	82	0	0
	MH3	77	91	0	0	0
	MH4	14	2	0	0	0
	MH5	0	0	0	0	0

Table 4.2: The average number of global bundle adjustments applied from a source sequence onto a target sequence for multi-3 runs.

		Target				
		MH1	MH2	MH3	MH4	MH5
Source	MH1	0	635	394	18	0
	MH2	478	0	210	59	0
	MH3	58	44	0	27	0
	MH4	0	0	53	0	0
	MH5	0	0	0	0	0

Table 4.3: The average number of global bundle adjustments applied from a source sequence onto a target sequence for multi-4 runs.

		Target				
		MH1	MH2	MH3	MH4	MH5
Source	MH1	0	232	121	0	0
	MH2	146	0	94	0	13
	MH3	26	28	0	27	36
	MH4	0	0	53	0	63
	MH5	0	0	4	148	0

Table 4.4: The average number of global bundle adjustments applied from a source sequence onto a target sequence for multi-5 runs.

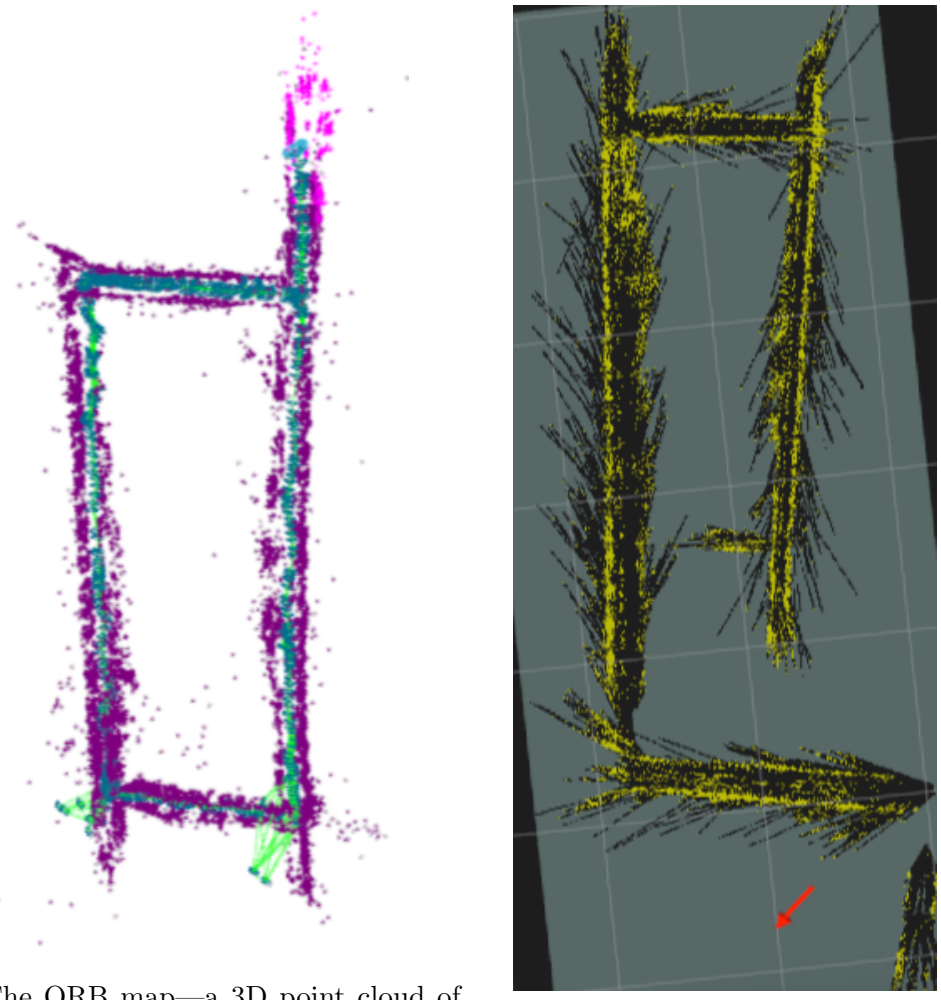
4.2 Hardware Trials

Finally, we demonstrate a basic rigging with real hardware to demonstrate the possibilities and restrictions in the application of our modified library. We previously describe a significant portion of our testing setup in Section about-hardware-platform and discuss the location and testing approach in Section rit-dataset.

4.2.1 Mapping

At the core of our real world trials is our ability to map. This does not mean just our ability to track keyframes and map points, but rather our ability to convert the ORB-SLAM map into a navigable map to make control decisions on as our MAVs fly around. As specified in our system design, we opted to attempt navigation using a 2D occupancy grid. To iterate on the grid quality, we took one of our full loop sequences from the RIT dataset and ran it in a singular MAV configuration. Mapping (point cloud, 3D Octomap, and 2D occupancy grid) are calculated in real-time.

First attempts to build the occupancy grid were unsuccessful since the mapping server did not account for constantly changing and adjusting keyframes. Since we are using monocular SLAM, keyframes and their respective map points are liable to move around frequently or disappear entirely when loop closing and bundle adjusting. In Figure 4.9 we can see an example of this, where the ORB node managed to correctly reconcile, but the occupancy grid is left in the dust.



(a) The ORB map—a 3D point cloud of map points.

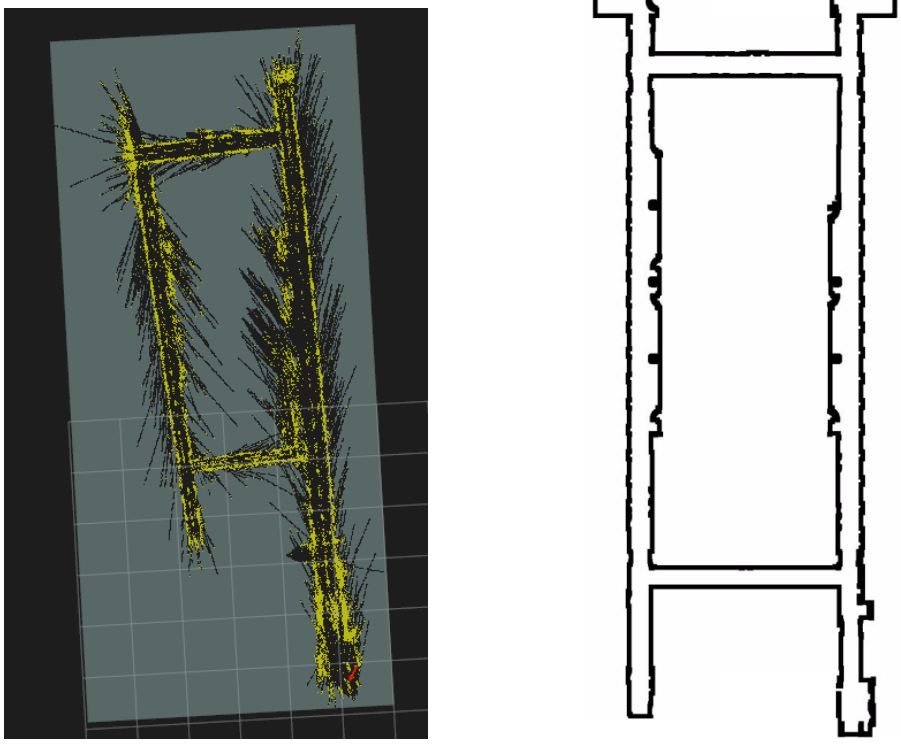
(b) The generated 2D occupancy grid.

Figure 4.9: An example of a less-than-optimal run with the mapping server. Note the bundle adjustment incorrectly propagated to the occupancy grid.

To account for this, we added support for the occupancy grid to backtrack occupancy and re-apply frames during bundle adjustments. Bundle-adjusted keyframes are re-published to the keyframe stream. A unique identi-

fier is included with the keyframe, so the mapping server is able to understand if we are adjusting an existing frame and correctly switches to occupancy correction mode. A small callout is that the mapping server is required to to store a local cached version of all the keyframes in order to correctly update old frames, which is trivial for the size and duration of maps we generated through testing, but could potentially be problematic for sufficiently large maps.

Accounting for bundle adjustments, we were able to produce promising occupancy grid results against the same sequence. In Figure 4.10 you can see the generated occupancy grid and a reference 2D ground truth map. Unfortunately with these real world trials, we have no ground truth for the path taken by the MAV itself.



(a) The generated 2D occupancy grid

(b) The ground truth floor plan of the building.

Figure 4.10: In these two figures, we can see a much more successful run of the occupancy grid, although we do see that the loop closure in the bottom right junction is incorrectly a few meters up the hallway from where it should have occurred.

4.2.2 Control System Performance

With a sufficient occupancy grid as defined in the previous section, we were able to attempt controlled flight. Avid readers got a sneak peek at the results when reading through the dataset description—ultimately independent exploration flight was not achieved in a real-world scenario.

4.2.2.1 Simulation Results

It is worth calling out that attempts were made to test our control system via the simulation platform. Unfortunately, as previously mentioned, we struggled with getting good readings via the platform. The simulation platform did provide a good opportunity to test that we could indeed launch multiple MAVs and guide them to explore simultaneously, but the aggressive loop closing prevented any real or actionable results. Further attempts to find more realistic and workable models for simulated monocular SLAM were unsuccessful and we did not pursue this option any further.

4.2.2.2 Basic Flight

Basic flight was achieved, for some definition of the word. Our control script was able to guide two MAVs through simultaneous takeoff and limited exploration, but this devolved very quickly—many crashes later, we were unable to make it further than a few meters collectively. We have multiple theories as to what is occurring here, and the exact cause is a little unknown.

Low-Quality MAVs Early on, this was identified as a potential risk of this thesis. By using low-cost MAVs, we are also using relatively low quality MAVs. First, it is worth noting that these MAVs can only be crashed a certain amount of time before they start to develop personalities, even after swapping out new shells, rotors, etc. Second, since we elected to use the onboard control loop for basic stabilization, we are also subject to its flaws. For manual flight,

the AR.Drones can be corrected by the user, but if the MAV starts drifting, an automated control system that is only worrying about the four degrees of freedom previously mentioned, will break down fairly quickly if the drift occurs in the two unaccounted-for degrees of freedom. For example, if the MAV tilts to one side ever-so-slightly, this can very quickly cause a crash if obstacles are nearby, which brings us to:

Tight Quarters The tests were predominantly done in a relatively confined space—hallways measuring just a few meters wide and tall. If the MAV loses control for a brief moment, it has very little time (imagine something less than 2 seconds, depending on the tilt) to not only recognize that it is slipping, but sufficiently counter the movement. Other times, a slight miscalculation on the ORB-SLAM side resulted in attempting to fly too high or too low, causing crashes. Frequently, the MAV would end up in a "wall-hugging" state—the MAV would shift to one side and slowly edge up against the wall, getting the protective hull stuck up against the wall. This acts somewhat as an anchor on one side of the MAV, causing the other side to flip up, either resulting in a crash or getting stuck against the wall. The MAV is unsure or unable to exit this state and requires a physical intervention to nudge it back on track. An example feed of this is seen in Figure 4.11.

Insufficient Flight Control Lastly, we would be remiss if we did not mention that we did not implement any crazy or wild flight control system. Basics

were added and tweaking was done to attempt to counter the crashing problem, but to very limited success. For example, bumping up the reactivity in order respond to drift in sufficient time resulted in over-compensating and crashing into the opposite wall instead. Various other small tweaks, such as easing, were attempted to no additional success. This is an exploration in its own right, as seen in previously cited works.

Figure 4.11: This wall-hugging scenario was not uncommon when testing in confined spaces. With the right side of the protective hull up against the wall, the drone starts to tilt ever-so-slightly upward, putting it in a weird control state. In less fortunate circumstances, this results in a crash, but this particular instance recovered by manually nudging the MAV off of the wall.

Chapter 5

Conclusion

We developed a multiple MAV system capable of supporting autonomous exploration and navigation in unknown environments using only a front-facing camera found in low-cost, commercially available MAVs. We modified a popular open-source SLAM library, ORB-SLAM, to support multiple input feeds. We demonstrated efficient group exploration with comparable mapping accuracy to the unmodified library on benchmark datasets with up to five MAVs. We compare accuracy by calculating root-mean-square error on the estimated vs. actual camera trajectory, and we also show system stability by tracking the volatility of the pose estimates over time during runs. We developed a mapping server for generating actionable maps and a control server for pushing commands to MAVs based on the mapping and real-time inputs from the modified ORB-SLAM—the two other pieces required for real-world trials. However, the real-world testing proved to be a formidable challenge on top of the other work demonstrated. We explore different potential causes for our unsuccessful hardware trials, but ultimately look to address this in future work.

5.1 Future Work

We would have liked to spend more time optimizing our multi-SLAM setup. A large portion of our effort was focused on getting it working, but several trade-offs were made in the process. The sheer volume of tasks pushed to the coordination server significantly bottlenecks the system—although testing with up to five simultaneous inputs is interesting, scaling and coordination problems are not particularly prevalent. Work to offload pieces of the coordination server, as well as the subsequent mapping and processing optimizations that would have to come with that would be very interesting and challenging.

DSO was released during the development of this work, so we were unable to trial it for adaptation to multi-SLAM. However, it showed significant promise both with accuracy and speed and would be an interesting candidate for investigating how alternative monocular SLAM systems can have a higher or lower propensity for multi-SLAM.

From an execution point of view, further work should focus on expanding the datasets used for analysis. It would be insightful to experiment and provide formal results across more datasets and dataset groupings to fully understand the effects of multi-SLAM. Test rigging (a challenge highlighted at the end of Section 3.5.2) for dataset sequences should be addressed for future related works.

In Section 3.5.2, we described our interest in supporting cross-thread relocalization, which is a good candidate for expanding the consistency and

completeness of multi-SLAM. While our existing multi-SLAM solution provides the ability to cross-reference generated maps and correct in real time, cross-thread relocalization has the potential to fill in gaps where previously, tracking and mapping might be lost for a large portion of a sequence.

This work focuses heavily specifically the adaptation of ORB-SLAM for multiple MAVs. With this domain restriction in place, it would be interesting to see an extension of the original ORB-SLAM library (as well as the multi-SLAM version) to accept things like heuristics on motion and location—for example, data coming from an IMU. This would not be too difficult to implement on top of the existing system—currently ORB-SLAM uses the previous frame information and assumes constant velocity to estimate where to look for new frame localization. This system would be an interesting starting point to incorporate a more (theoretically) accurate velocity model, based on estimated motion from IMU data.

Lastly, our experience with the Parrot AR.Drone 2.0s left us relatively unimpressed as time passed. We mention alternative, significantly cheaper MAVs that we were unable to pursue due to engineering restrictions, but supporting these cheaper (and ever-improving) MAVs is a key component to building out a robust network of not just two or three MAVs, but a full network. This enables real-world testing of the more complex challenges in multi-SLAM.

5.2 Contributions

The primary scientific contribution of this paper is the extension of ORB-SLAM to support multiple inputs while showing that mapping quality remains similar to that of the original library—thus opening up ORB-SLAM to the benefits of multi-entity systems as previously outlined. We attempted to demonstrate this in as scientific manner as possible, but also understand that this is perhaps a little less formal of a hypothesis/problem statement as compared to traditional scientific papers. We acknowledge that a good portion of this paper—the surrounding pieces (tooling, infrastructure definition, etc.)—are not a technical contribution and more of an engineering effort. However, we believe that describing the process and our experience around building out trials based on simulated and real-world MAVs will be very useful to future work in this domain.

5.3 Lessons Learned

On a more personal note and separate from the technical and formal contributions of this paper, I would like to take a moment to describe a few lessons learned throughout the thesis process.

5.3.1 Scope

In reflecting on the past few years, it is clear that this thesis grew significantly out of scope in terms of effort, knowledge, and experience needed. The (multi-part!) hypothesis of this thesis reads more like a long-term vision

for the field, not the focal point of a single Master's thesis. Finding the critical path—the minimum hypothesis that will still push the boundary of the field—is probably the most important scoping decision one can make when pursuing a thesis. Decisions down the line (do I implement x or y or z?) become much easier, as it is much simpler to answer the question "does this directly relate to my hypothesis?"

Getting specific in the context of this thesis, there was likely enough work to be done strictly around the challenges in adapting ORB-SLAM (and monocular SLAM systems in general) to support multiple inputs efficiently. The physical and simulated trials were not particularly necessary—those fall more under the umbrella of "long-term vision" and significantly increased the amount of investigation, tooling, and experimentation that was done.

5.3.2 Subject Matter Expertise

Another misstep that goes hand-in-hand with scope misestimation is understanding the effects of subject matter expertise and lab infrastructure. The papers that inspired this work (Computer Vision Group at TUM, UPenn's GRASP, ETH Zurich) are predominately coming from universities with robust UAV and/or computer vision labs with large amounts of expertise in one or multiple pieces of this work. The amount of subject matter expertise around you can inform how aggressive your scope can be—a lab that actively develops monocular SLAM or uses UAVs regularly in research will be much better poised to incrementally add on top of those subjects than those that do not.

I would be remiss if I did not mention that the aforementioned universities have actually done quite an excellent job in publishing significant amounts of their work and continue to maintain several open source projects. This effort did make getting started in the ecosystem somewhat straightforward—datasets, ROS adaptors, simulation tools, evaluation tools, the AR.Drone drivers, and so on.

5.3.3 Focus on the Measurable

Significant time was spent wiring pieces together, trialing different components of the finished product, but in the end, one of the most important components of the finished product was our benchmark analysis, which proved out the baseline ability to perform collaborative SLAM well. By prioritizing a functioning control system and simulation system early on, we were able to show that real world systems would in fact be able to coordinate in the way we wanted, but at the expense of dataset tooling (called out in Section 5.1) and more.

Appendix

Appendix 1

Supplementary Materials

We are extremely grateful to the individuals and groups that open sourced their work that made this work possible. We aim to release the code created in this process, including our extension to ORB-SLAM, the control system, and mapping server. The permanent, centralized link to view this code online will live at `https://github.com/ahollenbach/multi`.

Bibliography

[1] frontier-exploration. http://wiki.ros.org/frontier_exploration.

[2] joy. http://wiki.ros.org/joy.

[3] Phonedrone ethos. http://xcraft.io/phone-drone/. Accessed: 2016-06-17.

[4] Ilze Andersone. The characteristics of the map merging methods: A survey. *Scientific Journal of Riga Technical University. Computer Sciences*, 41(1):113–121, 2010.

[5] AndroFlight. Ar autoconnect. https://sites.google.com/site/androflight/arautoconnect. Accessed: 2016-06-17.

[6] AutonomyLab. Ardrone autonomy. https://github.com/AutonomyLab/ardrone_autonomy. Accessed: 2016-06-17.

[7] Michael Blösch, Stephan Weiss, Davide Scaramuzza, and Roland Siegwart. Vision based mav navigation in unknown and unstructured environments. In *Robotics and automation (ICRA), 2010 IEEE international conference on*, pages 21–28. IEEE, 2010.

[8] Roland Brockers, Martin Hummenberger, Stephan Weiss, and Larry Matthies. Towards autonomous navigation of miniature uav. In *Proceedings of the*

IEEE Conference on Computer Vision and Pattern Recognition Workshops, pages 631–637, 2014.

[9] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 2016.

[10] Andrew J Davison. Real-time simultaneous localisation and mapping with a single camera. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 1403–1410. IEEE, 2003.

[11] Oliver Dunkley, Jakob Engel, Jürgen Sturm, and Daniel Cremers. Visual-inertial navigation for a camera-equipped 25g nano-quadrotor. In *IROS2014 aerial open source robotics workshop*, 2014.

[12] Shoaib Ehsan and Klaus D McDonald-Maier. On-board vision processing for small uavs: Time to rethink strategy. *arXiv preprint arXiv:1504.07021*, 2015.

[13] Alberto Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

[14] J. Engel, V. Usenko, and D. Cremers. A photometrically calibrated benchmark for monocular visual odometry. In *arXiv:1607.02555*, July 2016.

[15] Jakob Engel and Daniel Cremers. Semi-dense direct slam. In *The IEEE International Conference on Computer Vision (ICCV) Workshops*, December 2015.

[16] Jakob Engel, Vladlen Koltun, and Daniel Cremers. Direct sparse odometry. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[17] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *Computer Vision–ECCV 2014*, pages 834–849. Springer, 2014.

[18] Jakob Engel, Jürgen Sturm, and Daniel Cremers. Camera-based navigation of a low-cost quadrocopter. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 2815–2821. IEEE, 2012.

[19] Christian Forster, Simon Lynen, Laurent Kneip, and Davide Scaramuzza. Collaborative monocular slam with multiple micro aerial vehicles. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 3962–3970. IEEE, 2013.

[20] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. Svo: Fast semi-direct monocular visual odometry. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 15–22. IEEE, 2014.

[21] Jorge Fuentes-Pacheco, José Ruiz-Ascencio, and Juan Manuel Rendón-Mancha. Visual simultaneous localization and mapping: a survey. *Artificial Intelligence Review*, 43(1):55–81, 2015.

[22] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[23] Slawomir Grzonka, Giorgio Grisetti, and Wolfram Burgard. A fully autonomous indoor quadrotor. *Robotics, IEEE Transactions on*, 28(1):90–100, 2012.

[24] Armin Hornung, Kai M Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[25] Hongrong Huang and Jürgen Sturm. tum_simulator. `https://github.com/tum-vision/tum_simulator`. Accessed: 2016-06-17.

[26] Niklas Karlsson, Enrico Di Bernardo, Jim Ostrowski, Luis Goncalves, Paolo Pirjanian, and Mario E Munich. The vslam algorithm for robust localization and mapping. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 24–29. IEEE, 2005.

[27] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *Mixed and Augmented Reality, 2007. ISMAR 2007.*

6th IEEE and ACM International Symposium on, pages 225–234. IEEE, 2007.

[28] Kurt Konolige, Dieter Fox, Benson Limketkai, Jonathan Ko, and Benjamin Stewart. Map merging for distributed robot navigation. In *IROS*, pages 212–217, 2003.

[29] THINC Lab. thinc_simulator. `https://github.com/thinclab/thinc_simulator`. Accessed: 2016-06-17.

[30] Giuseppe Loianno, Gareth Cross, Chao Qu, Yash Mulgaonkar, Joel A Hesch, and Vijay Kumar. Flying smartphones: Automated flight enabled by consumer electronics. *Robotics & Automation Magazine, IEEE*, 22(2):24–32, 2015.

[31] José Martínez-Carranza, Nils Loewen, Francisco Márquez, Esteban O García, and Walterio Mayol-Cuevas. Towards autonomous flight of micro aerial vehicles using orb-slam. In *Research, Education and Development of Unmanned Aerial Systems (RED-UAS), 2015 Workshop on*, pages 241–248. IEEE, 2015.

[32] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[33] Daniel Mellinger, Nathan Michael, and Vijay Kumar. Trajectory generation and control for precise aggressive maneuvers with quadrotors.

The International Journal of Robotics Research, page 0278364911434236, 2012.

[34] Nathan Michael, Daniel Mellinger, Quentin Lindsey, and Vijay Kumar. The grasp multiple micro-uav testbed. *Robotics & Automation Magazine, IEEE*, 17(3):56–65, 2010.

[35] Jack Morrison, Dorian Gálvez-López, and Gabe Sibley. Scalable Multi-device SLAM. In *Robotics Science and Systems. Workshop on Distributed Control and Estimation for Robotic Vehicle Networks*, July 2014.

[36] Christian Mostegel, Andreas Wendel, and Horst Bischof. Active monocular localization: towards autonomous monocular exploration for multirotor mavs. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3848–3855. IEEE, 2014.

[37] Raul Mur. evaluate-ate-scale. `https://github.com/raulmur/evaluate_ate_scale`. Accessed: 2017-03-10.

[38] Raul Mur-Artal, JMM Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *Robotics, IEEE Transactions on*, 31(5):1147–1163, 2015.

[39] Abdelkrim Nemra and Nabil Aouf. Robust cooperative uav visual slam. In *Cybernetic Intelligent Systems (CIS), 2010 IEEE 9th International Conference on*, pages 1–6. IEEE, 2010.

[40] Richard A Newcombe, Steven J Lovegrove, and Andrew J Davison. Dtam: Dense tracking and mapping in real-time. In *2011 international conference on computer vision*, pages 2320–2327. IEEE, 2011.

[41] BDW Remes, Dino Hensen, Freek Van Tienen, Christophe De Wagter, Erik Van der Horst, and GCHE De Croon. Paparazzi: how to make a swarm of parrot ar drones fly autonomously based 'on gps. In *IMAV 2013: Proceedings of the International Micro Air Vehicle Conference and Flight Competition, Toulouse, France, 17-20 September 2013*, 2013.

[42] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. IEEE, 2011.

[43] Sajad Saeedi, Michael Trentini, Mae Seto, and Howard Li. Multiple-robot simultaneous localization and mapping: A review. *Journal of Field Robotics*, 33(1):3–46, 2016.

[44] Kyle Simek. Dissecting the camera matrix, part 3: The intrinsic matrix. `https://ksimek.github.io/2013/08/13/intrinsic/`.

[45] J. Sturm. ardrone-joystick. `https://github.com/tum-vision/autonavx_ardrone/tree/master/ardrone_joystick`.

[46] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.

[47] Levi Valgaerts, Andrés Bruhn, Markus Mainberger, and Joachim Weickert. Dense versus sparse approaches for estimating the fundamental matrix. *International Journal of Computer Vision*, 96(2):212–234, 2012.