

Coupled Sensor Configuration and Planning with Unmanned Aerial Vehicles

A Major Qualifying Project Report
submitted to the Faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in Partial Fulfilment of the Requirements of the
Bachelor of Science Degree

in Aerospace Engineering

by

Alexandra Ballentine

Joseph Calomo

Jarrett Gulden








Peter Korfuzi

Jake Letourneau

Marina Nelson







Approved by:



Raghvendra V. Cowlagi, Advisor

in Aerospace Engineering and Computer Science

by

Thomas Lamar



Approved by:



Raghvendra V. Cowlagi, Advisor

Carlo Pincioli, Advisor

Fair Use Disclaimer: This document may contain copyrighted material, such as photographs and diagrams, the use of which may not always have been specifically authorized by the copyright owner. The use of copyrighted material in this document is in accordance with the “fair use doctrine” as incorporated in Title 17 USC §107 of the United States Copyright Act of 1976.

WPI Required Statement on Undergraduate Works: This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on the web without editorial or peer review.

Abstract

The goal of this project is to create an indoor laboratory environment for the collection of multi-modal datasets which may be used to validate path planning and sensor algorithms. The environment should include various sensors and sensor types, multiple robotic vehicles, and should use a motion capture system for localization. The main outcome of this MQP is a software framework capable of autonomously controlling the robotic vehicles, collecting data, and for wireless and wired communication between relevant entities in the experiment. The environment is relatively easy to operate for novice users, including PhD students in the advisor’s lab and is modular to allow for a wide variety of experiments. Data gathered from the MQP should be relevant in validation of coupled sensor configuration and planning algorithms, which means that path planning occurs simultaneously with sensor configuration.

Multiple robotic vehicles are developed—specifically wheeled vehicles and unmanned aerial vehicles (UAVs). The wheeled vehicles are solely coded to allow for proper navigation in the lab environment. The UAV is built, tested, and coded by the team. In the final environment, these robotic vehicles, multiple sensor types, and indoor localization are integrated into a software framework which can handle all communication, control, and data collection. A series of unit tests are conducted to validate the environment, code, and software framework. Successful tests demonstrate the feasibility of the software, the accuracy of the implemented control algorithms, and the validity of the UAV design.

Contents

1	Project Overview	1
1.1	Introduction	1
1.2	Project Objectives and Tasks	2
1.3	Literature Review	4
1.4	Design Requirements, Constraints, and Other Considerations	6
1.5	Project Management	6
1.6	Tasks and Timetables	7
1.6.1	A-Term	7
1.6.2	Execution	8
1.6.3	B-Term	8
1.6.4	C-Term	9
1.7	Relevant Engineering Standards	10
1.8	Methods	11
1.9	Broader Impacts	11
2	System Design	13
2.1	Project Objectives	13
2.1.1	Experiment 1: Computer Vision and UAV Flight	13
2.1.2	Experiment 2: UGV Navigation	14
2.1.3	Experiment 3: Multi-modal Sensors in a Dynamic Environment	14
2.2	Final UAV Design	15
2.2.1	Final Components	15
2.2.2	Wiring Diagrams	16
2.3	Wind Tunnel Test Design	17
2.4	Final UGV Design	17
2.5	Final Sensor Suite	18
2.6	Software	18
2.6.1	Modular Experiment Software System	18

2.6.2	Software System Integration	21
2.7	Lab Setup	22
2.7.1	VICON Motion Capture System Setup	22
2.7.2	Experimental Field Setup	23
3	Design Process and Analysis	25
3.1	UAV Selection and Development	25
3.1.1	Troubleshooting	25
3.1.2	Battery	25
3.1.3	Power Module and Power Distribution Board	28
3.1.4	Motors and Propellers	28
3.1.5	Electronic Speed Controllers (ESCs)	29
3.1.6	Frame	30
3.1.7	3D Printed Mounts	32
3.1.8	Flight Controller and Autopilot Software	33
3.1.9	Ground Station	34
3.1.10	Micro-Controller	35
3.1.11	Wi-Fi Board	36
3.1.12	Modularity	36
3.1.13	Safety	36
3.1.14	Electrical Diagram	37
3.2	Wind Tunnel Testing	38
3.2.1	Mathematical Modeling	38
3.2.2	Wind Tunnel Setup	40
3.2.3	Data Collection	41
3.3	UGV Selection and Development	42
3.3.1	AWS DeepRacer	42
3.3.2	TurtleBot3 Burger and Waffle Pi	42
3.4	Sensing Suite	46
3.4.1	Design Requirements	46
3.4.2	Sensor Evaluation and Selection	47
3.5	Software	49

3.5.1	Modular Experimental Software System (MESS)	49
3.5.2	Robot Operating System (ROS) Integration	56
3.5.3	Software System Integration	58
3.5.4	UAV Computer Vision	60
3.6	Experimental Lab Set	65
3.6.1	Hardware	65
4	Results	69
4.1	Experiment 1 Results	69
4.1.1	Threat Mapping with Ideal Conditions	69
4.1.2	Threat Mapping with Non-Ideal Conditions	71
4.2	Experiment 2 Results	72
4.2.1	UGV Vertex Navigation using OpenCR1.0 Odometry	72
4.2.2	UGV Line Following in the VICON Environment	73
4.2.3	UGV Vertex Navigation in the VICON Environment	74
4.2.4	Multi-UGV Vertex Navigation in the VICON Environment	76
4.3	Experiment 3 Results	77
4.3.1	Experiment Setup	77
4.3.2	Sensor Data	77
4.4	UAV Flight Testing	79
4.4.1	Indoor Flight	79
4.4.2	Autonomous flight (in SITL)	79
4.5	Wind Tunnel Results	80
4.5.1	Data Processing	80
4.5.2	Data Analysis	83
4.5.3	Conclusions and Future Work	84
5	Conclusions	85
5.1	Conclusions	85
5.2	Future Work	85
A	Raspberry Pi and Sensor Configuration	90
B	Raspberry Pi and Flight Controller Configuration	95

Acknowledgements

The team would like to thank our advisors and a few key people. Professor Cowlagi guided us throughout the project and provided key feedback and advice when we reached roadblocks. Professor Pincirolì provided useful feedback on ROS as the software implementation. Prakash Poudel and Jeffrey DesRoches helped us realize the desired outcomes for this MQP and the immediate use cases. Keval Shah taught the team how to calibrate and use the VICON motion capture system. Michael Beskid allowed the team to use the network from his research during initial testing. Finally, Professor Olinger provided invaluable help to realize wind tunnel testing of the UAV.

Table of Authorship

Section	Project Work	Primary Author	Editor
1.1 Introduction	-	AB	JL, MN, PK
1.2 Project Objectives	-	AB	JL, MN, PK
1.3 Literature Review	All	AB, PK	JL, MN
1.4 Design Requirements, Constraints, & Considerations	-	AB	
1.5 Project Management	All	AB	JL, MN
1.6 Tasks and Timetables	All	AB	JL, MN
1.7 Relevant Engineering Standards	-	AB, MN	
1.8 Methods	-	AB	
1.9 Broader Impacts	-	PK	
2.1 Project Objectives	All	All	PK, AB
2.2 Final UAV Design	AB, JG, JL	AB, JG, JL	
2.3 Wind Tunnel Test Design	JC	JC	
2.4 Final UGV Design	MN	MN	
2.5 Final Sensor Suite	PK	AB	
2.6 Software	TL	TL	
2.7 Lab Setup	JC, JG, PK	JG, PK	
3.1 UAV Selection and Development	AB, JG, JL	AB, JL	JC, JG
3.2 Wind Tunnel Testing	AB, JC, JG, JL	JC	
3.3 UGV Selection and Development	MN	MN	
3.4 Sensing Suite	PK, MN	PK	
3.4 Software	TL	TL, MN	
3.5 Experimental Lab Setup	JC, JG	JC, JG	
4.1 Experiment 1 Results	MN	MN	
4.2 Experiment 2 Results	MN	MN	
4.3 Experiment 3 Results	ALL	TL	

4.4 UAV Flight Testing	AB, JG, JL	AB, JL	
4.5 Wind Tunnel Results	JC	JC	
5.1 Conclusions	-	AB, TL, JG	
5.2 Future Work	-	JG, TL	
LaTeX Formatting	-	All	AB

1 Project Overview

1.1 Introduction

Unmanned aerial vehicles (UAVs) have benefits in many civilian and military applications including search and rescue operations, natural disaster and climate monitoring, and surveillance. UAVs can reach remote and inaccessible areas, making them suitable for tasks that may be difficult or impossible from the ground. For instance, UAVs may be used to determine the severity of flooding and map safe routes for emergency vehicles. These vehicles may operate at various levels of autonomy, ranging from fully tele-operated to fully autonomous. The primary function of autonomy is to remove or decrease the need for remote piloting. An onboard system may correct a path to minimize the risk of a collision or even determine the best route to a predetermined destination [1].

A key component of autonomy is *path planning*, which refers to finding the “best” route to a destination. Potential criteria to identify the best path include distance, duration, obstacle avoidance, and threat avoidance. Another critical aspect is *remote sensing*, which refers to acquiring data via sensor readings. Some quantities of interest may be images, chemical composition of air, and terrain topography. Remote sensing may have the sole purpose of gathering data for future use, such as climate monitoring. However, remote sensing may also provide the necessary information for path planning [2].

A recent innovation in this field, coupled sensor configuration and path planning (CSCP), allows for faster and more accurate path planning. Generally, path planning and sensor configuration are uncoupled, meaning that gathered sensor data is collected and then passed to a path planning algorithm. In two-dimensional path planning, the challenge of a robotic vehicle navigating from a starting location to a destination models this process. The vehicle might first map all the obstacles between its location and its desired location. After doing so, it determines the best possible path. In CSCP, sensor positioning and path planning occur simultaneously. The control algorithm determines the most relevant information and directs the sensors accordingly, making decisions while it learns about the environment. Consequently, the vehicle only gathers the most applicable information, which decreases processing requirements for sensor data.

This project aims to create a modular experimental system to verify coupled sensor configuration and autonomous path planning theory and algorithms. Once completed, the developed lab set-up will be invaluable to future researchers, including PhD students at the university. While algorithms may originate

from simulations, real-world experimentation provides critical validation. Therefore, our experimental setup can become invaluable to future users. We focus on configurability to allow users to add or replace sensors, change the number of active vehicles, and adjust the experimental scenario with minimum software development.

1.2 Project Objectives and Tasks

This project aims to design and run a set of experiments that demonstrate the modular, configurable capabilities of the lab and software infrastructure that we design and develop. The purpose of this experiment is to verify CSCP theory and algorithms, upon other applications, and will involve the integration of various sensors and vehicles according to a desired experimental scenario. To that end, the expected outcomes encompass the creation of individual vehicles as well as holistic integration. The following list details the expectations:

1. Flight demonstration of at least one UAV, which is capable of multiple sensor payloads, indoor flight, and communication with a ground station to relay sensor data.
2. Identification of aerodynamic and flight mechanical models of the UAV, followed by validation through flight testing.
3. Demonstration of a fleet of 2 UAVs with the above capabilities.
4. Demonstration of autonomous flight of at least one UAV, including obstacle avoidance.
5. Development of on-board path planning for at least one wheeled robotic vehicle.
6. Design and implementation of at least one experimental set-up that is capable of satisfying the project goals, including the installation of multi-modal sensors and multiple robotic vehicles.
7. Demonstration of a software framework that can do the following: collect and store data, facilitate wired and wireless communication between entities, and control wheeled vehicles and UAVs.

Based on these expectations and stakeholder interviews with our faculty advisor and PhD student advisors in the Autonomy, Controls, and Estimation (ACE) Lab, we formulate a statement of purpose for the project. The project aims to create and demonstrate a modular lab set-up to test CSCP theory and algorithms. This imposes two main guidelines on our work: to develop all necessary vehicles, sensors, and communications; and to create a set of scenarios that will allow us to integrate multiple vehicles and multi-modal sensors. To address this first goal, we develop two UAVs, UGVs, and infrastructure to gather

data from multi-modal sensors. Additionally, we create a software package that can coordinate actions and data collection between all involved vehicles and sensors. In conjunction with these overarching goals, we also flight-test a UAV in a wind tunnel to describe its aerodynamic characteristics. Advisor and stakeholder feedback allow us to identify two key scenarios. The first critical scenario is that of mapping a threat field. We realize our second scenario should mimic ESCAPE, which is discussed in more detail in the literature review. To obtain these goals, we propose five high-level tasks to be completed over three terms.

The overarching tasks are as follows: design the experiment, orient ourselves in the lab, develop a quadcopter and aerial fleet, select and familiarize ourselves with a model of ground vehicle, and develop a modular software framework. The first task lays out the steps to understand the project and its final goals. In this process, we meet with the PhD student advisors—Jeffrey DesRoches and Prakash Poudel, determine experiment design criteria, determine necessary sensor types, and finally propose the experimental design. To accomplish the second task, we familiarize ourselves with the provided equipment, specifically the VICON motion capture cameras. We mount and calibrate these cameras for future use with the experiments.

The third task leads us to the analysis and troubleshooting of prior MQP UAVs. We achieve stable flight, and then use the developed quadcopter to advance our project. This quadcopter becomes capable of waypoint following and obstacle avoidance. We then build a second quadcopter and determine the aerodynamic coefficients of these UAVs. In parallel with this task, we evaluate and configure different ground vehicles. These vehicles are attached to VICON for use as an indoor positioning system and to demonstrate autonomous path planning. Further, the ground vehicles selected are retrofitted with any necessary sensors. Finally, we develop a modular software framework that can communicate with all necessary vehicles, interface with sensors, manage data, and implement necessary controllers.

We then develop a task schedule—shown in Table 1.1—that describes milestones throughout our MQP.

Table 1.1: Main project tasks and milestones

Task	A-Term	B-Term	C-Term
Task #1: Design Experiment	Meet with stakeholders; Design experiment		
Task #2: VICON Calibration	Familiar ourselves with lab space; Begin mounting cameras	Mount and calibrate all cameras	

Task #3: Quadcopter & Fleet Development	Analyze existing UAVs	Achieve stable flight	Develop waypoint following and obstacle avoidance; Build a second UAV; Determine aerodynamic coefficients
Task #4: Ground Vehicles	Determine ground vehicle type	Select one ground vehicle and develop basic motion control; Use VICON as indoor positioning	Demonstrate autonomous path planning
Task #5: Sensor Integration	Determine options for a complete sensor suite	Determine communication protocol for sensors	Conduct sensor research and develop code to analyze sensor outputs
Task #6: Modular Software	Outline software requirements, framework, and useful libraries	Finish determining software structure and begin writing and testing code	Complete software code and integrate software and components

1.3 Literature Review

As UAV technology continues to evolve, the demand for new algorithms and data processing methods has grown. Today, UAVs have been incorporated into more and more applications, such as search and rescue, remote surveying, and industry. CSCP presents a novel method of handling large amounts of possible sensor data. We investigate prior work regarding sensor data collection and analysis to orient the reader regarding current technology and limitations. Key topics include sensor fusion, dynamic sensor activation, simultaneous localization and mapping (SLAM), and autonomous path planning algorithms. In the last category, we consider interactive planning and sensing (IPAS), and prior work done regarding CSCP. Finally, we consider experiments conducted to validate sensor and path-planning algorithms.

Sensor fusion is the process of combining measurements and data taken from multiple sensors. These sensors may be of different types and often have different characteristics—sampling rate, accuracy, and data type. Typical sensor types include inertial measurement units (IMUs), visual cameras, LiDAR, and radar. An example of sensor fusion may be an experiment that considers three sensors, each of which varies in characteristics. The challenge of sensor fusion is weighing the relevance and accuracy of these independent measurements [3]. However, there are generally processing limitations that restrict data collection and analysis. Yin and Lafortune [4] proposed a policy that will turn on and off sensors to accomplish a goal, for

instance, to control a system. Ideally, this sensor activation policy should optimize sensor input with respect to system characteristics. While the details of this policy are not relevant to this paper, it is important to note the need for a rule that limits energy and bandwidth consumption.

Once acquired, sensor data may be used in many different ways, such as SLAM. In SLAM, the vehicle does not know its surroundings, so it must map them while navigating the environment. Previously, there have been many iterations of SLAM using various sensors, including sonar sensors, IR sensors, laser scanners, and visual cameras. SLAM algorithms track identified features to both create a map and localize the vehicle within its known map. This technique is often used as a complement to autonomous path planning. Further, it can help eliminate error sources since the vehicle is constantly updating its position and recognizes when it has returned to a previously visited location [5]. Recently, SLAM has been adopted for UAVs, especially in GPS-denied environments. Onboard a UAV, it is computationally expensive to store a predetermined map of the environment. Instead, UAVs can rely on SLAM to maintain a local map, including only relevant landmarks. SLAM creates opportunities for further applications, including applications when GPS may be unavailable or unreliable and instances in dynamic environments [6].

Algorithms that extend SLAM's capabilities are IPAS and CSCP. SLAM restricts itself to localization and mapping. Both IPAS and CSCP develop an understanding of the surrounding environment for path planning. In IPAS, the algorithm places a finite number of point-wise sensors to provide the best information for path planning. Before moving into the environment, a few sensors might be placed according to the algorithm, and the vehicle would analyze the results. A use case for this might be in a threat field, where the vehicle tries to navigate with minimum exposure. IPAS has been shown to reduce the computational cost of path planning while converging closer to the absolute best path [7]. CSCP couples these two steps. In IPAS, the sensor measurements are taken first, while in CSCP, the sensor measurements are taken simultaneously with the path planning algorithm. This allows the algorithm to request new information during a mission as needed [8].

Recent advancements in sensor fusion require data sets for experimental validation. The Air Force Research Laboratory conducted an experiment to test and validate multi-modal sensor and data fusion methods. From this experiment, they released a data set—Experiments, Scenarios, Concept of Operations, and Prototype Engineering (ESCAPE)—that brings together six sources of data over various outdoor scenarios. The ESCAPE data set focuses on vehicle tracking while including typical outdoor disturbances [9]. In this MQP, we aim to provide an indoor dataset for similar research purposes. The goal of autonomous path planning is generally to do more—create more efficient navigation plans—with less data. This demand comes from limitations on real-time processing information and bandwidth. This MQP will provide a small-

scale experimental set-up that can be used for the validation of algorithms that will advance autonomous path planning in the future.

1.4 Design Requirements, Constraints, and Other Considerations

A few primary design requirements constraint the outcome of the MQP. Of those, the most important are the requirements outlined in Section 1.2. The final configuration of the experiment should the needs of ACE Lab graduate researchers. This imposes the requirement of being able to collect sensor data and configure and run multiple vehicles.

On the subsystem level, each actor vehicle, sensor, and software component are subject to their own constraints. Both the UAV and UGV must send and receive wireless communications without significant network delays and must navigate consistently and reliably. The UAV must support a variety of sensors and a Raspberry Pi to send and receive sensor data. Regarding sensors, these must either be suitable for a fixed location or be small and light enough to install on the UAV. Each sensor must send and receive data in such a way that is compatible with the developed software system.

The software system must be self-sufficient and takes a minimum number of user inputs. These inputs must be enough that the software can run a variety of different experiments, but they must be sufficiently limited that the software is easy to use. We assumed ease of use to follow these parameters: someone with limited coding knowledge can configure and run an experiment. This does not mean that there must be no coding involved: simply there should be as little as possible. In practicality, the user will likely input a JSON file with specifications, but they will not have to open and modify aspects of the code (for a typical use case).

Since the lab requires key card access, the only safety concerns pertain to authorized, educated users. As a result, only the UAV is considered as a potential threat. The secondary lab setup concern entails the integrity of the VICON cameras. These cameras must be mounted securely on the wall, out of reach of tampering by an incautious user (reducing the potential for damage or accidental miscalibration), and must be sufficiently protected from all actor vehicles in the experiment.

1.5 Project Management

We divided tasks among team members as much as possible. Through the first part of the project, we worked mostly synchronously to plan our experimental scenarios and to understand the complete scope of the project. Once this was accomplished, we began to split into specialized sub-teams:

We split into sub-teams to allow members of the MQP to specialize in areas and therefore provide

Table 1.2: Team member responsibilities for each part of the project

Task	A-Term	B-Term	C-Term
Task #1: Design Experiment	ALL	-	-
Task #2: VICON Calibration	JC, JG	JC, JG	-
Task #3: Quadcopter & Fleet Development	AB, PK, JL, MN	AB, JG, JL	AB, JC, JG, JL
Task #4: Ground Vehicles	MN	MN	MN
Task #5: Sensor Integration	PK	PK	PK, MN
Task #6: Modular Software	TL	AB, JC, TL	JC, TL, MN

targeted efforts. Within each sub-team, the members coordinated times to work on their parts of the project and to make progress. We met as a whole team twice weekly for most of A-term with our advisors. We also held weekly team meetings for the first half of the term to discuss our proposed outcomes for the project. In B- and C-term, we met once weekly with our advisors. We held all team meetings periodically, specifically when we needed to meet before a deadline or to make a major decision. Outside of these times working together, we used text messaging as our primary internal communication method. To communicate with our advisors and store important team documents, we used Microsoft Teams and Microsoft SharePoint.

1.6 Tasks and Timetables

At the beginning of each term, we set out a proposed timetable for what we wanted to accomplish. In this report, we show our actual timeline and comment on the differences between our desired and actual timelines.

1.6.1 A-Term

The Table 1.3 shows a weekly breakdown of our goals in A-Term. Originally, we wanted to have the VICON cameras fully mounted and the UAV flying by the end of A-term. Due to delays and unforeseen circumstances, we set less ambitious goals. In the lab, we found that we needed to wait for materials to arrive and that some of our original plans were not optimal. For instance, the studs in the lab are made of metal, meaning that we had to shift to drywall mountings for the cameras. Additionally, we were expecting that we could easily connect the UAV and begin working where the 2022 MQP had left off. However, we had issues consistently connecting the UAV from Mission Planner and QGroundControl, two well-known software options for UAV flight control. Once we were able to connect, we found new problems not mentioned in the 2022 MQP report.

Table 1.3: A-term actual timeline

Week	Task #1: Design Experiment	Task #2: VICON Calibration	Task #3: Quadcopter Development	Task #4: Ground Vehicles	Task #6: Modular Software	
Week 1	Propose tasks and outcomes.					
Week 2	Refine tasks and outcomes.					
Week 3	Design experimental scenarios	Determine a system to wall-mount the VICON cameras and manage cables			Define software requirements and outcomes	
Week 4	Refine experimental scenarios		Connect to the UAV and begin flight testing		Establish a framework for communication with vehicle controllers and sensors	
Week 5		Acquire parts and measure the lab	Troubleshoot the UAV			
Week 6		Mount VICON cameras				Research necessary integration with existing software
Week 7					Acquire an AWS DeepRacer to determine if it is suitable	

1.6.2 Execution

1.6.3 B-Term

Table 1.4 shows our progress throughout B-Term. Overall, we met most of our goals and exceeded some. We successfully on-boarded a UGV and performed a wall-following experiment. Further, we demonstrated the sensing capabilities and data collection of a Raspberry Pi camera. Originally, we hoped to finish the UAV this term and begin our experiments. Our main roadblock with the UAV was achieving stable flight. Without stable flight, we could not substantially advance any of the outcomes requiring the UAV. Aside from this setback, we accomplished our goals.

Table 1.4: B-term actual timeline

Week	Task #2: VICON Calibration	Task #3: Quadcopter Development	Task #4: Ground Vehicles	Task #5: Sensor Integration	Task #6: Modular Software
Week 1	Mount VICON cameras	Check UAV components and order new parts	Determine if AWS DeepRacer is suitable.	Acquire a Raspberry Pi and camera	Installation of necessary external software
Week 2		Conduct thrust stand testing and analyze the UAV frame		Connect RPi and camera and begin unit testing	
Week 3	Calibrate VICON cameras	Rewire UAV onto new parts and test new configuration	Setup ground station desktop.	Determine how to communicate	Establish MESS use cases and system needs
Week 4			Develop waypoint navigation ROS package for TurtleBot3.	Set up ROS environment	Complete MESS software architecture planning
Week 5		Determine the reason for unstable flight	Gather necessary external libraries		
Week 6			Develop line-following ROS package for TurtleBot3 and collect experimental data using VICON feedback		Implement VICON Bridge
Week 7					Code software solution that implements planned software architecture
Week 8					

1.6.4 C-Term

Table 1.5 shows our accomplishments each week of C-Term. By the end of the term, we met most of our project goals. We succeeded in running multiple experiments to validate our configuration and coding. However, we were unable to connect the flight controller to mavros for autonomous flight.

Table 1.5: C-term actual timeline

Week	Task #1: Implement Experiment	Task #3: Quadcopter Development	Task #4: Ground Vehicles	Task #5: Sensor Integration	Task #6: Modular Software
Week 1		Assemble UAV with new components; simulate a UAV mission with ArduPilot	Add code to calibrate TurtleBot sensors for occlusion from VICON		Create a basic GUI; test class structure
Week 2	Construct a lab environment	Fully develop code to model a simple mission in simulation	Test TurtleBot code in the lab environment	Map recorded images to lower resolution	Implement basic mission planning capabilities
Week 3		Finish assembly and calibrate both UAVs	Fix calibration issues and improve logging capabilities	Connect multiple images together; begin using RPi NoIR camera	Create functionality for user to save and load experiments
Week 4		Calibrate UAV; connect the flight controller to mavros; begin wind tunnel testing	Implement launch files to allow for vehicle differentiation	Map visual images to threat intensity readings	Add support for custom messages and creating launch files
Week 5	Improve lab environment; conduct Experiment 1 trials	Troubleshoot UAV; Complete wind tunnel testing	Build and configure new ground vehicles	Improve vision mapping	Improve functionality to launch ROS and collect data
Week 6					
Week 7	Conduct Experiment 3	Obtain stable flight; troubleshoot connection to mavros			Use software to run experiment

1.7 Relevant Engineering Standards

Robot Operating System (ROS) is an open-source framework standard used for control and communication management in robotic applications. A ROS environment consists of nodes that serve different operational purposes. Each node publishes and subscribes to topics that communicate data as

ROS messages. We use the ROS Noetic distribution to control vehicles and communicate data between a ground station, vehicles, and sensors.

PX4 and QGroundControl are open-source UAV software. PX4 is a flight controller software that we upload to the UAVs for onboard stabilization and mission planning. QGroundControl is a software that we use as a ground station to handle UAV calibration and testing.

1.8 Methods

Table 1.6: Sections of the project and corresponding methods

Subsystem	Process	Method
UAV	Creating secure wiring connections	Soldering
UAV	Designing and printing frame mounts	SOLIDWORKS and 3D printing
UAV	Analyzing wind tunnel data	MATLAB
UGV	Analyzing performance and paths	MATLAB
Sensors	Image processing	MATLAB

1.9 Broader Impacts

The development and advancement of UAV technology and navigation techniques lead to several impacts over economic, environmental, and social contexts. As innovations in autonomy progresses, UAVs will become more prevalent in societies and will disrupt many sectors of industry. This growing market carries with it noticeable social impact: increased search and rescue capabilities, rural access to basic needs [10], autonomous infrastructure inspections [11], and many more. In 2023, Zipline, an autonomous UAV-based company operating an expansive logistics and delivery system, released a health impact report highlighting how its fixed-wing, autonomous UAVs have delivered 1.5 million vaccines in parts of Nigeria with high rates of zero-dose children and contributed to a 67% reduction of blood products wasted in their Rwandan operating regions [12]. Whether it is delivery, monitoring, or communication systems, autonomous UAVs play key roles in driving social impact. However, socioeconomic concerns exist as well. Mohammed Yagot and Brenno Menezes suggest that autonomous UAV capabilities render tasks once difficult to standardize now able to be standardized, resulting in a new level of automation. Similar to the industrial revolution, this can result in a loss of jobs or migration of labor markets [13]. Admittedly, while autonomous UAV technologies can augment human jobs as much as replace them, there

is still the issue of income disparity as low-skilled worker incomes stagnate or decrease in a booming economy enabled by highly-skilled workers that develop and leverage these innovations [13]. Ultimately, disruptive technologies like autonomous UAVs introduce both benefits—such as improved safety—and drawbacks—such as exacerbated economic disparities. Mitigation efforts are active talking points in the literature as policies, regulations, and best practices emerge and evolve with the evolution of this technology.

2 System Design

2.1 Project Objectives

Through information gathered from stakeholders, we design experiments to both allow for incremental testing and integration and to satisfy the required outcomes. We prioritize phased testing to manage the complexity of the deliverable. In this section, we present the three experiments that guide the system design of this project, including component selection.

2.1.1 Experiment 1: Computer Vision and UAV Flight

In the first experiment, we demonstrate a UAV's flight capability and a visual sensor by recording a threat field. We define this threat field as a grid arrangement of differently colored squares. This phase requires a single UAV equipped with a visual-light camera. The vehicle should photograph the testing environment—a grid of construction paper. From this experiment, we measure a grid of RGB values to recreate a threat field. For details regarding the actual implementation of this experiment, refer to Section 4.1.

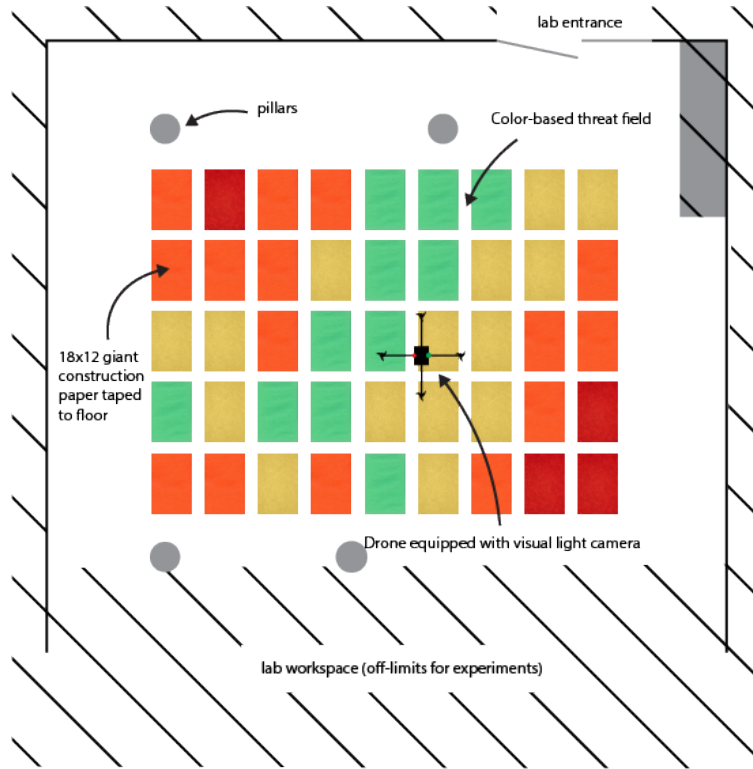


Figure 2.1: Experiment to validate the threat field identification capabilities of the UAVs.

2.1.2 Experiment 2: UGV Navigation

We design the second experiment to demonstrate the autonomous path-planning ability of an unmanned ground vehicle (UGV) in an obstacle field. This phase ensures familiarity with the UGV and can operate independently of the first phase. The obstacle field should consist of three-dimensional objects, such as cardboard boxes, set on the ground in a discrete grid arrangement. Rather than sensing the obstacle field, the UGV knows the location of all obstacles. We design this phase with future criteria in mind, such as data streaming a UAV-sensed obstacle field to a UGV and capabilities for a UGV to reach a dynamic target. For details regarding the implementation of this experiment, refer to Section 4.2.

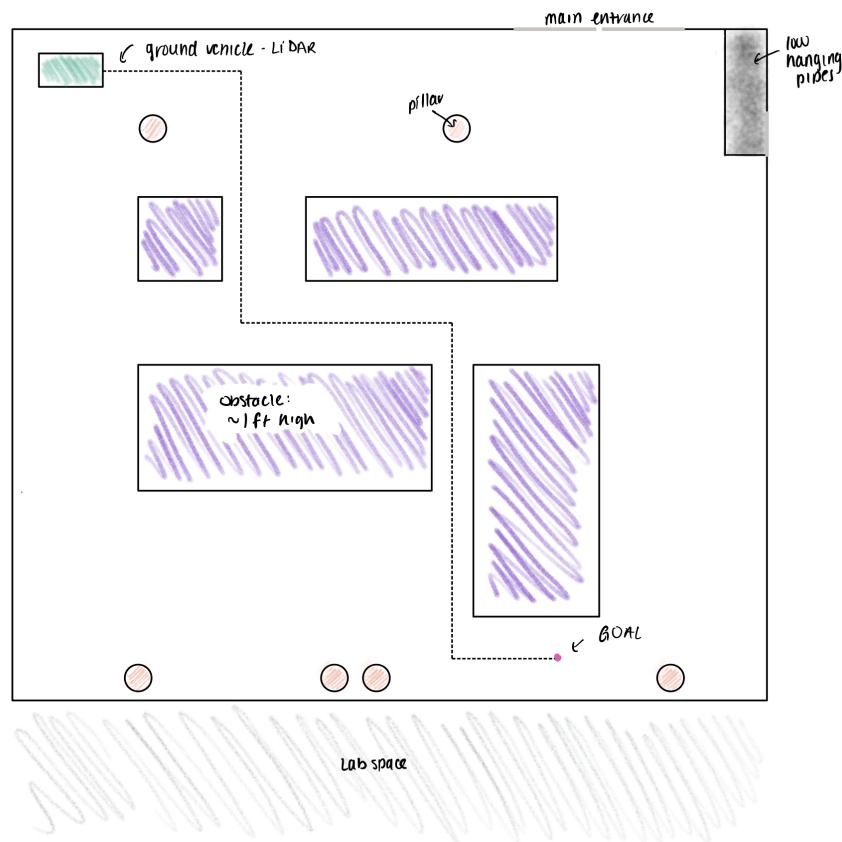


Figure 2.2: Experiment to verify the ability of a ground vehicle to autonomously navigate through an obstacle field.

2.1.3 Experiment 3: Multi-modal Sensors in a Dynamic Environment

Our final experiment aims to combine the previous two phases. This phase has multiple ground vehicles navigating on pre-determined paths. We equip one UGV with a heating element to give it a unique heat signature. Two UAVs fly within the environment, one equipped with an infrared camera and another with a visible light camera. We also use a static visual light camera to collect video. For details regarding the implementation of this experiment, refer to Section 4.3.

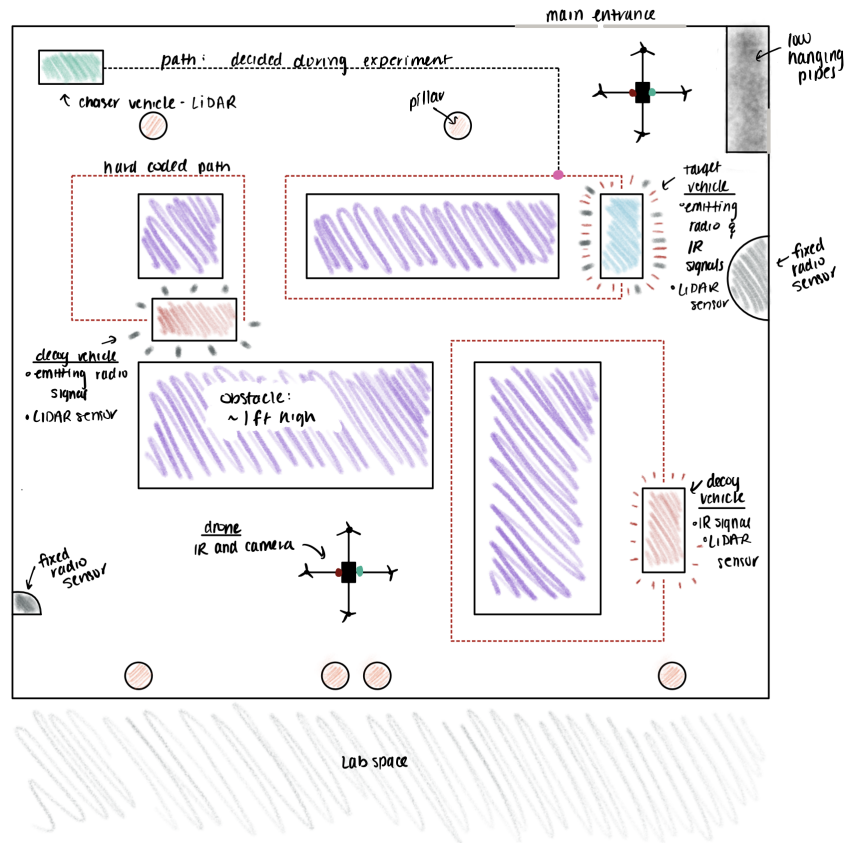


Figure 2.3: Experiment to validate our software system, robotic vehicles, and lab environment

2.2 Final UAV Design

2.2.1 Final Components

The final UAV should be capable of multiple sensor payloads, a reasonable flight time, and stable flight. For more details of the selection of components, refer to Section 3.1. Table 2.1 details the final components we choose.

Table 2.1: Final UAV Components

Components	Specification
Frame	250MM Quadcopter QAV250 Drone Frame (Carbon Fiber)
Frame Standoffs	35 mm M3 Female Threaded Hex Standoffs
Fasteners	M3x16x16 mm M3x19x19 mm M3x20x20 mm M3x30.5x30.5 mm
Propellers	5 inch diameter, 3 inch pitch, 3 blades

3D Printed Mounts	For power distribution board, power module, microcontroller, and flight controller
Bullet Plugs	2 mm male and female
Heat Shrink	Heat shrink tubing kit of various diameters as well as 3/8 inch tubing
Flight Controller	ARKV6X, with bus
Microcontroller	Raspberry Pi
Power Module	ARK PAB Power Module
Battery	Zeee 5200mAh 50C 11.1V RC Lipo Battery
Power Distribution Module	Matek PDB-XT60
Motors	T-MOTOR MN2206 KV2000 Brushless Electric Motors
Electronic Speed Controllers (ESCs)	Hobbypower Brushless 20A BLheli-S ESC Oneshot125
ESC Wires	20AWG Silicone wire
Servo Adapter	Servo Adapter: ARK Electronics
Extension Cable	2 Pairs ShareGoo 10cm 100mm XT60 Male Female Connector Plug RC with 12AWG Silicon Cable Wire
Battery to Module Adapter	Deans T Plugs to XT60 Adapter Connector Male Female for RC Lipo Battery Charger

2.2.2 Wiring Diagrams

Figures 2.4 and 2.5 show the wiring diagram for our UAV. Figure 2.4 shows the proper wiring for the power module, power distribution board, and the motors. Figure 2.5 shows the proper connection between the flight controller, servo adaptor, and the on-board Raspberry Pi 3. Note that the data-streaming connection between the flight controller and Raspberry Pi is correct according to documentation, but we are unable to experimentally validate this connection. For more information, see Section 4.4.2.

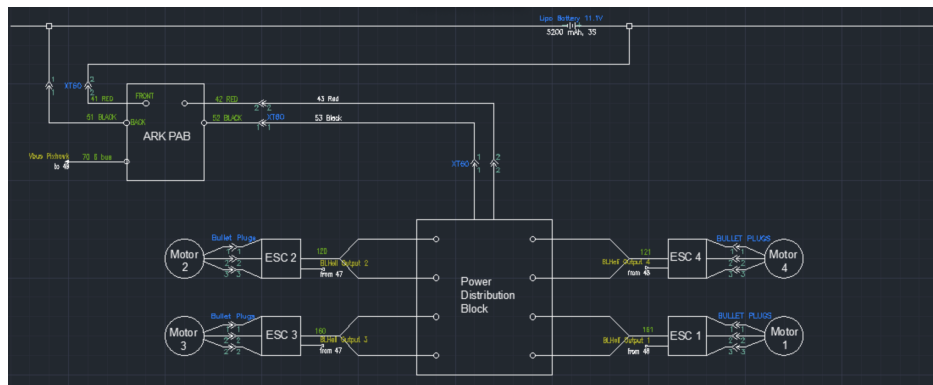


Figure 2.4: UAV Wiring Diagram for Power Distribution

environment. Table 2.2 details the final components of the TurtleBot3 Burger and Table 2.3 details the final components of the TurtleBot3 Waffle Pis.

Table 2.2: Final TurtleBot3 Burger Components

Components	Specification
Battery	Zeee 2200mAh 35C 11.1V RC Lipo Battery
Microcontroller	Raspberry Pi 3 Model B
Sensors	360 Laser Distance Sensor LDS-01

Table 2.3: Final TurtleBot3 Waffle Pi Components

Components	Specification
Battery	1800mAh 11.1V Lipo Battery
Microcontroller	Raspberry Pi 4 Model B/4GB
Payload	2x Plate Support M3x35mm 12V 12W Flexible Polyimide Heater Plate
Sensors	360 Laser Distance Sensor LDS-02 Raspberry Pi Camera Module 2

The full parts list is found in ROBOTIS’s TurtleBot3 e-Manual [14]. Two of the four TurtleBot3 Waffle Pis house the payload suite. The remaining two TurtleBot3 Waffle Pis utilize the Raspberry Pi Camera Module 2 Noir.

2.5 Final Sensor Suite

Table 2.4 shows our final sensor suite. For more details on our selection criteria, sensor considerations, and justifications regarding these choices, refer to Section 3.4.

Table 2.4: Final Sensor Selection

Sensor Type	Sensor Model
Visual Light	Raspberry Pi Camera Module V2
Infrared	Raspberry Pi Camera Module NoIR V2
Radio Transmitter	Ettus Research USRP B200mini
Radio Receiver	ADALM-Pluto SDR

2.6 Software

2.6.1 Modular Experiment Software System

We aim to create a custom software to assist in running the experiments. The software must be a centralized application to plan experiments, collect data, and command vehicles. To provide future

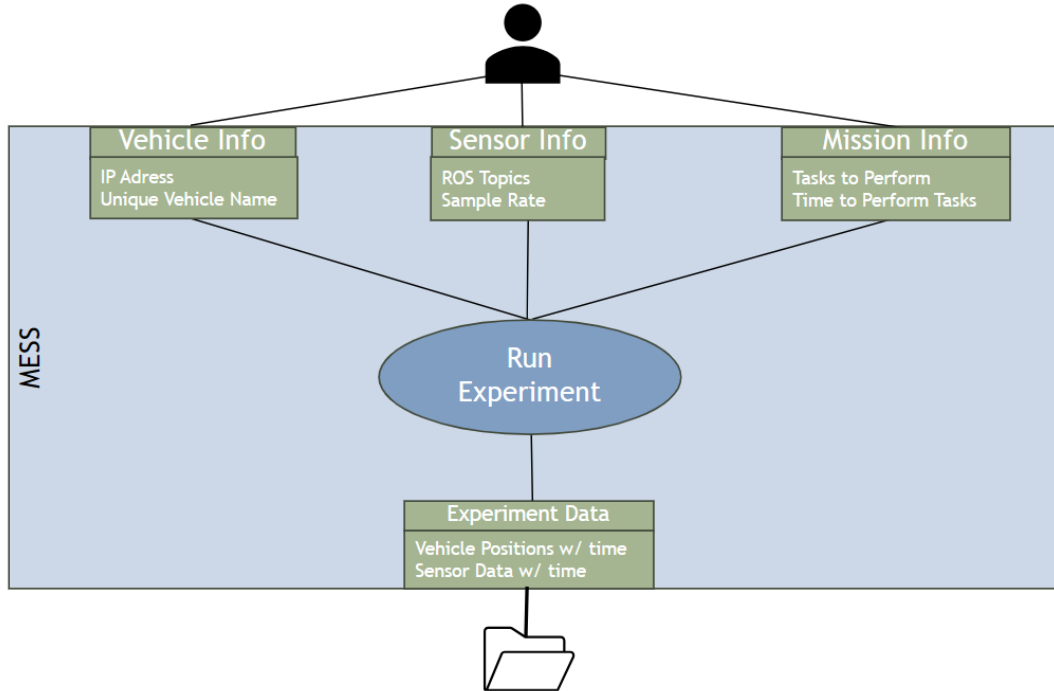


Figure 2.7: MESS Input and Output Diagram

versatility, the software should be modular to allow a user to easily set up and perform experiments with different vehicles and sensors. Due to its modular nature, we named this application the Modular Experimental Software System (MESS).

2.6.1.1 Design Requirements

The software design must take into account three system requirements: mission planning, centralized data collection, and vehicle command.

Mission Planning The first and primary goal of the MESS is to serve as a centralized location to plan and run experiments while minimizing the need for the user to write new code. To achieve this, certain inputs are required from the user. We decide the user should provide information on the UAVs, UGVs, sensors, and mission plan. This is visualized in Fig. 2.7.

A mission plan consists of tasks and times to perform those tasks. A task is an action to be performed by a sensor or a vehicle. Having a UAV takeoff, a UGV navigate to a way point, or a camera take a picture are all examples of a task.

Centralized Data Collection The MESS serves as a centralized location for data collection. Because experiments involve multiple vehicles and sensors, pulling and synchronizing data from each source separately would be time-intensive and cumbersome. The MESS solves this problem by accumulating all

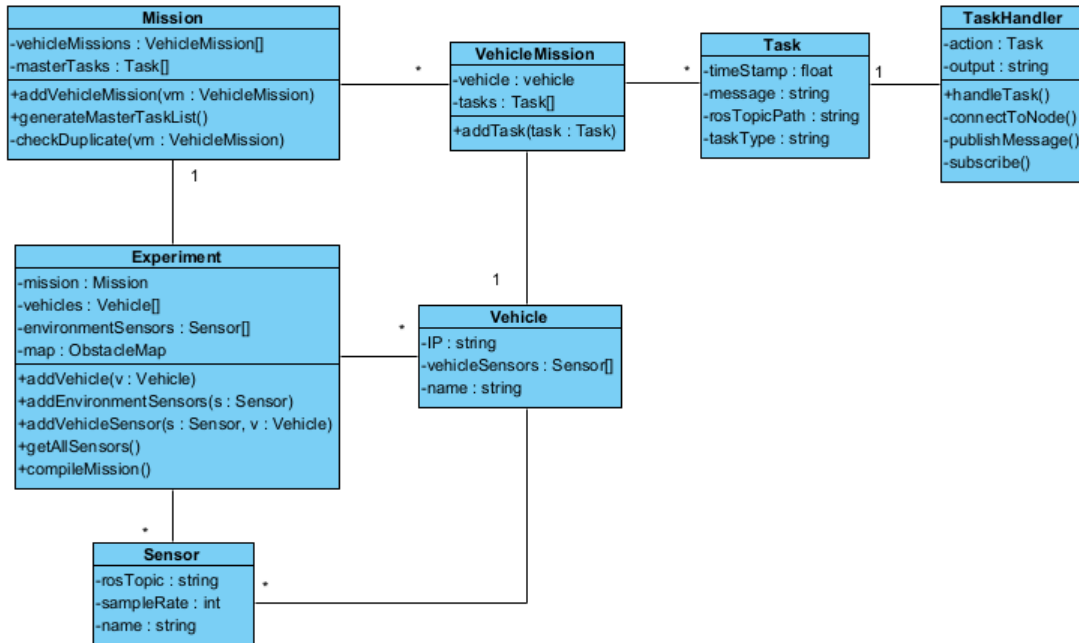


Figure 2.8: MESS Class Diagram

data in a central location. This data can be then post-processed in a program of the user’s choice.

Vehicle Command Vehicle command is the ability of the MESS to control the positions of the UAVs and UGVs. This is necessary for the MESS to be able to execute missions.

2.6.1.2 Mission Planning Implementation

The MESS is a desktop application programmed in Python. In order to achieve the needs outlined in the previous the section need to be satisfied within a code framework. Figure 2.8 shows the class diagram for the MESS that meets the design requirements.

Experiment The Experiment Class is the central class of the MESS software. It consists of a mission to perform, vehicles used in the experiment, and sensors used in the experiment that are not on a vehicle.

Vehicle The vehicle class corresponds to physical vehicles used in experiments. It holds each vehicle’s sensors, IP address—for ROS communication, and name used in the VICON environment.

Sensor The sensor class corresponds to sensors used in the experiment. It holds a string for each sensor that represents the path to its ROS topic. This allows the MESS to access the data published by the sensor.

Mission A mission is a collection of tasks to be performed within the experiment. It consists of multiple vehicle missions, which are aggregated to create a master collection of tasks.

Vehicle Mission A vehicle mission is a collection of tasks to be performed by a specific vehicle.

Task A task consists of an action and a time to perform that action. Actions are represented as a string that corresponds to a ROS message. The task also tracks where and when it needs to send the message.

Task Handler A task handler is the piece of code that executes the tasks. It connects to a ROS node to either publish or subscribe to a topic. If publishing to a topic

2.6.2 Software System Integration

In order to execute experiments, our software system integrates with many pre-existing external software programs through established communication protocols. Figure 2.9 shows a graphical representation of the different software components we use within our system.

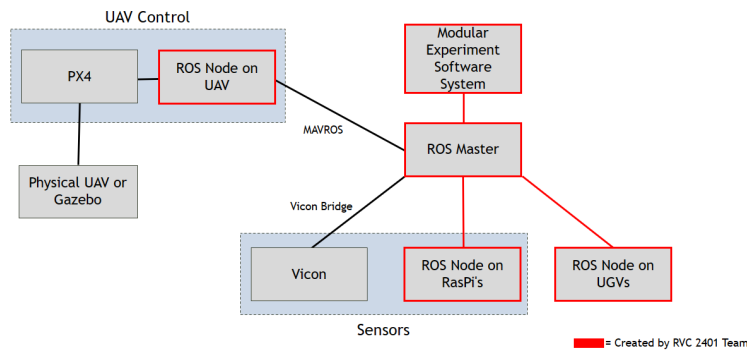


Figure 2.9: Software System Integration

PX4 PX4 is an open-source flight control software that operates on many different vehicle controllers. We initially opted to use ArduPilot due to its versatility of being supported on many different flight controllers as well as ample documentation when compared to PX4, but we only achieved stable flight when using the PX4 software.

Gazebo Gazebo is a simulation tool used to test the MESS in a virtual environment. It can simulate both ground and aerial vehicles and interfaces with ROS. When given commands to a ROS node, it is able to simulate the movement of a turtle bot. Since MESS interfaces with ROS nodes, a Gazebo simulation can provide proof that the MESS interface is successful in integrating the environments, independent of any hardware complications that may interfere.

MAVLink MAVLink is a communication protocol for sending data to UAVs and systems that control UAVs. It is designed to be lightweight and is widely used in industry.

VICON The VICON system is a motion capture system that utilizes multiple high speed cameras to provide positioning data for vehicles that operate in the lab environment. See Section 3.6.1.

VICON Tracker VICON Tracker is proprietary software released by Vicon Motion Systems that is used for tracking objects within the VICON environment.

ROS Master ROS Master is a centralized location that has access to all the ROS nodes and can publish and subscribe to every topic from a centralized location.

ROS Nodes ROS Nodes are instances of the ROS environment that can publish and subscribe to different topics. Nodes on UGVs allow for commands to be sent to the vehicle controller. ROS Nodes also allow for sensor data to be transmitted from the Raspberry Pis to ROS Master.

2.7 Lab Setup

2.7.1 VICON Motion Capture System Setup

The available lab space is a 34 x 24 foot room with roughly 10 feet of usable air space. The room has inherent obstacles including five support pillars, which are blind spots for the VICON cameras. This helps simulate an outdoor environment where natural obstacles are present, such as trees or buildings. In addition to the pillars, parts of the room contain low hanging vents and pipes which also create a non-ideal environment. The lab has three solid walls and a fourth open “wall” where we define the edge of the usable floor space. Figure 2.10 provides a detailed top view of the lab space, where the dotted line represents the open wall.

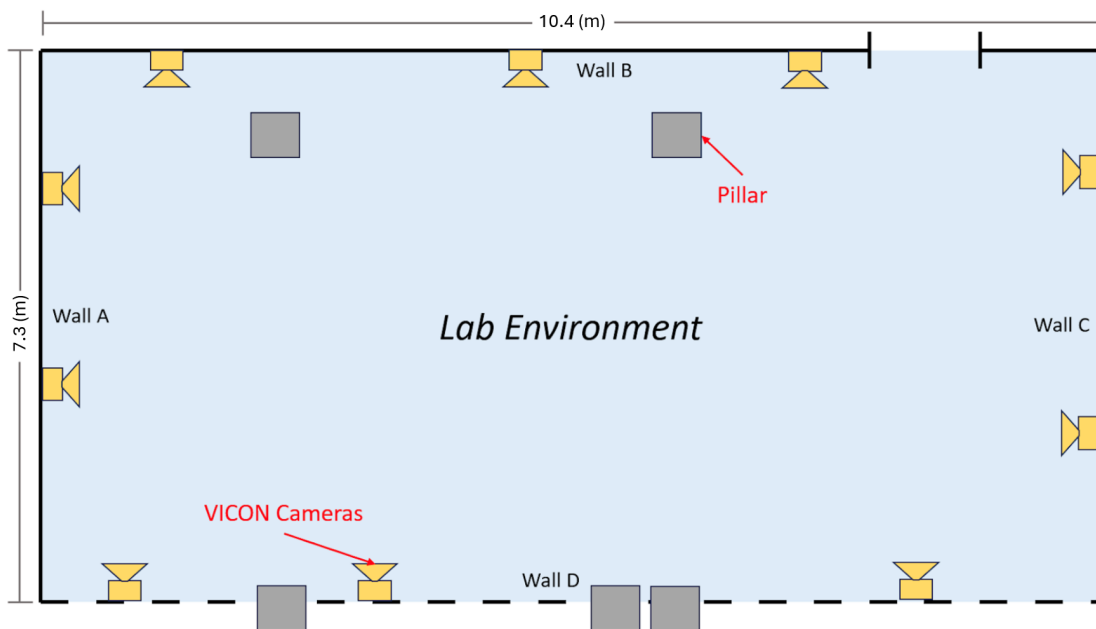


Figure 2.10: Lab environment setup

The final lab setup features a VICON motion capture system with mountings designed and constructed

by the team. It features ten wall mounted VICON Valkyrie motion tracking cameras evenly spaced around the room. We clamped each camera onto 4-foot long, 1-inch diameter aluminum pipe mounted through stainless steel wall/ceiling brackets. The connecting wires run through a wall mounted wire sleeve to the ground station switch box to avoid floor clutter. Excess wiring is coiled and hung beneath each camera on wall mounted j-hooks, which maintains open floor space.

The ground station consists of two computers, one exclusively for running the VICON Tracker software and the second for running all other software regarding communication with UAVs or UGVs.

2.7.2 Experimental Field Setup

The experimental field lies within the 5.8 x 4.3 m rectangular space on the floor and is free of obstacles. This is an adequate space to construct an experimental field. A key aspect of the field is that it contains four independent circuits that the UGVs traverse without collision (Figure 2.11). These roadways for UGVs are linear with perpendicular intersections, apart from a two-lane roadway which is at a 45 degree angle. One- and two-lane roads are approximately 0.6 and 0.9 m wide, respectively. A Turtlebot3 Waffle, with dimensions less than 0.3 m x 0.3 m, is thus able to traverse the roadways with adequate clearance.

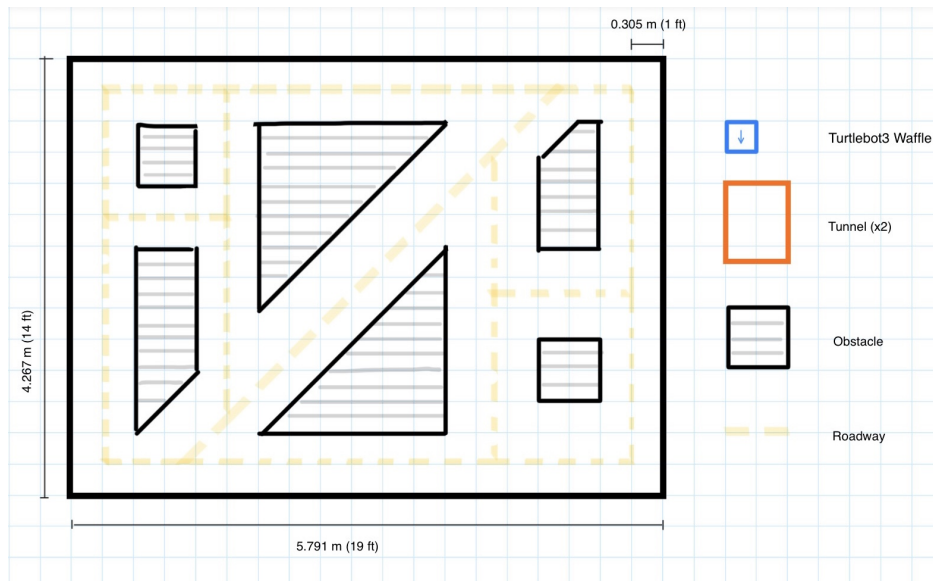


Figure 2.11: Experimental field setup floorplan.

Obstacles use small, black cardboard mailer boxes as corner posts. Duct tape wraps around the perimeter to form a barrier that is in line with the Turtlebot3's LiDAR sensing plane for navigation purposes (Figure 2.12). Additional mailer boxes are internally aligned with the duct tape perimeter to prevent sagging and deterioration of the obstacle formations. Taller obstacles are constructed from black foam core boards to provide visual interference with VICON motion capture system. One tunnel achieves almost complete visual

interference with the VICON cameras for testing purposes. This experimental field is satisfactory for the scope of this MQP, enabling a modular and configurable experimental setup. However, it cost approximately \$100 in materials and is subject to gradual deterioration.

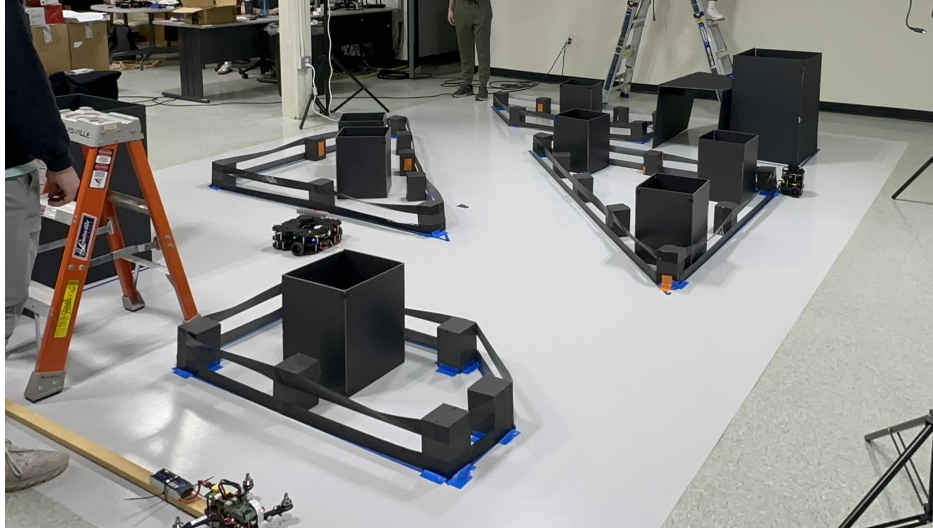


Figure 2.12: Constructed experimental field environment.

A more robust, modular approach could utilize wooden posts—such as a square end baluster—that are connected to a wooden base with wood glue. These posts can be cut to various sizes to adjust the maximum wall height. The base could be secured via heavy duty mounting tape to the lab floor. This setup would be more resistant to forces inducing tipping or rotation. These forces stem primarily from any tight perimeter wrapping around the wooden posts. Typical materials are duct tape or a nonadhesive tape-like material. This wrapping may contain more than three tiers, and may support vertical foam core panels which create high standing walls and barriers to the VICON field of view.

3 Design Process and Analysis

3.1 UAV Selection and Development

To start the MQP, we have three UAV options: use one created by the 2022 MQP, one created by the 2018 MQP, or create a new design. The 2018 UAV uses Pozyx as its positioning system, a Raspberry Pi, and a PixHawk Pixracer flight controller. The 2022 UAV attempted to integrate with the VICON motion capture system and uses an ODROID and a PixHawk 4 Mini flight controller. Given our objectives we will reuse the 2022 MQP's UAVs since it provides a convenient starting point and was already designed with the intention of integrating with the VICON system.

3.1.1 Troubleshooting

According to the report and videos from the 2022 MQP, the UAV was never capable of prolonged stable flight. The UAV was able to lift off and fly briefly, but quickly became unstable and needed to land before crashing. The first task is to determine the cause of any instabilities. We first consider the mass distribution, however, when we try to test fly the UAV, it is unable to lift off and nearly flips over. This leads us to believe that the issue is likely more fundamental than the mass distribution given that the flight controller should compensate for reasonable mass imbalances. Consequently, we investigate the hardware and software to pinpoint any issues. We then look into the power distribution board, analyze the ESC outputs, as well as the motor capabilities. Ultimately, we conclude that the entire setup is faulty. The batteries do not have their full capabilities, the flight controller is broken due to a crash from the previous users, and ESCs do not communicate well with the flight controller. We decide to replace these components, and flash a new firmware onto the flight controller. We start by using ArduPilot but switch to PX4. With these changes we achieve stable flight.

3.1.2 Battery

The 2022 MQP selected a 4500 mAh 3s (11.1 V) lithium polymer (LiPo) battery manufactured by HOOVO. They selected their battery based on 15 minutes of flight time. Following several failed flight tests and discovering the batteries have exceeded their life expectancy, we decide to order two new batteries with similar specifications but a slightly higher capacity. The batteries we choose are 5200 mAh lithium-ion polymer battery manufactured by Zeee. These are approximately the same size and shape as the HOOVO battery, meaning that they also fit well on the UAV frame.

When first charging the batteries, we improperly connect them to the charger, which we later realize

led to many of our thrust and motor issues. Improper charging led to unbalanced and depleted batteries, which cause a variety of power issues. This is reflected in the motor outputs, which are shown in Figure 3.1 and Figure 3.2. Figure 3.1 shows the pulse width modulation (PWM) signal of the motors, a quantity that is directly linked to the RPM of the motors and therefore the thrust of the motors. In the test, we evenly increase the throttle until maximum power. However, the PWM signals begin to diverge quickly. Only one motor outputs full power, while two output slightly lower values, and the last motor quickly drops to the minimum PWM value that corresponds to spinning.

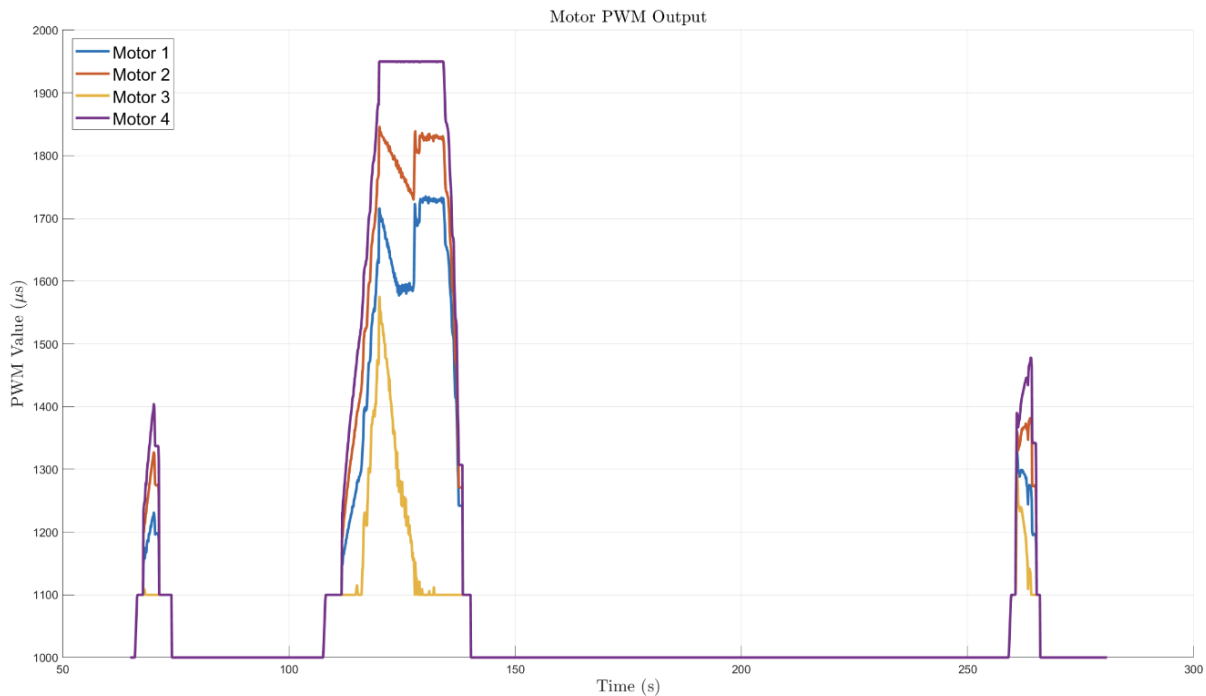


Figure 3.1: PWM Motor Outputs with Depleted Battery

Figure 3.2 shows substantially improved motor outputs while using the new and fully charged batteries. While the PWM outputs are not uniform, they are approximately the same value, can sustain maximum power (arbitrarily cut off at $1800 \mu s$ after watching the peak output of the motors through various trials), and the noise in PWM outputs is small enough to be easily processed by the flight controller.

While there are other hardware issues with the initial UAV besides improper battery charging, correctly charging and maintaining the LiPo batteries initially poses a major roadblock. All motors receive equal power with fully charged LiPo batteries, making the flight substantially more stable.

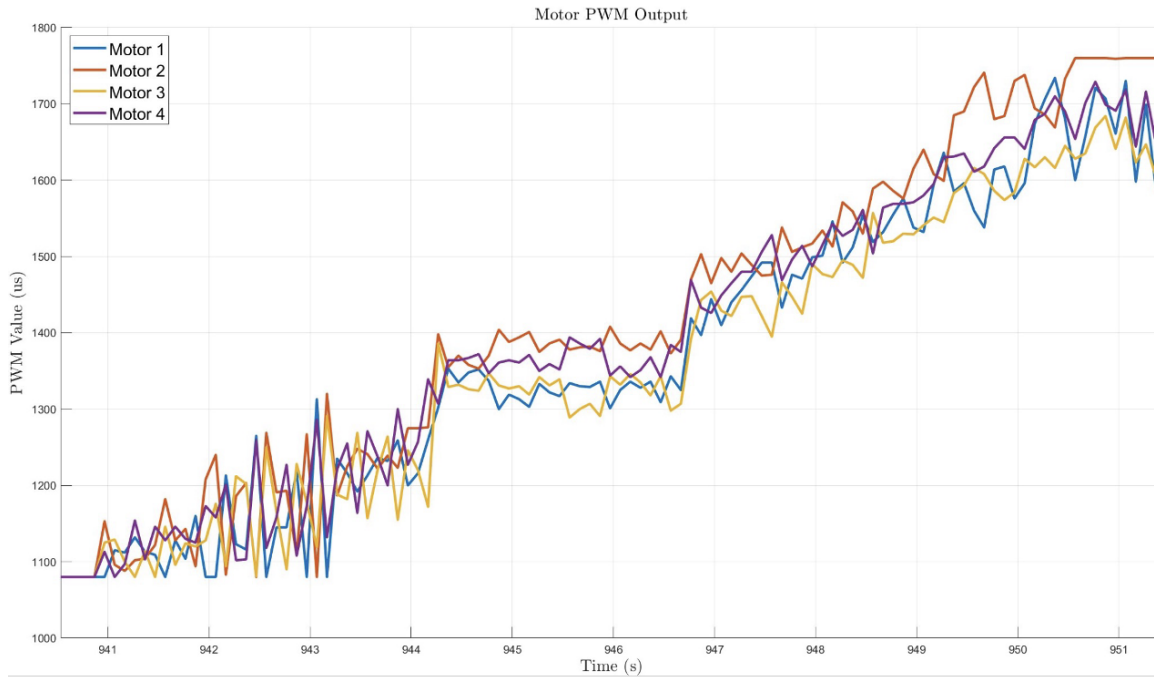


Figure 3.2: PWM Motor Outputs with Fully Charged Battery

3.1.2.1 Battery Charging

When charging the battery it is important to understand proper charging and discharging techniques. For the project we use 5200 mAh, 3 cell batteries. When charging, these batteries plug into the charge balance side—at the 3 cell port, and the power wire plugs into the paired cable for the positive and negative terminals. These cables are labeled in Figure 3.3 With the cables correctly attached, the battery should charge at a rate of $5200/1000$, or at a current of 5.2 A. When checking the voltage of each cell, it is important to note that full capacity is 4.2 V. When using the battery or discharging the battery the voltage per cell should remain above 3.8 V. When storing, the cells must be equally charged to 3.8V, as this is the most stable state. The battery should be left at storage voltage if it is idle for more than 12 hours. If not properly stored, the battery will fill with gas and swell, increasing the chance of fire and explosion.



Figure 3.3: UAV Battery, with Important Components Labeled

3.1.3 Power Module and Power Distribution Board

For the power distribution board, we use the Matek PDB-XT60, which is the same as the 2022 MQP's board. The board has a female XT60 connector to receive power from the power module. The function of the power distribution board is to evenly provide power to the motors. The distribution board is powered by the power module, which connects directly to the battery with a female XT60 connector. The power module powers the flight controller and regulates the current drawn from the battery. At first, we use the Pixhawk power module 5.3V BEC XT60; however, after switching to the ARKV6X flight controller, we use the ARK PAB Power Module to ensure compatibility.

While flight testing one UAV, we notice sometimes the motors shut off prematurely and none of the motors spin at full power. We find one cause of the problem is a faulty connection to the power distribution module. There are two red LED lights that turn on when the power distribution module has a good connection to the battery. Sometimes, these lights do not appear, and other times only one light illuminates. We determine the cause is a loose connection between the power cable and the board. After re-soldering this connection, the motors began spinning more evenly and no longer shut off during flight.

Since this significantly improved flight performance, we choose re-solder all the connections. In doing this, we learn the connections from the power board to the ESCs were faulty and some even burned out. Re-soldering all wires helped provide correct power to the different components, allowing the UAV to fly properly. For the second UAV, we successfully replicate the process from the first UAV.

3.1.4 Motors and Propellers

The motors on the 2022 UAV are T-Motor 2206 KV2000 brushless motors with 5-inch diameter, 3-inch pitch, 2-blade propellers. These motors are designed for multi-motor small vehicles. We first check each of the motors individually and find that one is broken. It does not spin under full power and the only motion is a slight twitching of the propellers. We switch the motor with a working one to see if it is an issue with the motor or the ESC. After switching the motors, we determine that the motor is the issue.

Before ordering a replacement motor, we need to verify the size of the motors using a thrust stand. We connect one of our motors to the Tyto Robotics 1520 thrust stand, using the provided battery and ESC. We use these instead of our own ESC to minimize error. We test our motor at low, medium, and high inputs, e.g. 50%, 75%, and 100% thrust.

According to the manufacturer's specifications, a singular motor should be able to lift approximately 195g at half power and 340 g at full power using 5-inch diameter propellers with a 3-inch pitch. This configuration matches the configuration of our motors. We test the motor using cutoffs to ensure that we do not damage the motor, especially since we are working with an unfamiliar battery and ESC. At low power,

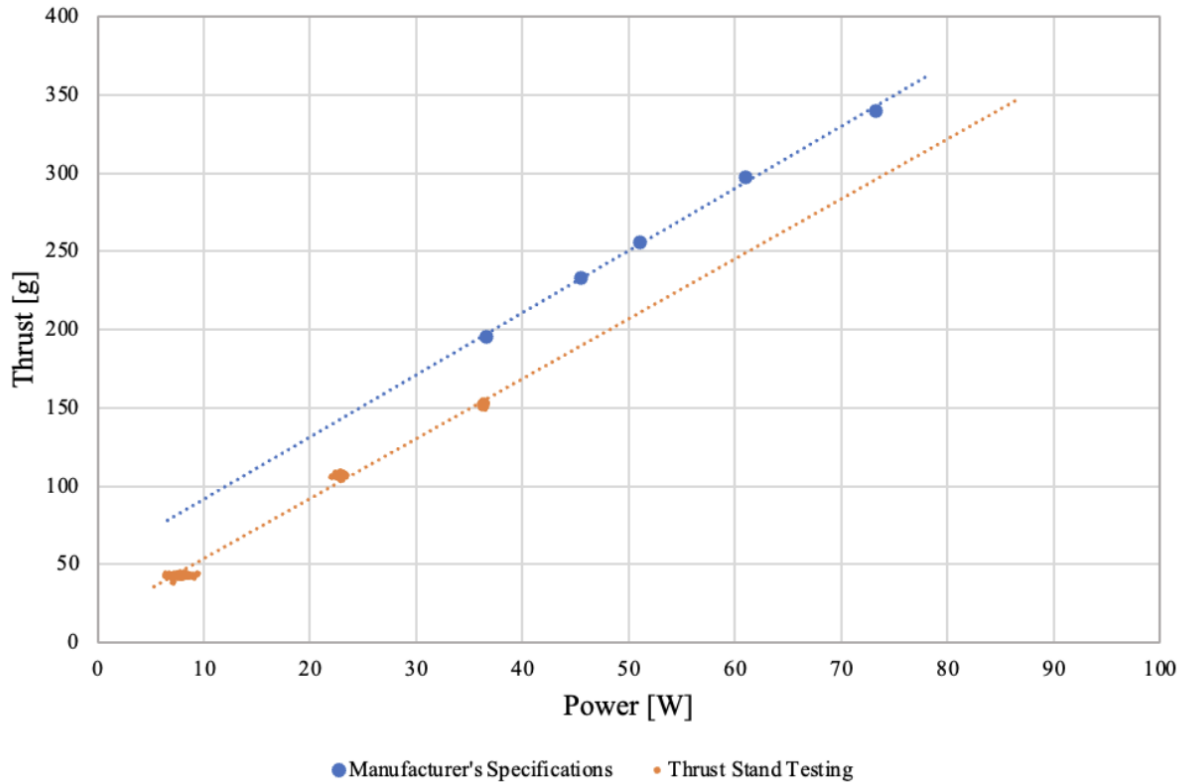


Figure 3.4: Thrust (g) vs. Power (W) plot for a single motor

approximately 7 W, we find that the motor generates 42 g of thrust. At high power, approximately 35 W, the motor generates approximately 140 g of thrust. Figure 3.4 shows a visual of our generated thrust versus the manufacturer's specifications.

While our generated thrust values are lower than those of the manufacturer, even at similar power, this is likely due to an inherent error in our setup. When we first use the thrust stand, we do not center the motor on the back-plate. This leads to exceedingly low thrust values. When moving the motor toward the center, we find higher thrust values for the same power inputs. Since we visually center the motor using one screw, this measurement may still be slightly off-center. Other sources of error may be the motor or configuration of the thrust stand. If we are not running the motor at exactly ideal conditions, we may be generating a lower thrust due to imperfections in provided power, wear on the motor, or different environmental conditions. Since 140 g of thrust at half power is sufficient to lift the quadrotor, we were not concerned with the inconsistency.

3.1.5 Electronic Speed Controllers (ESCs)

The ESCs chosen by the 2022 UAV did not output useful data. The original flight controller, the PixHawk, can communicate with the ESCs; however, they do not log data or output the correct power. We perform a series of voltage tests to ensure that the power distribution board and ESCs receive the correct

voltage. The power distribution board and two of the ESCs output the correct voltage, but two of the ESCs do not. We test this by spinning the motors up to full throttle. At this point, the ESCs should output 11.1V to the motors, but the faulty ESCs do not.

Instead of continuing to troubleshoot these ESCs, we order new ones from a different manufacturer, but ensure they have the same voltage rating and software. The new model is the Hobbypower Brushless 20A BLheli-S ESC Oneshot125. These ESCs have the BL-Heli-S software so that they log PWM data. When testing the new ESCs, we are able to track the PWM signal of the motors, which allows us to diagnose problems with individual motors quickly.

When setting up the new ESCs we must solder three wires to each one. Because we are already taking time to solder wires to the ESCs, we decide to also purchase bullet plugs which we can solder to the other end of the wires. This makes motor configuration far easier than the previous UAV, where the motors were soldered directly to the ESCs. Having motors directly soldered makes it difficult to change out motors because we would have to cut the wires and re-solder each connection again. Using bullet plugs makes changing motors as simple as unplugging and plugging back in.

3.1.6 Frame

The frame of the 2022 UAV is a 3D-printed PLA frame with 3 mm thick arms. One of the notes on the previous MQPs final design is that a PLA frame is insufficient for the UAV, namely its strength is not sufficient to prevent bending under motor thrust. We perform calculations and basic observations of the UAV to justify the cost of a carbon fiber frame.

First, we calculate how much the frame would deform under various loads. We approximate the problem using beam bending equations for a point load applied near the end of a beam:

$$\delta = \frac{WL^3}{3EI_z} \quad (3.1)$$

The arms vary in width, so we calculate the deflection using the smallest width (17 mm) to determine an estimate for deflection. The force applied by the motor occurs at 7.5 mm away from where the arm connects to the body. The deflection at this location is the deflection we care about, as it represents the change in force direction as the power to the motor changes. The modulus of elasticity varies based on the material, so we referenced common values for PLA and carbon fiber.

PLA has an elastic modulus of approximately 3500 MPa. From these calculations, we know that the PLA frame will deflect about 2 mm at half power (195 g of thrust) and 3.5 mm at full power (340 g of thrust). A carbon fiber frame with an elastic modulus of 228 GPa of similar dimensions will only deflect a

maximum of 0.03 mm and 0.07 mm under the same loading conditions. These results are summarized in Table 3.1.

Table 3.1: Deflection table for PLA and carbon fiber UAV frames

Material	Elastic Modulus	Deformation at 195 g [mm]	Deformation at 340 g [mm]
PLA	3500 MPa	2 mm	3.5 mm
Carbon Fiber	228 GPa	0.03 mm	0.05 mm

We verify the deflection of the arms by taking a video of an arm with the motor at full power. The arm visibly deflects upward when the motor is at half power, and deflects a larger amount when the motor is at full power. The video is taken with an iPhone camera propped stationary on a table to make sure the camera angle does not shift. We hold the UAV in place while spinning only one of the motors. By taking a screenshot of the video at an instance when the motor is at full power and comparing that to an instance when the motors are not receiving power, we can approximate the magnitude of the deflection, as is shown in Figure 3.5.

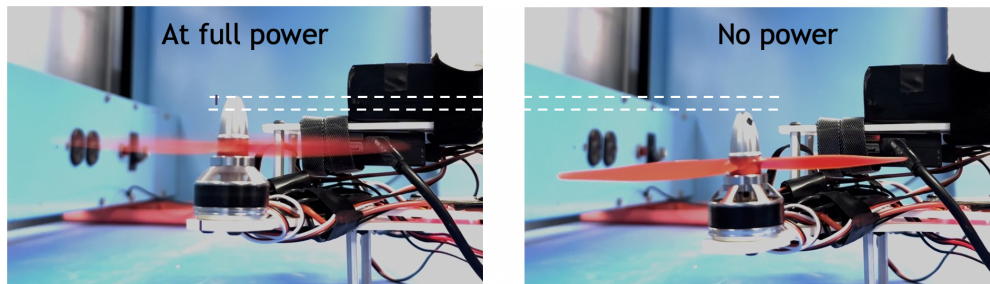


Figure 3.5: Deflection of the PLA frame under full power

Using the width of the frame (3 mm) as a reference, we determine the deflection of the frame (identified by the deflection of the top of the motor) to be approximately 3 mm, which matches our theoretical calculations. It also indicates that the PLA frame is not strong enough. Since the arms bend upward as the motors provide more thrust, this likely causes some interference with the flight controller, potentially a reason why the quadrotor can't attain stable flight. Since the flight controller relies on constant gains, it likely is not equipped to handle a continuously changing direction of normal force.

We order a carbon fiber frame based on Holybro's QAV250 quadcopter. The new carbon fiber frame provides stability and sturdiness to the UAV. We repeat the same test with the new frame and achieve much better results. Figure 3.6 shows the deflection of the carbon fiber frame with one motor at high power. The arm barely flexes upward. In Figure 3.6, the lower red line represents the height of the motor when the

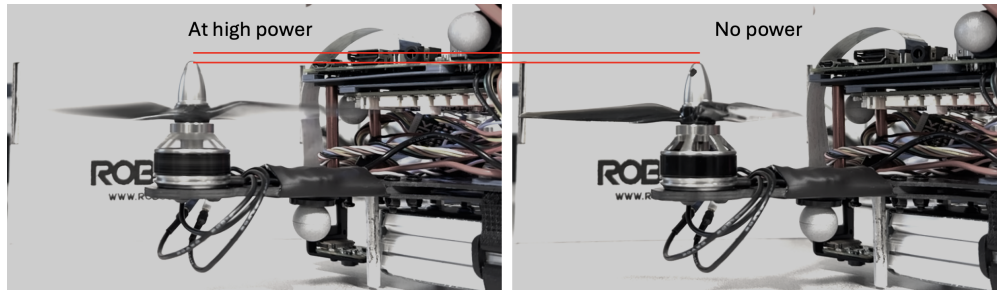


Figure 3.6: Deflection of the carbon fiber frame under high power

propellers are at rest. The higher line represents the height of the motor if the arm were to deflect by 3 mm (by once again using the width of the frame as a reference). When throttle the motor further, the UAV begins to physically lift off the table, so we elect to maintain the test at this value instead of at full throttle. Note that the increase in thrust of the motors with a properly charged battery and the increased rigidity of the carbon fiber frame leads the motor to exert much more upward force than in the first test.

3.1.7 3D Printed Mounts

Following the first assembly of the UAV on the new carbon fiber frame, we find that the frame heats up and smokes slightly when powering on the drone. Upon inspection, we realize that the power cable between the power module and the power distribution module has exposed solder on the underside of the boards. Consequently, the carbon fiber frame completes the circuit between the positive and negative terminals, causing a short circuit. None of the components fail, but the frame has a small mark burned into it. As a result, we decide to create small 3D-printed PLA mounts for these two components.

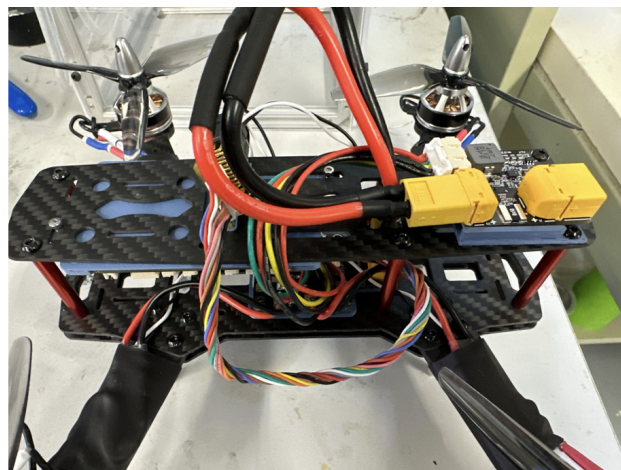


Figure 3.7: Image of the UAV without the Raspberry Pi

We also print mounts for the flight controller and the Raspberry Pi so we can attach those components nicely to the frame. Neither the controller nor the Raspberry Pi align with holes in the frame so the mounts

will provide a much more secure connection as opposed to the zip ties we had used. Figure 3.7 shows the assembled UAV with the new mounts (colored blue) for the power module and distribution module as well as the flight controller.

To create these mounts, we began by finding a similar 3D model from GrabCAD of the ZMR250 frame. We measure the spacing between holes on the top of the frame, and compare those dimensions to the frame we purchased and find that the CAD model is a good representation of our physical frame. We then model basic frames based on the size of the components and aligned screw holes to connect to the frame. When necessary, we use M2 thread inserts. Figure 3.8 shows the CAD models of the four mounts that we created. The mounts have elevated corners for the flight controller and Raspberry Pi to allow for the solder and ports underneath those boards. Similarly, the top of the mount for the power distribution module is slightly recessed to allow for the connection between the wire plug and the circuit board. The power module mount has an elevated left side to clear a screw on the right side, and a solid bar along the right edge to hold the power module in place.

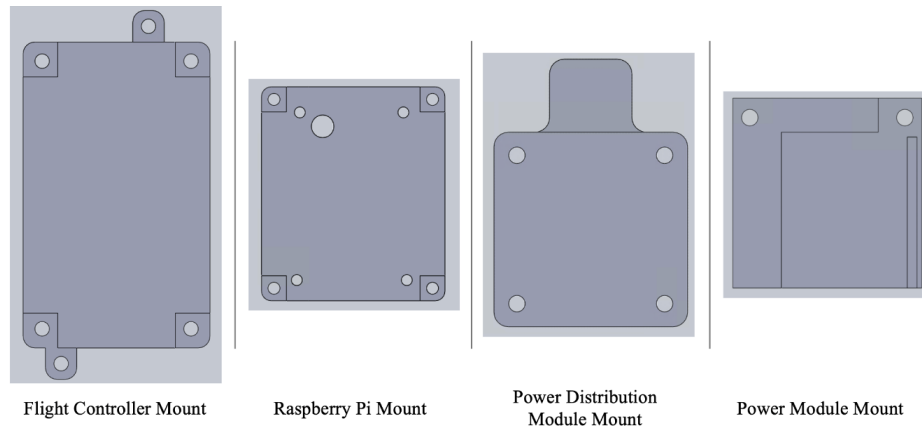


Figure 3.8: 3D Printed PLA Mounts for UAV Components

3.1.8 Flight Controller and Autopilot Software

To begin the project, we use the PixHawk 4 Mini from the 2022 MQP. We first flash ArduPilot software onto the flight controller. ArduPilot is an open-source flight controller software with constant community support and updates. ArduPilot also has documentation on using the VICON motion capture system as an indoor GPS. In the first flight test with the PixHawk, the UAV is unable to achieve a stable flight. Later, we will realize that the PixHawk’s internal sensors may be damaged. Inevitably, the Pixhawk proves to be problematic and we need to purchase a new flight controller. We select the ARKV6X as it is well-reviewed, and is an American company. Holybro, the main competitor we consider, ships from China and requires us to wait for customs to receive the new flight controller. Choosing the ARK makes shipping much faster and

still provides a high-quality controller.

We again flash ArduPilot onto the ARK board as we did for the old PixHawk. Once we properly configure the board and change basic parameters [15], we successfully connect to the motors and begin flight testing. Unfortunately, we notice that the UAV is still very unstable. Despite checking all the motors and re-calibrating the onboard sensors, the UAV quickly experiences oscillations and uneven flight after taking off. In an attempt to solve this problem, we flash PX4 software onto the flight controller. With the new software, we are quickly able to fly. Following online PX4 documentation [16], we configure the sensors, ESCs, and other relevant parameters and complete our first successful flight.

The main difference between ArduPilot and PX4 is the addition of a few key parameters. With PX4, we can set the ESC type and monitor the battery voltage. Consequently, we believe the PWM signal to the motors is more reliable and the flight controller has accurate information regarding the voltage to each motor at any time. Some additional parameters are unique to PX4, however, voltage monitoring should be available with ArduPilot. Since the board is designed off the PX4 standards, it may be that it is simply more compatible with PX4 than with ArduPilot.

3.1.9 Ground Station

We use QGroundControl as the command software to communicate with the UAVs. QGroundControl and MissionPlanner are the two main ground station software options. QGroundControl can command multiple UAVs, which we want to maintain as an option until we finalize communication protocols. Further, QGroundControl is compatible with MacOS, Windows, and Linux; while Mission Planner is only compatible with Windows and Linux. Aside from these differences, QGroundControl and Mission Planner provide approximately the same functionality, especially for basic use cases.

We use QGroundControl to calibrate our flight controllers, perform indoor flight testing, and troubleshoot any issues. QGroundControl is compatible with both ArduPilot and PX4, making our decision to switch autopilot software trivial. It allows the user to flash either software onto any known flight controller. Further, it walks the user through all basic calibrations and configurations; however, the user will most likely require documentation from the relevant autopilot software to fully configure the vehicle. While QGroundControl makes configuration easy, some steps are not apparent within the software. QGroundControl uses MAVLink communication protocols and allows the user to download .bin log data from the flight controller. These log files contain the vehicle's understanding of its location, relevant commands, and motor data.

3.1.10 Micro-Controller

After our first successful flight tests with manual control, we use a Raspberry Pi 3 (RPi) to remotely control the UAV. The Raspberry Pi runs ROS and mavros. Mavros is a robotics software specifically designed to communicate with UAVs. It relies on the MAVLink communication protocol, which both ArduPilot and PX4 use. By creating simple scripts in Python, we can send and receive commands and information. We first test our scripts in simulation. We perform testing using ArduPilot. Since the software is open source, we install it on a Linux virtual machine running Ubuntu 20.04 and also install ROS and mavros. We then create simple Python scripts that can command a virtual UAV launched through ArduPilot's Software in the Loop (SITL) functionality. Figure 3.9 shows the virtual UAV moving to a waypoint after taking off. The small pink arrow indicates the direction of motion.

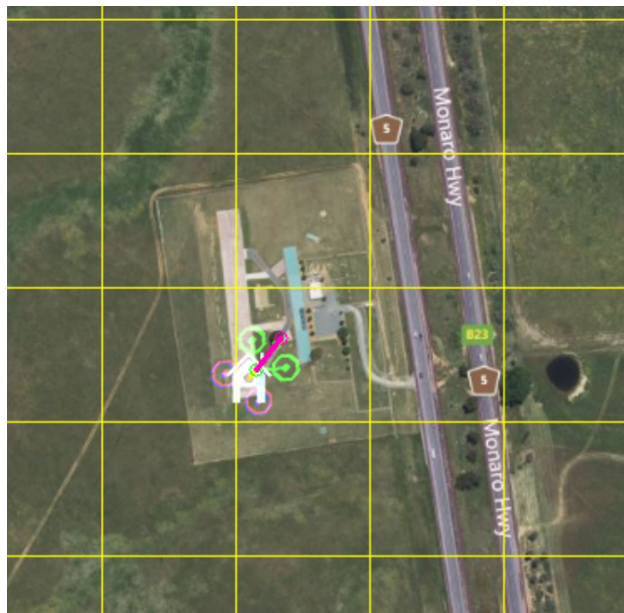


Figure 3.9: Simulated UAV Mission

To test our software, we connect the RPi to the flight controller using the Rx and Tx pins on each board (the Tx pins connect to Rx). We also connect 5V and ground to power the RPi. We first create a ROS launch file that would launch mavros and identify the port on the RPi that connects to the flight controller. Assuming that that connect uses pins 8 and 10 on the RPi, the port is “/dev/ttyAMA0” on the RPi 3, given that Bluetooth is disabled. Since we need a high baud rate to communicate with mavros, we disable Bluetooth to free the PL011 port, which allows greater processing power. If Bluetooth is not disabled, the appropriate port is “/dev/ttyS0,” which is the miniUART also used for serial port communications.

With the flight controller and RPi connected, we set the baud rate on both the flight controller and

in the ROS launch file. Ideally, the baud rate should be 921,600 B/s, which must be set for MAVLink communication on the flight controller, for the port on the flight controller, and in the ROS launch file on the RPi. If the wrong baud rate is set for any of these components, the flight controller and RPi will be unable to communicate. With these components connected, we monitor the status of the flight controller from the ground station via ROS. To execute commands, we either use the command line on the desktop or run a Python script.

3.1.11 Wi-Fi Board

To start we consider using ESP-32 Wi-Fi boards to communicate telemetry data. Wi-Fi telemetry allows for easier command of multiple vehicles when compared to radio telemetry. The ESP-32 Wi-Fi boards allow the UAV to be controlled from a computer connected to the same Wi-Fi network without needing a radio transmitter. After we switch to mavros, however, we simply use the Wi-Fi chip on the Raspberry Pi instead of the Wi-Fi boards.

3.1.12 Modularity

One of the primary goals of the team is to create an environment where every component is easily replaceable as a whole, as well as in parts. For the UAV, we want to be able to interchange the motors easily as well as change their directions. This is all done on the hardware side, so to achieve this, we soldered bullet plugs to the ESCs, and we ordered motors that already come with bullet plugs. On the 2022 UAV, as mentioned previously, they did not have the bullet plugs, and instead soldered those connections, making it a hassle to change out motors and even switch their directions. Another piece that we believe is important in making the UAVs modular, is to build them with the exact same components, and exact same layout. This is easier said than done because it is just as easy to want to reuse parts from other UAVs to minimize the cost.

3.1.13 Safety

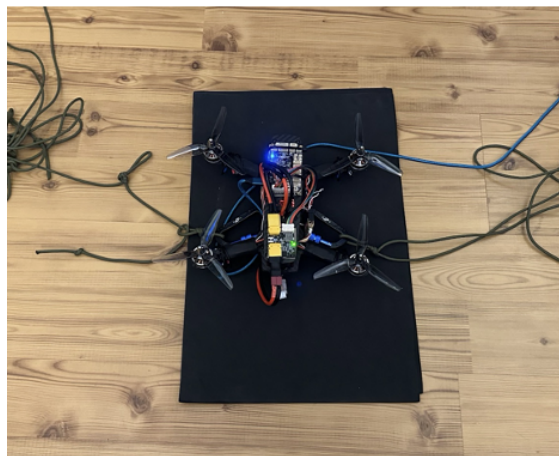


Figure 3.10: Tethered UAV for Flight Tests

Flying UAVs can be dangerous if not done properly. This is one thing we do not want to learn from experience, so we take some important precautions. On the electrical side of the UAVs, they carry 11.1 V throughout the whole structure. To minimize any chances of getting electrocuted, we utilize heat shrink to cover all electrical connections, solder joints, and ESCs. This leaves no exposed wires and little chance of any wires becoming exposed over time and is a significantly safer than electrical tape. In terms of the flight testing, we always make sure to tether the UAV before flying. This ensures the UAV can't fly erratically and injure teammates. The tether system we use can be seen in Figure 3.10.

3.1.14 Electrical Diagram

One goal, as mentioned previously, is to create an environment where every component is replaceable and where anybody can repeat our experiments. One aspect that is crucial to the experiment is the UAV. We face many hurdles throughout the project while building the UAV, so we made an electrical drawing to help future teams understand how the UAV is powered and where each component communicates to each other. Below in Figure 3.10, is the diagram displaying the power distribution across the UAV, and Figure 3.11 shows the input/output communication connections.

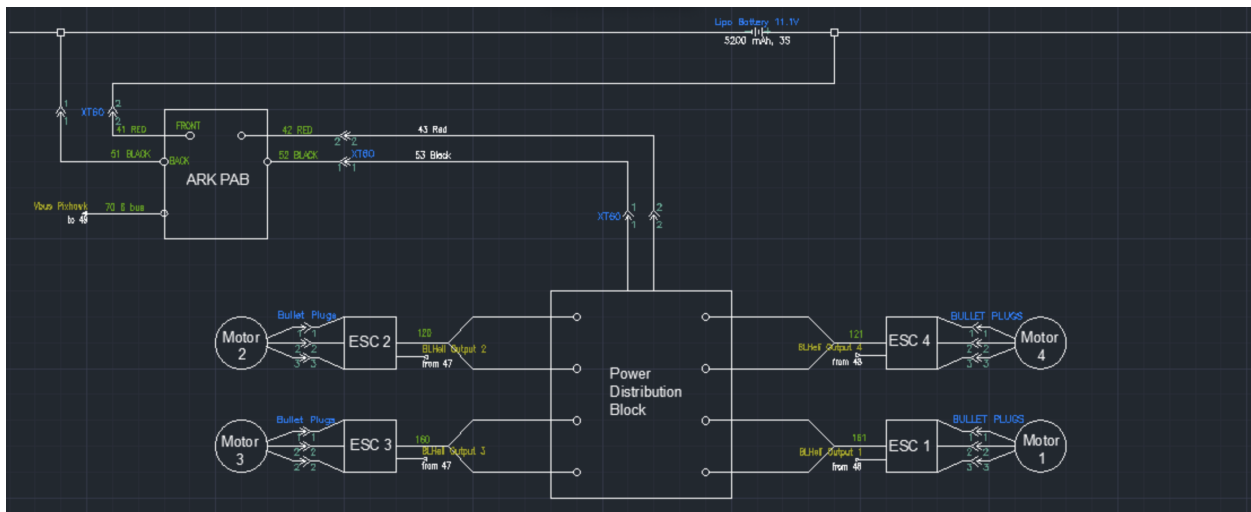


Figure 3.11: UAV Power Distribution

The power distribution diagram displayed above provides the connections from the 11.1V battery to the power module. The power module then powers the power distribution board and the flight controller (ARKV6X). The diagram also includes several soldered components. Every wire attached to the distribution board is soldered, as well as the end of the ESCs that connect to the motors. There is also an extension wire from the power module to the distribution board that is soldered and heat shrunk in the middle. This is displayed with the plug symbol. We have the option to use bullet plugs for this connection, however, it is not

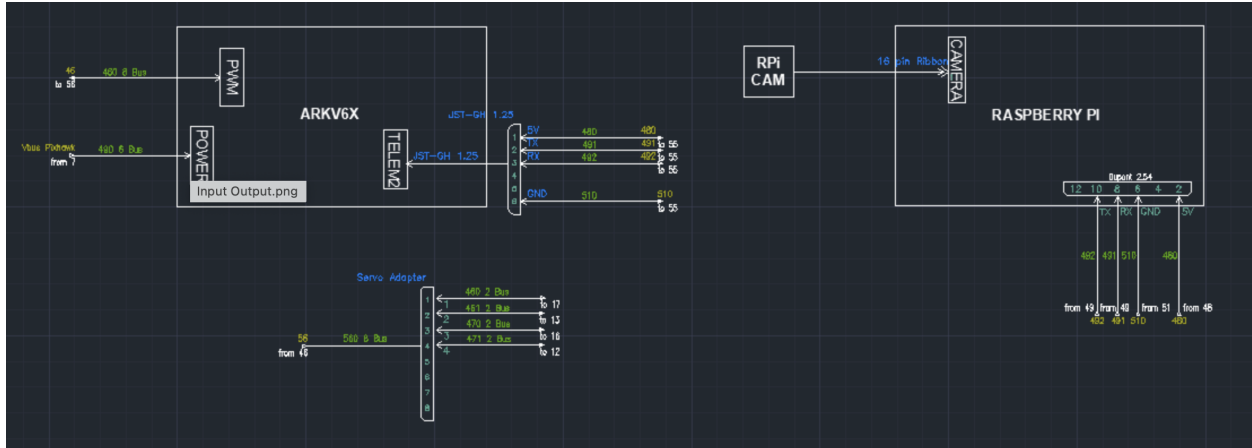


Figure 3.12: UAV I/O Connections

going to need to be unplugged ever so we decided it best to splice the two wires and solder them. All of the source wires in the power distribution diagram correlate to the respective wire numbers on the I/O diagram.

The input/output diagram above displays the connections that control the UAV's movement. These connections involve the flight controller, ESCs, and Raspberry Pi. These connections involve wires that plug into JST connectors, which then plug into the respected location which can be seen in the diagram. The wires that are connected to the connectors (ex. the vertical component on the bottom of the diagram) are custom made for the needs of the UAV. An important kit to purchase for this portion of the UAV is called a JST to Dupont kit. It provides the necessary wires and plug in connections to make the flight controller and Raspberry Pi compatible.

3.2 Wind Tunnel Testing

3.2.1 Mathematical Modeling

To determine the typical aerodynamic flight parameters for the UAV, we begin by selecting a mathematical model for our UAV. Selecting this allows us to determine which aerodynamic coefficients we need. We choose a simple model that captures the forces and moments acting on the body-fixed Cartesian coordinate system shown in Figure 3.13

Based on this coordinate system, a mathematical model for the UAVs moment M^b and force F_b dynamics can be written as follows in Equations (3.2) and (3.3), where F_i is the thrust due to motor i , M_i is the torque due to motor i and L is the distance from UAV center of gravity to each motor. Note that we assume L remains constant.

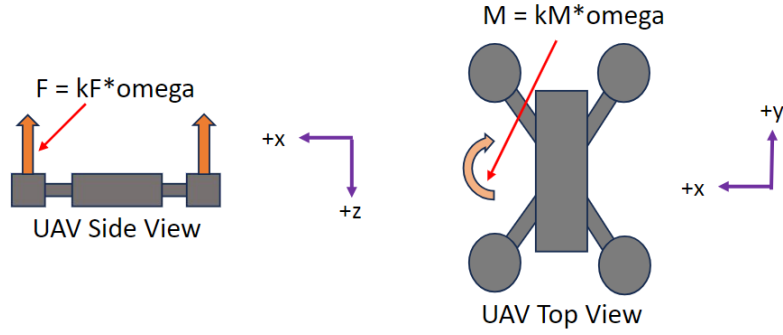


Figure 3.13: UAV body fixed Cartesian coordinate system

$$F_b = \begin{bmatrix} 0 \\ 0 \\ -(F_1 + F_2 + F_3 + F_4) \end{bmatrix} \quad (3.2)$$

$$M^b = \begin{bmatrix} -F_2L + F_4L \\ F_1L - F_3L \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} \quad (3.3)$$

To quantify the forces F_i and moments M_i , we determine the variation in these parameters with motor spin rate, the quantity we control via the flight controller. To simplify the model, we assume that the forces and moments generated by a given motor are quadratically related to its spin rate with a constant gain. The Equations (3.4) and (3.5) govern this relationship, where Ω_i is the motor spin rate in rpm and k_F and k_M are experimentally determined constants.

$$F_i = k_F \Omega_i^2 \quad (3.4)$$

$$M_i = k_M \Omega_i^2 \quad (3.5)$$

The forces and moments generated by each motor depend on many parameters—such as atmospheric density, UAV flight speed, and room temperature. To use the simplified model, we test our UAV at multiple flight conditions and measure the forces and moments acting on the vehicle to estimate k_F and k_M . We conduct a wind tunnel test using facilities on the WPI campus.

3.2.2 Wind Tunnel Setup

We use a Series 1585 Thrust Stand to collect required thrust and torque data. The Series 1585 was designed mainly to test UAV motor and propeller combinations; however, it is ideal for our purposes as it can measure both thrust and torque. To conduct the test, we bolt the UAV to the Series 1585—as seen in Figure 3.14—to reduce the risk of accidents occurring during testing. We also zip tie the UAV to the stand to reduce yawing motions.

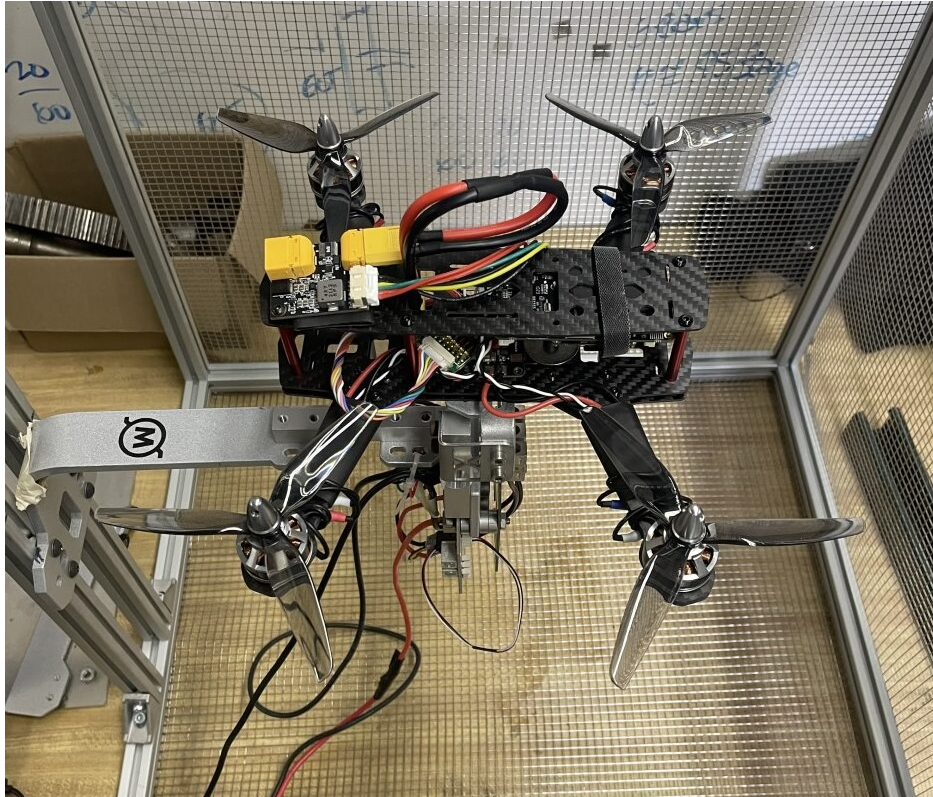


Figure 3.14: UAV mounted to Series 1585 Thrust Stand

The Series 1585 setup includes a steel wire mesh surrounding the thrust stand to catch any stray debris in the event of a failure. We remove this protective mesh to reduce turbulence and to ease the process of mounting the apparatus in the wind tunnel. Although we remove the protective layer, wind tunnel has clear polymer walls that protect bystanders from stray debris.

We mount the entire Series 1585 thrust stand in the wind tunnel using duct tape as seen in Figure 3.15. In this figure the airflow in the wind tunnel flows from right to left. We intentionally orient the thrust stand downstream the UAV to minimize any negative impacts of the structure. We feed control cables through the underside of the wind tunnel to collect data from both the thrust stand and UAV.

The follow steps outline the process to collect relevant data:

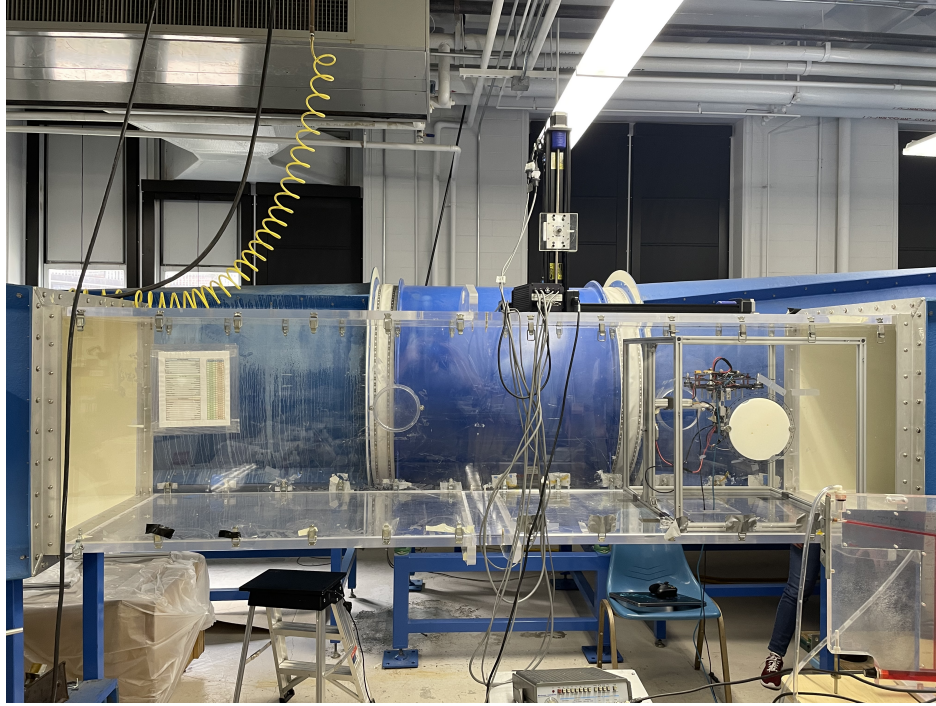


Figure 3.15: Side view of wind tunnel setup with thrust stand mounted

1. Tare the thrust stand data stream to zero thrust and torque sensor readings
2. Plug the UAV battery onto the UAV and spin up motors to initiate data logging.
3. After the motors are spinning as slowly as possible, spin up the wind tunnel.
4. Once the wind tunnel reaches steady state, ensure the thrust stand is streaming data.
5. Ramp the UAV motors up from the armed steady-state rpm to maximum rpm.
6. Collect sufficient data then decrease the rpm of the motors and turn off the wind tunnel.
7. Disconnect the UAV the battery
8. If desired, repeat the process at a different airspeed.

3.2.3 Data Collection

Before conducting the wind tunnel test, we decide which test conditions to run. Originally, we plan to conduct a series of tests at wind speeds ranging from 0 m/s to 2.5 m/s to cover the typical conditions of indoor flight. However, we did not realize this plan due to limitations of the wind tunnel itself. The facility at WPI cannot support wind speeds less than 5 m/s; thus our final test conditions only involve speeds greater than 5 m/s.

The final flight envelope consists of 9 wind speeds ranging from 5.0 m/s to 25.0 m/s at 2.5 m/s intervals. We also conduct a tenth test at 0 m/s to include hovering conditions. Since our UAVs will always fly at sea level atmospheric conditions, we will vary the flight speed and disregard air density variation. Therefore, these 10 wind speeds give the team sufficient data to characterize the relationship between motor rpm and UAV body forces and moments.

3.3 UGV Selection and Development

We consider two options for UGV hardware: an AWS DeepRacer and TurtleBot3s. The AWS DeepRacer is a four-wheel drive vehicle with forward facing cameras and is upgradable to include a LiDAR sensor. TurtleBot3s are differential drive vehicles equipped with an OpenCR1.0 controller, a Raspberry Pi 3B, and a LiDAR sensor. Both vehicle options allow for a versatile sensor suite and can operate using ROS, increasing the modularity of this project. We initially decide to investigate both options since either vehicle can use onboard path-planning to navigate between vertices.

3.3.1 AWS DeepRacer

We first evaluate the suitability of the AWS DeepRacer. Although the vehicle's primary use case is racing with reinforcement learning models, the hardware supports ROS functionality. The AWS DeepRacer that we borrow from the Novel Engineering of Swarm Technologies (NEST) Lab is an older model preinstalled with Ubuntu 16.04. We attempt to follow the NEST Lab's setup instructions for the vehicle using a desktop machine with Ubuntu 16.04 natively installed, but we are unable to access the vehicle console [17]. We consider flashing Ubuntu 20.04 onto the AWS DeepRacer and installing the ROS Noetic distribution that we use for this project; however, because of the setup issues, the need for a slower vehicle, and the lack of official ROS Noetic support for AWS DeepRacers, we decide not to use the AWS DeepRacer.

3.3.2 TurtleBot3 Burger and Waffle Pi

We switch our focus to using TurtleBot3s because they have well documented ROS support and differential drive vehicles have simple dynamical system models.

3.3.2.1 Vehicle Setup

We follow ROBOTIS's TurtleBot3 Quick Start Guide to set up the vehicle [14]. We modify the onboard .bashrc file to include two additional lines: "export TURTLEBOT3_MODEL=burger" and "export LDS_MODEL=LDS-01". The addition of these lines eliminates the need for a user to manually specify the model arguments. The model arguments are now automatically sourced, eliminating the need for a user to manually export these arguments whenever a new terminal is opened. The argument values "burger" and "LDS-01" are hardware-dependent. With the setup complete, we test the vehicle's functionality. We

successfully control the TurtleBot remotely using ROBOTIS's turtlebot3_teleop package and WASDX inputs on the keyboard connected to the ground station desktop.

3.3.2.2 System Model and Control

We initially use the onboard OpenCR1.0 odometry to test the navigation node while we configure the VICON environment. These tests include virtual Gazebo simulations and physical experiments using the hardware. To model the system, we define three coordinate frames (shown in Figure 3.16): a global frame from the VICON environment, a local frame used during translations between vertices, and a body-fixed frame used for the control of the TurtleBot3. During each transition, we redefine the origin and x-axis of the local frame as the starting point in the global frame and the unit vector from the starting point to the target vertex in the global frame, respectively.

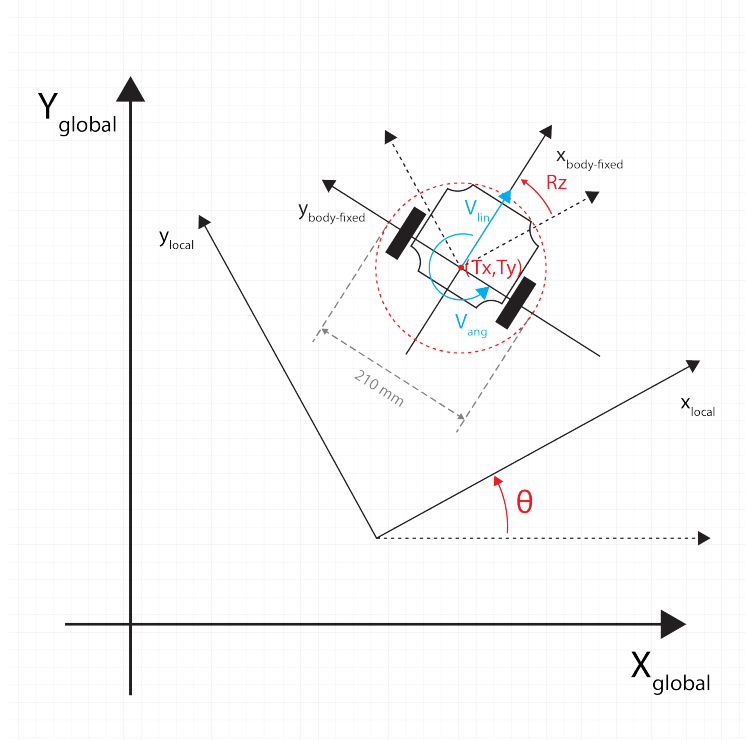


Figure 3.16: TurtleBot3 coordinate frames

In the global frame, we define the TurtleBot3 system model as:

$$x = \begin{bmatrix} T_x \\ T_y \\ R_z \end{bmatrix}, \quad \dot{x} = \begin{bmatrix} V_{lin} \cos(R_z) \\ V_{lin} \sin(R_z) \\ V_{ang} \end{bmatrix}, \quad u = \begin{bmatrix} V_{lin} \\ V_{ang} \end{bmatrix} \quad (3.6)$$

where T_x and T_y are the TurtleBot3's x and y-positions, R_z is the TurtleBot3's z-rotation, and V_{lin} and V_{ang} are the TurtleBot3's control input linear and angular velocities. We implement a time-optimal control framework to transition the TurtleBot3 between vertices. In this framework, we define vertices as unique states of the TurtleBot3 in the global frame. Transitions consist of either 1) a rotation, 2) a rotation and a translation, or 3) a rotation, a translation, and a rotation [18]. We implement this framework by defining a custom ROS message type and defining functions within our navigation node that handle pure rotation and pure translation. While the node is running, users publish a *MessToUGV* message containing a state and an operation index. The operation indices correspond to the three possible transitions. After receiving a *MessToUGV* message, the TurtleBot3 rotates and translates to the new state within user-defined tolerances. After completing each transition, the TurtleBot3 waits for a new state and operation index.

In initial tests, we experimentally determine that the TurtleBot3 rotates to within 0.01 radians of a target vertex. For translations that occur after rotations, the initial z-rotation error causes the TurtleBot3 to miss the target vertex. The TurtleBot3 also exhibits slight lateral motion during translations without an initial rotation, likely due to sensor drift. To remedy these sources of error, we define a line-following system model in the local frame.

$$x = \begin{bmatrix} e \\ \psi \end{bmatrix}, \quad u = [V_{ang}] \quad (3.7)$$

where e equals the local y-position error and ψ equals the local z-rotation error. We design a proportional-derivative (PD) controller for the angular velocity control input in the translation phase of the vertex transition. We assume a control input of the form:

$$V_{ang} = -eK_e - \dot{\psi}K_\psi \quad (3.8)$$

We create a simulation of the system in MATLAB and obtain gains using pidtool. A controller with gains $K_e = 9.5116$ and $K_\psi = 17.5623$ works in Gazebo simulations; however, these gains are too large for the real hardware. We experimentally decrease the gains to $K_e = 2.6779$ and $K_\psi = 7.6092$. We calculate the local y-position and z-rotation errors by projecting the vector from the initial vertex to the TurtleBot3's current position onto the vector from the initial vertex to the target vertex, both in the global frame. As these errors approach zero, the TurtleBot3 converges to the local x-axis and translates towards the target vertex.

3.3.2.3 Navigation in the VICON Environment

We modify the navigation node for use in the VICON environment after establishing communication between VICON Tracker and ROS [19]. We place five VICON pearl markers on the TurtleBot3 and create an

object for the vehicle in VICON Tracker. The orientation of this object in VICON Tracker is not aligned to the body-fixed frame, so we add a calibration to the navigation node where the TurtleBot3 translates for one second. We assume that any drift that occurs in this one second is negligible. The onboard computer stores the TurtleBot3's initial and final position during this translation, then updates four calibration parameters, C_1 and C_2 . We define C_1 as the initial z-rotation of the object in VICON Tracker and C_2 as the angle of the vector from the initial to final position in the global frame. C_3 and C_4 are the initial x-position and y-position in the global frame at bringup.

$$R_z^V = \text{atan2}(2Q_w^V Q_z^V + 2Q_x^V Q_y^V, (Q_w^V)^2 + (Q_x^V)^2 - (Q_y^V)^2 - (Q_z^V)^2) \quad (3.9)$$

$$\begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix} = \begin{bmatrix} R_{z1}^V \\ \text{atan2}(T_{y2}^V - T_{y1}^V, T_{x2}^V - T_{x1}^V) \\ T_{x1}^V \\ T_{y1}^V \end{bmatrix} \quad (3.10)$$

The onboard computer calibrates subsequent VICON callbacks by subtracting C_1 to zero the orientation and then adding C_2 to obtain the true orientation.

$$R_z^{GV} = R_z^V - C_1 + C_2, \quad -\pi \leq R_z^{GV} \leq \pi \quad (3.11)$$

Although the navigation node primarily utilizes VICON localization, there are cases where VICON localization may be unavailable. Users may want to intentionally simulate interference where VICON localization is either entirely unavailable or callback data is distorted. Additionally, VICON localization may be unavailable due to interference from obstacles and tunnels. During these periods, the TurtleBot3s rely on their onboard odometry to estimate their states. The onboard computer calibrates subsequent odometry callbacks by rotating the measurements by C_2 and then translating the position measurements by C_3 and C_4 .

$$T_x^{GO} = C_3 + T_x^O \cos(C_2) - T_y^O \sin(C_2) \quad (3.12)$$

$$T_y^{GO} = C_4 + T_x^O \sin(C_2) + T_y^O \cos(C_2) \quad (3.13)$$

$$R_z^{GO} = C_2 + R_z^O, \quad -\pi \leq R_z^{GO} \leq \pi \quad (3.14)$$

To ensure the odometry is correctly calibrated, the TurtleBot3 bringup must be fully launched before the navigation package, and the TurtleBot3 cannot move between bringup and the launching of the navigation

package. To account for further sensor drift, an additional calibration is applied to odometry callbacks. After each VICON callback, a function calculates the difference between the VICON state measurements and the odometry state measurements. When the onboard computer determines it should switch to the odometry state measurements, it begins applying a new transformation to correct the drifted odometry frame.

$$\begin{bmatrix} C_5 \\ C_6 \\ C_7 \end{bmatrix} = \begin{bmatrix} T_x^{GV} - T_x^{GO} \\ T_y^{GV} - T_y^{GO} \\ R_z^{GV} - R_z^{GO} \end{bmatrix} \quad (3.15)$$

$$T_x^{GO*} = C_5 + T_x^{GO} \cos(C_7) - T_y^{GO} \sin(C_7) \quad (3.16)$$

$$T_y^{GO*} = C_6 + T_x^{GO} \sin(C_7) + T_y^{GO} \cos(C_7) \quad (3.17)$$

$$R_z^{GO*} = C_7 + R_z^{GO}, \quad -\pi \leq R_z^{GO*} \leq \pi \quad (3.18)$$

3.4 Sensing Suite

Sensing suite design and development consider both lab- and vehicle-fixed, active and passive sensors. A selection of multi-modal sensors is vital to enable a variety of experimental applications for CSCP, particularly for unknown or dynamic lab environments. Through our design and development process, we achieve substantial modularity and configurability.

The sensing suite consists of three sensing types: visual light, infrared light, and radio frequency. We select these types to enable collection of a multi-modal sensed dataset while remaining within the scope of this project. Within each type, we research and evaluate sensing products according to design requirements outlined in Section 3.4.1.

3.4.1 Design Requirements

The design requirements guiding the selection of sensor products for each sensing mode are as follows: use case satisfaction, system integration compatibility, and ease of integration. Simply stated, does it address the needs of our experiment? Can it be integrated into our current work? And, how easy is it to integrate?

3.4.1.1 Use Case Satisfaction

Overall, the use case requirement evaluates whether the identified products are applicable to the current experimental goals while remaining useful in diverse experimental scenarios. Considering that the project will be leveraged for CSCP applications, this requirement will enable a user to specify desired sensor types and parameters and therefore gather the most relevant information. Accuracy, precision, cost, scalability,

and configurability of each sensor product is critical. Each chosen sensor must output reliable data, be cost effective within its category, and should allow for future expansion of experimental scenarios. We consider broad and configurable functionality to select sensors that can collect data in diverse experimental scenarios.

3.4.1.2 System Integration Compatibility

It is imperative that the sensor products we identify and evaluate are compatible with the MESS and ROS infrastructures. Specifically, for vehicle-fixed sensors, we research existing software libraries and application programming interfaces (APIs) that are compatible with the ROS 1 Noetic distribution environment. This eliminates the need for a custom software package to connect the sensor software to the MESS, which would divert software development focus from other aspects of the project.

3.4.1.3 Ease of Integration

It is possible that a compatible sensor product may require an elaborate hardware system or discontinued (no longer supported) software packages. We look for an active support and software development community for issue tracking, consulting, feature expansion, and optimization over time. Robust documentation which addresses the platforms and applications specific to our systems is sought as well. Other relevant characteristics are the sensor product's power requirements, interface versatility, and software maturity and interoperability.

3.4.2 Sensor Evaluation and Selection

Given the design requirements, we develop the following sensing suite.

3.4.2.1 Visual Light Cameras

We select the Raspberry Pi Camera Module V2 as a vehicle-fixed visual light camera. We affix these cameras to one UAV, which already host a Raspberry Pi. We select this camera due to its proven record of satisfying design requirements in past MQP work. Further, it is easy to integrate into the MESS. Ubiquity Robotics has a robust, open-source library for sending captured images and videos through publisher/subscriber communication channels in the ROS Noetic distribution.

3.4.2.2 Forward-looking Infrared Cameras

We select the Teledyne Forward-Looking Infrared (FLIR) A50/A70 series cameras as a lab-fixed infrared sensor. These cameras leverage a software development kit, Spinnaker SDK (SSDK), which serves as an API. Further, it is one of the only product lines with a robust and supported ROS Noetic code base with a package, `flir_camera_driver`, that bridges the ROS distribution with SSDK, which is compatible with our system. Note that integration requires creating a mounting system and an on-boarding period for lab

members to gain familiarity with its features and use.

However, given that this product is a significant investment, it does not become available to our team during the course of our MQP. We thus resort to the Raspberry Pi Camera Module NoIR V2. Similar to the Raspberry Pi visual light camera, we mount this camera on one of the UAVs. However, it lacks the broad and configurable functionality of the FLIR A50/70 series, making it a second choice.

3.4.2.3 Software-Defined Radio

Due to Federal Communications Commission (FCC) regulations, we are limited to broadcasting signals in ISM bands—unless a lab member holds a HAM Radio License. ISM bands are a portion of the radio frequency (RF) spectrum specifically reserved for industrial, scientific and medical (ISM) applications. Standard ISM frequency bands available in the U.S. are 902-928 MHz and 2.4 - 2.48 GHz. We recommend the latter frequency band range as it allows for higher data transfer rates. If considerably large distances exist between transmitters and receivers, then the former frequency band range would be more suitable, but this is not applicable in the lab space.

Given this, we select the Ettus Research USRP B200mini as a single software-defined radio (SDR) receiver. It serves as a full duplex (transmitter and receiver) with a 56 MHz bandwidth for high data transfer rates. With a frequency range of 70 MHz - 6 GHz, it follows the ISM band broadcast regulations as imposed by the FCC. We pair the B200mini with the Ettus Research VERT2450 Antenna. If set to transmit, the VERT2450 Antenna only broadcasts in the 2.4 - 2.5 and 4.9 - 5.9 GHz bands. Consequently, it enables ISM band transmission if the B200mini is the transmitter.

However, we did not select The B200mini as a transmitter. Instead, as we select two Analog Devices' ADALM-Pluto SDR as transmitters. A primary factor was cost with the B200mini and ADALM-Pluto costing approximately \$1,323 and \$291.36, respectively. The ADALM-Pluto satisfies our requirements, serving as a half or full duplex with a 20 MHz bandwidth for less but still adequate data rate transfer. Given the transmitters feasibly only need to broadcast frequency tones 1 MHz apart for unique detection, the product specifications are acceptable. Its frequency range also lies within the ISM band at 325 MHz - 3.8 GHz.

While the B200mini and ADALM-Pluto do not contain any integration support for our existing infrastructure, they can be interfaced with SDR software such as GNU Radio and MATLAB/Simulink. Therefore, they are potentially a standalone sensing system and would require a different automation process to integrate into the MESS. Given that this sensing system does not become available to our team during this MQP, we do not conduct testing or software exploration. It should be noted that WPI hosts a Wireless

Innovation Laboratory on campus that conducts extensive work with SDRs, a resource we use to orient ourselves to the SDR landscape.

3.5 Software

A substantial part of ensuring successful experiments is the software system. This encompasses both the creation of new custom software and substantial integration of pre-existing software frameworks. We detail the creation of an experiment management software and all necessary integration in this section.

3.5.1 Modular Experimental Software System (MESS)

We aim to create a custom software to assist in running experiments. The software should be one centralized application that can plan experiments, collect data, and command vehicles. To provide future versatility, the software should be modular to allow a user to easily configure and conduct experiments with different vehicles and sensors. Due to its modular nature, we name this application the Modular Experimental Software System (MESS).

3.5.1.1 Design Requirements

While designing the software, we design for three system needs: mission planning, centralized data collection, and vehicle command.

Mission Planning The first and primary goal of the MESS is to serve as a central place to plan and run experiments while minimizing the need for the user to write new code. To achieve this, certain inputs are required from the user. We decide user should provide information on the UAVs, UGVs, sensors, and mission plan. This is visualized in Figure 3.17.

A mission plan consists of tasks and times to perform those tasks. A task is an action to be performed by a sensor or a vehicle. Having a UAV takeoff, a UGV navigate to a way point, or a camera take a picture are all examples of a task.

Centralized Data Collection The MESS serves as a centralized location for data collection. Because experiments involve multiple vehicles and sensors, pulling and synchronizing data from each source separately would be time-intensive and cumbersome. The MESS solves this problem by accumulating data in a central location. This data can be then post-processed in a program of the user's choice.

Vehicle Command Vehicle command is the ability of the MESS to control the positions of the UAVs and UGVs. This is necessary for the MESS to execute missions.

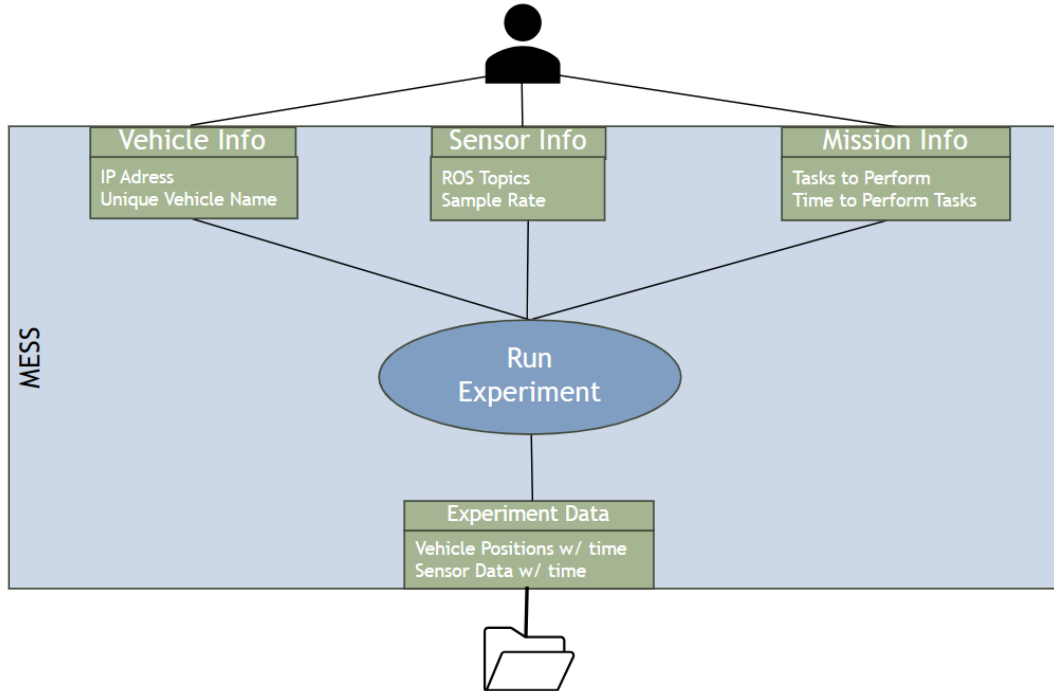


Figure 3.17: MESS Input and Output Diagram

3.5.1.2 Mission Planning Implementation

The MESS is a desktop application programmed in Python. To achieve the needs outlined in Section 3.5.1.1, we must use a code framework. Figure 3.18 shows the initial class diagram for the MESS that meets the design requirements, and 3.19 shows the final implemented class diagram.

Experiment The experiment class is the central class of the MESS software. It consists of a mission, relevant vehicles, stationary sensors, and an obstacle map to represent the environment.

Vehicle The vehicle class corresponds to the physical vehicles used in experiments. It holds all equipped sensors and the name used within the VICON environment VICON.

Sensor The sensor class holds all the sensors used in the experiment. It stores ROS topic to which each sensor publishes so that MESS can access this data.

Mission A mission is a collection of tasks to be performed within the experiment. It consists of multiple vehicle missions, which it aggregates to create the master collection of tasks to be performed.

Vehicle Mission A vehicle mission is a collection of tasks to be performed by a specific vehicle.

Task A task consists of an action and a time to perform that action. Actions are represented as a string that corresponds to a ROS message. The task also tracks where and when it needs to send the message.

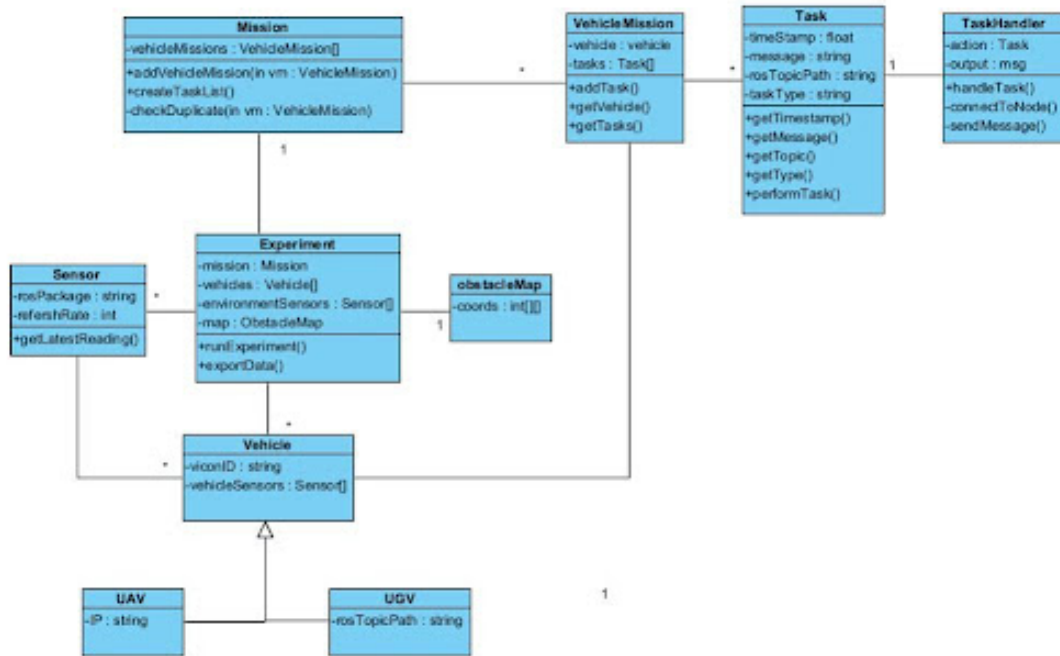


Figure 3.18: Initial MESS Class Diagram

Task Handler A task handler is the piece of code that executes the tasks. It connects to a ROS node to send a message to the relevant vehicle or sensor to have it perform that action.

Figure 3.19 shows the final implemented MESS class structure. The two changes are the removal of the UAV and UGV sub-classes and the removal of the Obstacle Map class.

We remove the UAV and UGV classes after we opt to use MAVROS for UAV communication. Since the UAVs integrate into the ROS network, it is irrelevant if a vehicle is a UAV or UGV as the communication protocol is the same for both vehicle types. This allowed the UAV and UGV classes to be combined into their parent class, Vehicle.

We remove the obstacle class map since we never implement it. As the project scope decreases, we abandon the idea of centralized tracking of multiple ground and aerial vehicles for obstacle avoidance.

The last key component of the MESS is that it can save and load experimental setups. To achieve this, the MESS saves configuration as a JSON file using Python’s json.dumps command from the built in JSON package. We create a custom JSON decoder to convert the JSON object into an experiment class object.

3.5.1.3 Centralized Data Collection

Since we have limited bandwidth in the lab, we opt to store sensor data onboard each individual vehicle during each experiment. At the conclusion of the experiment, we transfer the stored data into the MESS. In

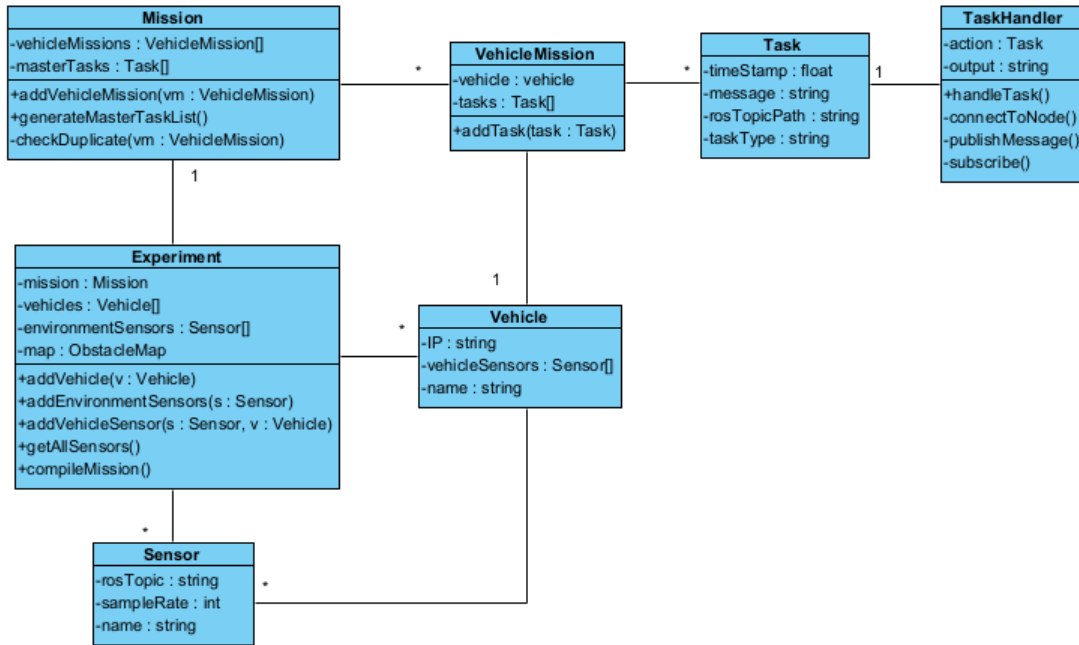


Figure 3.19: MESS Class Diagram

the event that data needs to be transferred live during the experiment, a Task in MESS could subscribe to the ROS topic to which the sensor publishes.

To collect the data, we equip each vehicle with a log directory where it can save data. This directory is identical across all vehicles, which is necessary for MESS to access the same folder regardless of the vehicle identification. MESS uses secure copy protocol (SCP) to copy the contents of the log folder from the vehicle to a location chosen by the user. Each experiment logs its contents in a single folder named by the experiment title and timestamp. Within this folder, each vehicle has a sub-folder. 3.20 shows an example of the structure of the collected data.

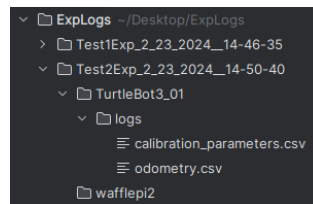


Figure 3.20: Collected Data File Format

Since each sensor is unique, the user must write a logger script to collect data for each sensor.

3.5.1.4 Vehicle Command Implementation

We design the MESS to use ROS for all vehicle communications. Due to difficulties obtaining stable flight and implementing mavros with the UAVs, MESS vehicle command has only been tested with UGVs.

Launch Files We use a ROS launch file to initialize all the topics for each node. Each vehicle hosts a ROS node and, as such, needs a launch file to initialize all relevant ROS topics. Furthermore, these topics must be unique across the experiment to allow for independent operations of UAVs and UGVs. For this reason each vehicle must have its own launch file, so it can create uniquely named topics.

To achieve unique naming, each UGV has a base set of topics that we rename. The launch file appends a prefix—based on the name of the vehicle—to each topic. The MESS creates this launch file and automatically adds the vehicle prefix.

The launch file must run on each vehicle before an experiment. To do this, MESS utilizes SCP to place the launch file in the src (source) folder of the ROS workspace of each vehicle. Then MESS uses a secure shell (SSH) connection to execute the launch command on each vehicle.

Waypoint Navigation MESS uses ROS messages to publish waypoint coordinates to each vehicle. Since future users are likely interested in path planning algorithms, the user can choose how the vehicle should move to each waypoint by modified that vehicle’s code. Section 3.3 describes how we implement path planning on the UGVs.

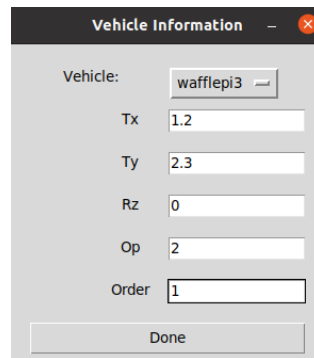


Figure 3.21: Way Point Entry in MESS

Figure 3.21 shows the input screen MESS utilizes to collect waypoint information from the user. Tx is the desired x-coordinate in the VICON frame, Ty is the y-coordinate, Rz is a value in radians for in-place rotations, Op is a value corresponding with travel mode (0 for rotation only, 1 for translation only, and 2 to translate and then rotate), and order dictates the sequence of waypoints.

3.5.1.5 MESS Development

For MESS to gain access to the rospy library, we develop it in a Python virtual environment launched from a ROS catkin workspace. We use the PyCharm IDE. The source code for MESS and a tutorial for setting up the development environment can be found at https://github.com/TommyLamar/MESS_PY.

3.5.1.6 MESS Example Demonstration

The following shows how to utilize MESS to command a UGV to navigate in a triangular loop. Figure 3.22 shows the launch window of MESS. Here a user can create new experiment or load a previous experiment setup.

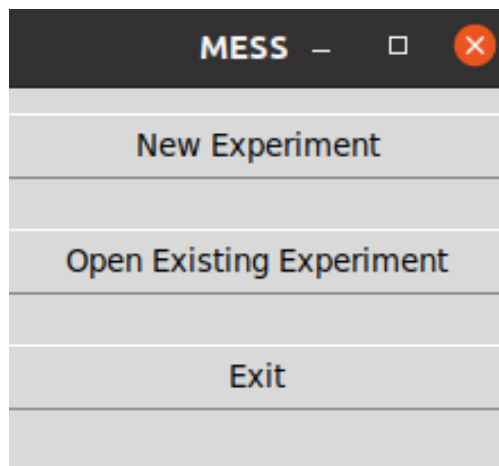


Figure 3.22: MESS Launch Window

Once an experiment has been created or loaded, the user is greeted with the experiment window, as seen in Figure 3.23.

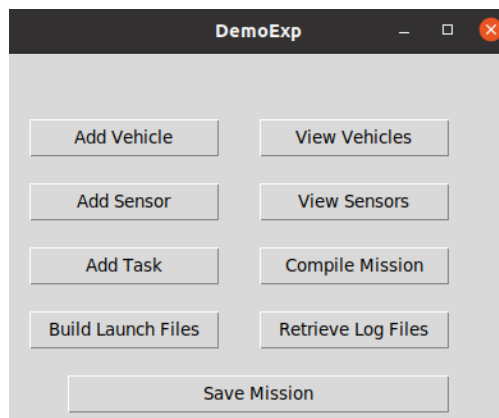


Figure 3.23: MESS Experiment Window

The "Add Vehicle" button brings the user to the screen seen in Figure 3.24, where they enter the vehicle

name and IP.

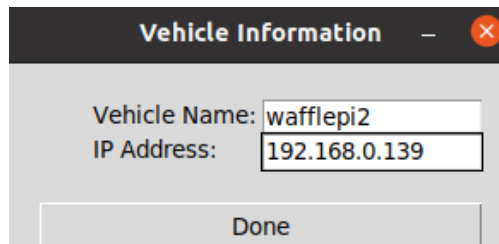


Figure 3.24: MESS Vehicle Info

After the vehicle is added, using the "Add Task," button and selecting "Way Point," the user is able to enter a way point command as seen in Figure 3.25. Tx and Ty are the coordinates in the VICON frame to translate to. Rz is the heading to rotate to. The op command can have the values 1, 2, or 3, with 1 being a rotation to heading Rz, 2 being a translation to position Tx and Ty, and 3 being a translation to Tx and Ty followed by a rotation to heading Rz. The order command is what order this way point should be executed. A way point command is needed for each translation.

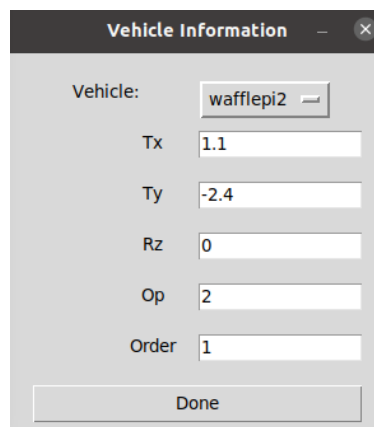


Figure 3.25: MESS Waypoint Navigation

Figure 3.26 shows the results of hitting the "Compile Mission" button after entering in all the way points.

Time	Vehicle	Topic	Message
1.0	wafflepi2		MessToUGV {Tx: 1.1 Ty: -2.4Rx: 0.00p: 2.0}
2.0	wafflepi2		MessToUGV {Tx: 0.5 Ty: 0.1Rx: 0.00p: 2.0}
3.0	wafflepi2		MessToUGV {Tx: 0.8 Ty: 1.2Rx: 0.00p: 2.0}
4.0	wafflepi2		MessToUGV {Tx: 1.1 Ty: -2.4Rx: 0.00p: 2.0}

Figure 3.26: MESS Mission Overview

3.5.2 Robot Operating System (ROS) Integration

We use ROS packages to enable communication between the ground station desktop and the TurtleBot3s, Raspberry Pi cameras, VICON Tracker, and other sensors. Nodes launched in the ROS environment transfer data by publishing and subscribing to specific topics. MESS publishes and subscribes to several topics in the ROS environment to coordinate experiments.

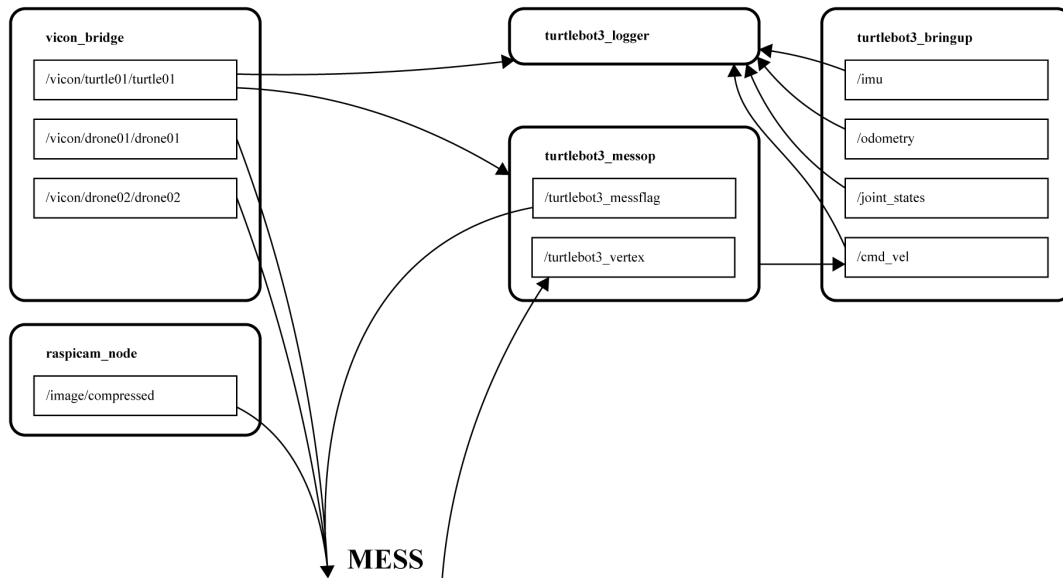


Figure 3.27: Interactions between ROS Nodes and MESS

3.5.2.1 ROS Installation

We flash Ubuntu 20.04 onto the ground station desktop. To install ROS, we follow the documented instructions for an Ubuntu install of ROS Noetic. After the install is complete, we create a catkin workspace to house all ROS packages used in this project.

3.5.2.2 raspicam_node

We use Ubiquity Robotics’s raspicam_node package to receive visual-light images from a Raspberry Pi Camera Module 2 in the ROS environment [20]. Although there is ROS Noetic support for raspicam_node, we follow our own procedure to add missing files before the node can be launched (see Appendix A).

3.5.2.3 turtlebot3_bringup

The turtlebot3_bringup package by ROBOTIS initiates a subscriber for the linear and angular velocities control inputs and publishers for the onboard IMU data, the OpenCR1.0 odometry, and the joint states of each wheel [21]. Control inputs received by the TurtleBot3 are transformed to pulse-width modulation

(PWM) signals that control the motors, enabling motion. To launch the node, we establish a Secure Shell (SSH) connection with the TurtleBot3 from the ground station desktop and execute “roslaunch turtlebot3_bringup turtlebot3_robot.launch” in the terminal.

3.5.2.4 turtlebot3_logger

We create a package that logs all data onboard the TurtleBot3 to .csv files. We install the package by executing “git clone https://github.com/marinarasauced/turtlebot3_logger.git” in a terminal on the TurtleBot3 to download the package to the “src” directory in the catkin workspace. We establish a connection with the TurtleBot3 using SSH protocol and execute “roslaunch turtlebot3_logger logger” in the terminal. We shut the node down by publishing a Bool message type to the “/turtlebot3/logger” topic. After the message is received, five log files are created in the current directory of the terminal. We download the directory containing the log files from the TurtleBot3 to the ground station desktop using Secure Copy Protocol (SCP) and analyze the log files using MATLAB.

3.5.2.5 turtlebot3_messop

We develop a package that transitions the TurtleBot3 between two vertices. We install the package to the “src” directory in the catkin workspace by executing “git clone https://github.com/marinarasauced/turtlebot3_messop.git” in a terminal on the TurtleBot3. We launch the node by establishing an SSH connection with the TurtleBot3 and executing “roslaunch turtlebot3_messop messop” in the terminal. Once the node is launched, it performs a calibration and the TurtleBot3 waits for a new vertex to be published. The TurtleBot3 then performs one of three operations, specified by the user as part of the vertex input: 1. rotate towards a heading; 2. rotate towards a heading and then translate to a point; 3. rotate towards a heading, translate to a point, and rotate towards another heading. Although we only utilize the second of these three operations in this project, the package provides future users with a high degree of flexibility.

3.5.2.6 vicon_bridge

Using the vicon_bridge package by ETH Zürich’s Autonomous Systems Lab, we receive data from VICON Tracker in the ROS environment [19]. Position and orientation data of all tracked objects are passed from VICON Tracker through the Datastream SDK to the vicon_bridge node using C++ and a local area network (LAN) connection between the two desktops. Before launching the node for the first time, we configure the launch file in the vicon_bridge package to receive data from the VICON desktop’s static IP address. We launch the node by executing “roslaunch vicon_bridge vicon.launch” in a terminal on the ground station

desktop.

3.5.3 Software System Integration

To execute experiments, our software systems integrates with many pre-existing external software programs through established communication protocols. Over the course of the project, the integration changes and simplifies. Figures 3.28, 3.29, and 3.30 show a graphical representation of the different software components we use within our system.

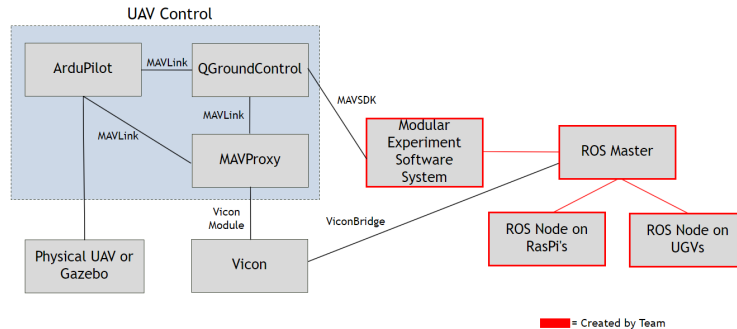


Figure 3.28: Software System Integrations

Figure 3.28 shows the first plan for software integration. We initially have two communication systems, one for the UAVs and a separate one for the ROS network (for the UGVs and sensors). In this case, the MESS would harmonize the two communication methods. Below we have a summary of the relevant software and connection protocols.

ArduPilot ArduPilot is an open source flight control software that operates on many flight controllers. We select ArduPilot due to its versatility and ample documentation.

QGroundControl QGroundControl (QGC) is a ground station software used for UAV command and control. It communicates with any flight controller running ArduPilot. QGC is equipped with several flight modes that handle tasks like taking off, landing, stabilizing flight, and waypoint navigation. It also allows for easy calibration of the flight controller. Due to its powerful built in functions and ease of use, we select it as the main ground station software for the UAVs.

Gazebo Gazebo is a simulation tool used to test the MESS in a virtual environment. It can simulate both ground and aerial vehicles and interfaces with ROS. When given commands to a ROS node, it is able to simulate the movement of a turtle bot. Since MESS interfaces with ROS nodes, a Gazebo simulation can provide proof that the MESS interface is successful in integrating the environments, independent of any hardware complications that may interfere.

MAVLink MAVLink is a communication protocol for sending data to UAVs and systems that control UAVs. It is designed to be lightweight and is widely used in industry.

MAVSDK MAVSDK is an application program interface (API) used to communicate QGC commands from a C++ or Python application. The MAVSDK API allows the MESS to send commands to QGC.

VICON The VICON system is a motion capture system that utilizes multiple high speed cameras to provide positioning data for vehicles that operate in the lab environment. See Section 3.6.1.

VICON Bridge Vicon Bridge is a ROS Package that allows for the easy transmission of object data from VICON Tracker to a ROS node. More information can be found in Section 3.5.2.6.

VICON Module Vicon Module is an add on to MAVProxy that allows MAVProxy to receive data from the VICON System.

ROS Master ROS Master is a centralized location that has access to all the ROS nodes and can publish and subscribe to every topic from a centralized location.

ROS Nodes ROS Nodes are instances of the ROS environment that can publish and subscribe to different topics. Nodes on UGVs allow for commands to be sent to the vehicle controller. ROS Nodes also allow for sensor data to be transmitted from the Raspberry Pis to ROS Master.

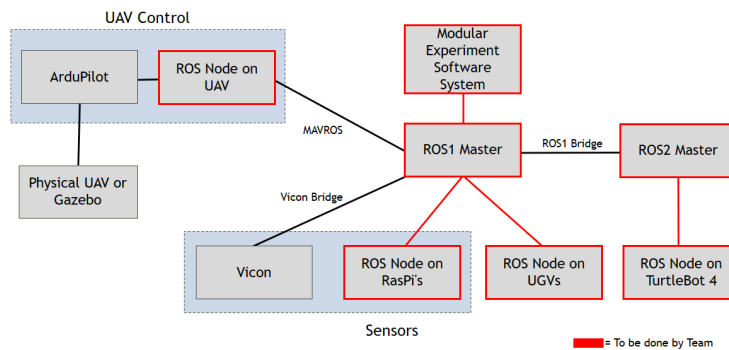


Figure 3.29: Software System Integrations

Figure 3.29 shows the necessary integration after we opt to use mavros. Mavros is a ROS package that interfaces with UAVs using the MAVLINK protocol. This eases the communication methods needed by MESS as all communications are now done through the ROS network.

The switch to mavros simplifies the integration by removing the need to use both MAVProxy and QGC. MAVROS enables MESS to communicate directly with the UAVs. Additionally, since we are not using QGC, we do not need MAVProxy for multi-vehicle flight.

At this point, we investigate the feasibility of using a TurtleBot4. The TurtleBot4 is designed for use

with ROS 2 with no easy way of running ROS 1 on the hardware. Since the TurtleBot3s and mavros both depend on ROS 1, we would need to use `ros1_bridge`, which translates ROS 1 messages to ROS 2 and vice versa. However, due to the necessity of having both ROS 1 and ROS 2 on the same computer, added complexity, and the larger size of the TurtleBot4, we choose not to use the TurtleBot4s.

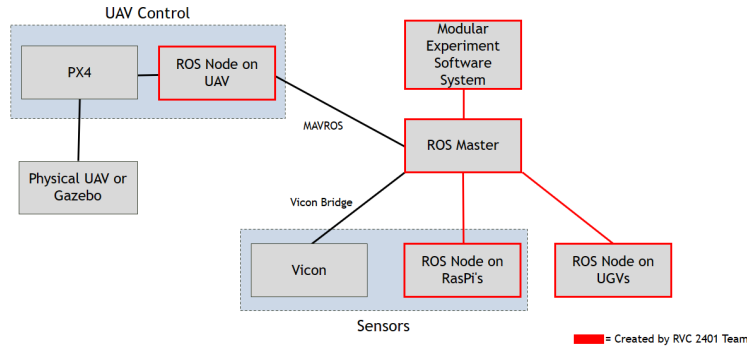


Figure 3.30: Software System Integrations

3.30 shows the final state of software integration. The two notable changes from the prior iteration are the removal of ROS 2 Master and the ROS 2 node of the TurtleBot4—we remove this functionality when we decide against the TurtleBot4—and the swap from ArduPilot to PX4.

Like ArduPilot, PX4 is an open-source flight control software that operates on many flight controllers. While we initially opt not to use PX4 due to its perceived smaller documentation and active community compared to ArduPilot, we obtain yields stable using PX4 and uncontrolled flight with ArduPilot. We are unsure why PX4 creates such a large change.

3.5.4 UAV Computer Vision

To estimate the threat of the laboratory environment, sensors must measure information about the environment. ROS compatible Raspberry Pi cameras mounted on the UAVs record RGB values either within the visual light or infrared frequencies. These images are mapped to the VICON environment using the camera optics and VICON localization of the respective UAV.

3.5.4.1 Assumptions

A global three-dimensional matrix of user-defined resolution tracks the state of each position-discretized vertex within the laboratory environment.

$$x = \left[T_{x,px}^G \quad T_{y,px}^G \quad R^G \quad G^G \quad B^G \quad N^G \right]^T \quad (3.19)$$

The pixel dimensions in each image must be less than the user-defined global matrix resolution. A preliminary analysis suggests that images taken from the UAV at a height of three meters is approximately one millimeter. In the laboratory environment, the UAVs must operate lower than three meters due to the physical constraints of the space. Given that the finest global matrix resolution is one centimeter, there should be at least one pixel per point during the image mapping process. Additionally, the threat plane is assumed relatively flat. Since RGB values are measured, threat values must be obtainable as a function of RGB values.

3.5.4.2 Projective Image Transformations in Homogeneous Coordinates

RGB values in sampled images are mapped to the threat plane by transforming a normalized matrix of position values to the true area that the image covers in the threat plane. The dimensions of the normalized mesh equal the dimensions of the sampled image. Source and destination corner vertices are used to calculate a homogeneous transformation matrix, which is then applied to all points in the normalized matrix.

3.5.4.3 Source Corner Vertices

The source corner vertices equal the positions of the corners of the normalized matrix. The index of each corner represents to the quadrant that corner occupies if the image is taken at the origin and surface normal to the threat plane.

$$\begin{bmatrix} S1 \\ S2 \\ S3 \\ S4 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -1 & 1 \\ -1 & -1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \hat{i} \\ \hat{j} \end{bmatrix} \quad (3.20)$$

3.5.4.4 Destination Corner Vertices

The destination corner vertices are calculated using UAV's pose and the field of view of the camera. The vector that is surface normal to the camera when the UAV has zero roll and pitch is a negative unit vector along the global z-axis.

$$\vec{v}_0 = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix}^T \quad (3.21)$$

A transformation calculates the position of the camera with respect to the measured pose of the UAV object in VICON Tracker. A translation is applied followed by an XYZ Euler rotation in the UAV body-fixed frame.

$$\begin{bmatrix} T_{x,cam}^G \\ T_{y,cam}^G \\ T_{z,cam}^G \end{bmatrix} = \begin{bmatrix} T_{x,uav}^G \\ T_{y,uav}^G \\ T_{z,uav}^G \end{bmatrix} + \begin{bmatrix} \cos(R_{z,uav}^G) & -\sin(R_{z,uav}^G) & 0 \\ \sin(R_{z,uav}^G) & \cos(R_{z,uav}^G) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(R_{y,uav}^G) & 0 & \sin(R_{y,uav}^G) \\ 0 & 1 & 0 \\ -\sin(R_{y,uav}^G) & 0 & \cos(R_{y,uav}^G) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(R_{x,uav}^G) & -\sin(R_{x,uav}^G) \\ 0 & \sin(R_{x,uav}^G) & \cos(R_{x,uav}^G) \end{bmatrix} \begin{bmatrix} d_{cam} \\ 0 \\ h_{cam} \end{bmatrix} \quad (3.22)$$

The vectors from the camera to the corners of each image depend on the roll and pitch of the UAV and the field of view of the camera.

$$\vec{v}_1 = \left[\tan(R_{x,cam}^G + FOV_{wide}) \quad \tan(R_{y,cam}^G + FOV_{high}) \quad -1 \right]^T \quad (3.23)$$

$$\vec{v}_2 = \left[\tan(R_{x,cam}^G - FOV_{wide}) \quad \tan(R_{y,cam}^G + FOV_{high}) \quad -1 \right]^T \quad (3.24)$$

$$\vec{v}_3 = \left[\tan(R_{x,cam}^G - FOV_{wide}) \quad \tan(R_{y,cam}^G - FOV_{high}) \quad -1 \right]^T \quad (3.25)$$

$$\vec{v}_4 = \left[\tan(R_{x,cam}^G - FOV_{wide}) \quad \tan(R_{y,cam}^G - FOV_{high}) \quad -1 \right]^T \quad (3.26)$$

The distance in the threat plan from the UAV to each destination vertex is calculated using the angle between the global z-axis and each vector.

$$\gamma_i = \arccos\left(\frac{\vec{v}_0 \cdot \vec{v}_i}{\|\vec{v}_0\| \|\vec{v}_i\|}\right) \quad (3.27)$$

$$d_i = T_{z,cam}^G \tan(\gamma_i) \quad (3.28)$$

The camera field of view is also used to determine the angle from the global x-axis to the each destination vertex in the threat plane.

$$\beta = \arctan\left(\frac{\tan(0.5FOV_{high})}{\tan(0.5FOV_{wide})}\right) \quad (3.29)$$

$$\left[\alpha_1 \quad \alpha_2 \quad \alpha_3 \quad \alpha_4 \right]^T = \left[\beta \quad \pi - \beta \quad -\pi + \beta \quad -\beta \right]^T \quad (3.30)$$

The destination vertices are subsequently calculated.

$$\begin{bmatrix} D_{i,x} \\ D_{i,y} \end{bmatrix} = \begin{bmatrix} T_{x,cam}^G \\ T_{y,cam}^G \end{bmatrix} + \begin{bmatrix} \cos(R_{z,cam}^G - 0.5\pi) & -\sin(R_{z,cam}^G - 0.5\pi) \\ \sin(R_{z,cam}^G - 0.5\pi) & +\cos(R_{z,cam}^G - 0.5\pi) \end{bmatrix} \begin{bmatrix} d_i \cos(\alpha_i) \\ d_i \sin(\alpha_i) \end{bmatrix} \quad (3.31)$$

3.5.4.5 Homogeneous Transformation Matrix

A three-by-three transformation matrix transforms the normalized source points to the threat plane [22].

$$H = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & 1 \end{bmatrix} \quad (3.32)$$

The eight unknown transformation coefficients are solved algebraically using the linear relationship between the source and destination points [23].

$$\begin{bmatrix} H_{11} \\ H_{12} \\ H_{13} \\ H_{21} \\ H_{22} \\ H_{23} \\ H_{31} \\ H_{32} \end{bmatrix} = \begin{bmatrix} S_{1,x} & S_{1,y} & 1 & 0 & 0 & 0 & -S_{1,x}D_{1,x} & -S_{1,y}D_{1,x} \\ 0 & 0 & 0 & S_{1,x} & S_{1,y} & 1 & -S_{1,x}D_{1,y} & -S_{1,y}D_{1,y} \\ S_{2,x} & S_{2,y} & 1 & 0 & 0 & 0 & -S_{2,x}D_{2,x} & -S_{2,y}D_{2,x} \\ 0 & 0 & 0 & S_{2,x} & S_{2,y} & 1 & -S_{2,x}D_{2,y} & -S_{2,y}D_{2,y} \\ S_{3,x} & S_{3,y} & 1 & 0 & 0 & 0 & -S_{3,x}D_{3,x} & -S_{3,y}D_{3,x} \\ 0 & 0 & 0 & S_{3,x} & S_{3,y} & 1 & -S_{3,x}D_{3,y} & -S_{3,y}D_{3,y} \\ S_{4,x} & S_{4,y} & 1 & 0 & 0 & 0 & -S_{4,x}D_{4,x} & -S_{4,y}D_{4,x} \\ 0 & 0 & 0 & S_{4,x} & S_{4,y} & 1 & -S_{4,x}D_{4,y} & -S_{4,y}D_{4,y} \end{bmatrix}^{-1} \begin{bmatrix} D_{1,x} \\ D_{1,y} \\ D_{2,x} \\ D_{2,y} \\ D_{3,x} \\ D_{3,y} \\ D_{4,x} \\ D_{4,y} \end{bmatrix} \quad (3.33)$$

All normalized pixels are mapped and then re-scaled to the global coordinate frame.

$$\rho = \begin{bmatrix} H_{31} & H_{32} & 1 \end{bmatrix} \begin{bmatrix} S_{x,px} \\ S_{y,px} \\ 1 \end{bmatrix} \quad (3.34)$$

$$\begin{bmatrix} T_{x,px}^G \\ T_{y,px}^G \end{bmatrix} = \frac{1}{\rho} \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \end{bmatrix} \begin{bmatrix} S_{x,px} \\ S_{y,px} \end{bmatrix} \quad (3.35)$$

3.5.4.6 Resolution Shift

Since the pixels of each sampled image are mapped to a user-defined resolution, the computational efficiency is increased when the mapped destination vertices of all pixels are rounded to the user-defined resolution and then sorted.

$$\begin{bmatrix} T_{x,px}^G \\ T_{y,px}^G \end{bmatrix} = \text{res} \begin{bmatrix} \text{round}\left(\frac{T_{x,px}^G}{\text{res}}\right) \\ \text{round}\left(\frac{T_{y,px}^G}{\text{res}}\right) \end{bmatrix} \quad (3.36)$$

3.5.4.7 Global Matrix Updating

In the global matrix, the RGB values are updated in batches using a weighted sum of the previous RGB averages and the new RGB averages. The weights are proportional to the number of previously mapped pixels and newly mapped pixels, respectively. First, the batch of RGB values are averaged, for N_0 equals the number of previously mapped pixels and N_1 equals the size of the new batch.

$$R_1 = \sqrt{\frac{1}{N_1} \sum_{i=1}^{N_1} R_i^2}, \quad G_1 = \sqrt{\frac{1}{N_1} \sum_{i=1}^{N_1} G_i^2}, \quad B_1 = \sqrt{\frac{1}{N_1} \sum_{i=1}^{N_1} B_i^2} \quad (3.37)$$

The global RGB values are updated with the new batch average.

$$R^G = \sqrt{\frac{N_0 R_0^2 + N_1 R_1^2}{N_0 + N_1}}, \quad G^G = \sqrt{\frac{N_0 G_0^2 + N_1 G_1^2}{N_0 + N_1}}, \quad B^G = \sqrt{\frac{N_0 B_0^2 + N_1 B_1^2}{N_0 + N_1}} \quad (3.38)$$

The weight of previously mapped pixels is also updated.

$$N^G = N_0 + N_1 \quad (3.39)$$

3.5.4.8 Threat Retrieval

A .csv file contains a column of float values ranging from zero to one, representing the normalized threat intensity, and three columns of RGB values generated using a MATLAB colormap and the threat intensity column vector. The threat resolution is user defined. To calculate the threat at a tracked point, the current RGB values are subtracted from all RGB values in the colormap .csv file. Each RGB vector is normalized, and the index of the minimum is used to retrieve the threat intensity from the colormap .csv file.

3.6 Experimental Lab Set

3.6.1 Hardware

To achieve the stated project objectives, we must develop an easily configurable indoor lab setup to run experiments. This indoor setup reduces the need for large-scale, expensive, and time consuming outdoor experiments that may be infeasible. Further, an indoor configurable lab allows for year round experimentation irregardless of outside temperature and weather. While the indoor laboratory configuration presents several advantages, its principal drawback resides in the absence of a dependable Global Positioning System (GPS) signal.

To resolve this issue, we use a motion-capture VICON system to provide accurate position data for the UAVs and UGVs. The VICON system emulates GPS data allowing the autonomous vehicles to operate as if they were connected to a standard GPS receiver. This system also provides ground truth data, allowing us to study how accurately the autonomous vehicles estimate their positions in 3D space.

The initial setup of the lab is inadequate due to several factors, including the placement of VICON cameras and the ground station setup. The cameras are mounted on large tripods, as seen in Figure 3.31, which can obstruct the movement of UAVs within the lab space and pose significant danger of collisions. Additionally, the VICON system requires recalibration—which can take several hours—if any of the cameras are moved or bumped so it is imperative that we find a more stable lab configuration.



Figure 3.31: Original laboratory setup

Our solution to the aforementioned problems is to mount the cameras on the walls of the lab. We have access to 10 VICON cameras, which is more than sufficient to track any object within the lab. Figure 3.32 shows a top-down overview of the lab environment with the approximate locations of each VICON camera. Walls A-C are solid boundaries of the lab; however, Wall D is an imaginary wall that bisects the lab space.

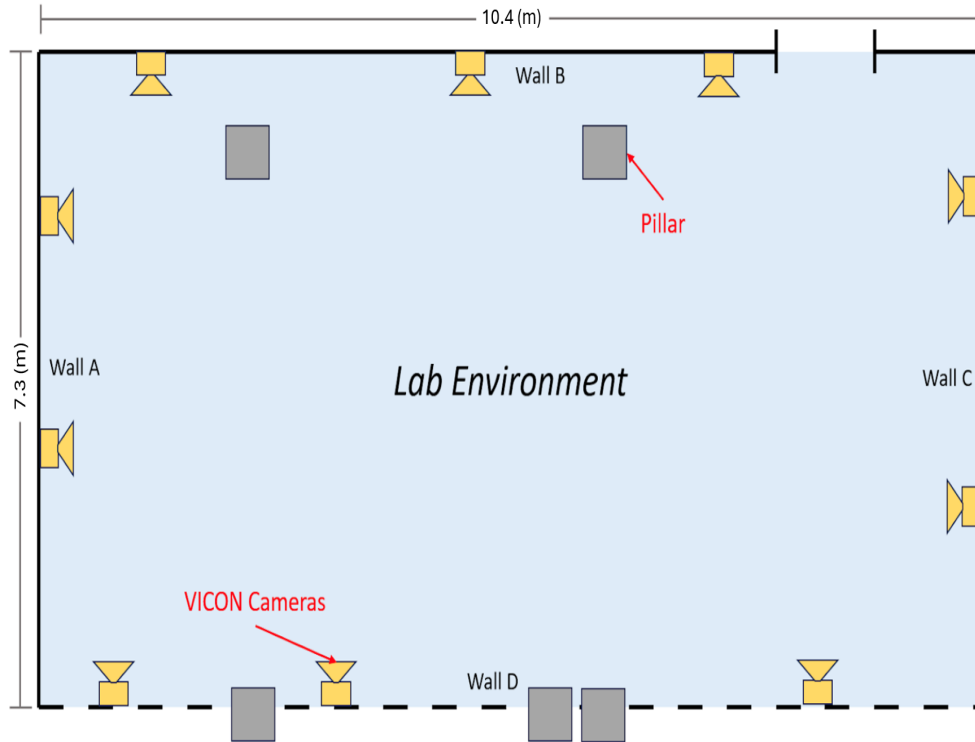


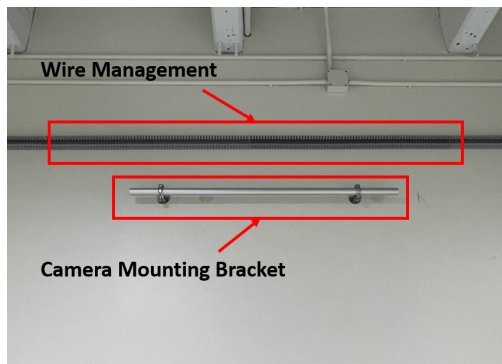
Figure 3.32: Top-Down VICON camera setup

This is because the lab environment does not encompass the entirety of the room in which the experiments take place. The VICON cameras located on Wall D are mounted on a wooden structural support beam unlike the rest of the cameras which are mounted on their respective walls.

The cameras on Walls A-C are mounted horizontally using one inch aluminum pipes and steel brackets secured with drywall anchors. Each camera has its own 4 ft section of pipe for lateral adjustments which allows the camera placement to be reconfigured should a future user deem it necessary. The cameras are mounted as high as possible to avoid interfering with any autonomous vehicles. The cameras mounted on Wall-D are mounted vertically with the steel brackets attached to the ceiling beam.

Above the cameras, we mount plastic cable sleeves and feed the VICON wires to the ground station switchbox. Figure 3.33a shows an example of one of the camera mounts and wire management sleeves before the cameras are mounted. This wire management helps properly organize the 100ft camera wires making the lab environment cleaner and safer. Along Wall D, the wires are mounted directly to the ceiling beam using wire staples due to their convenient location close to the ground station and the difficulty in mounting cable sleeves in that area. The excess cable for each camera is coiled up next to its respective camera and mounted to the wall using J-hooks as seen in Figure 3.33b.

To organize our hardware and clear table space, we shift the computer hardware to a small server rack



(a) Wall camera mount with cable sleeve



(b) Final lab design with cameras mounted, wire sleeves mounted, and wires coiled and hung against the walls

Figure 3.33: Cable sleeves and final lab design

as in Figure 3.34. The main hardware components on the server rack are three switches used to link all ten cameras, a desktop dedicated to running only the VICON software, and a smaller desktop used as a ground control station. With this setup, we can efficiently collect data from lab mounted equipment such as the cameras and any autonomous vehicles operating in the lab environment.

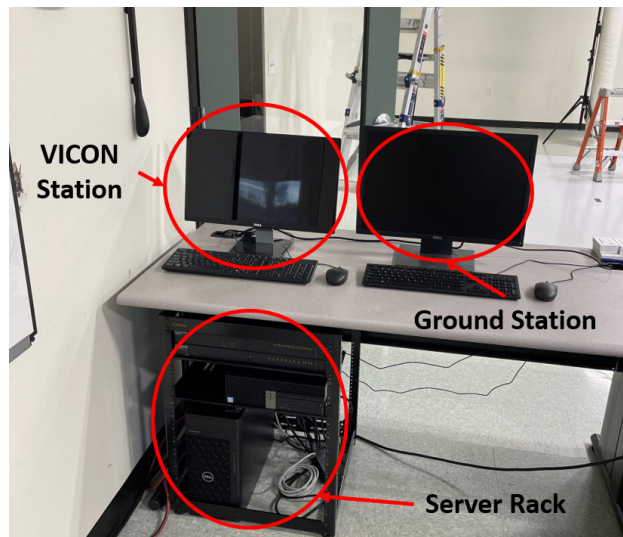


Figure 3.34: Ground station and server rack setup

Having mounted all of the cameras, next we must configure each camera to ensure any motion tracking markers within the lab space are in focus, allowing for accurate data collection. The first step is to line the borders of the lab space with reflective markers and place three additional markers in the center of the lab space as shown in Figure 3.35.

The next step involves securing and positioning a single camera as intended, then adjusting its focus, aperture, and zoom settings to their maximum levels. An iPad is then linked to the Vicon tracking software



Figure 3.35: VICON calibration lab setup

allowing us to see exactly what the camera sees. At this point, the camera view is completely dark and no markers are visible. We close the aperture slightly and slowly close the focus until all of the markers come into view as shown in Figure 3.36.

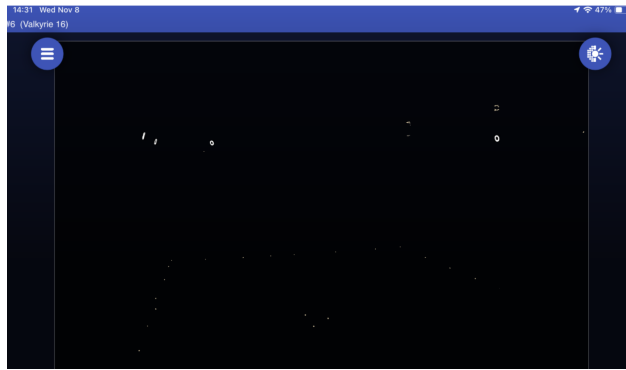


Figure 3.36: Sample Vicon camera view of the lab space with markers. Note, the white rings visible at the top of the screen are other cameras, not markers.

Finally, we zoom in on one of the center three markers on the iPad screen and further adjust the focus to ensure the marker is perfectly in focus and appears white instead of light gray. This step requires fine adjustments and some trial and error with aperture/focus combinations to get the desired focus. Figure 3.37 shows the zoomed in marker before and after adjusting the camera focus.



Figure 3.37: VICON camera focus adjustment process

4 Results

4.1 Experiment 1 Results

4.1.1 Threat Mapping with Ideal Conditions

A unit test consisting a Raspberry Pi Camera Module 2 pointing towards a screen through a pinhole demonstrates threat mapping with near ideal conditions. A known threat field is split into two sections, each with an aspect ratio equivalent to that of the screen. The Figure Figure 4.1a contains the top of the threat field, and Figure Figure 4.1b contains the bottom of the threat field.

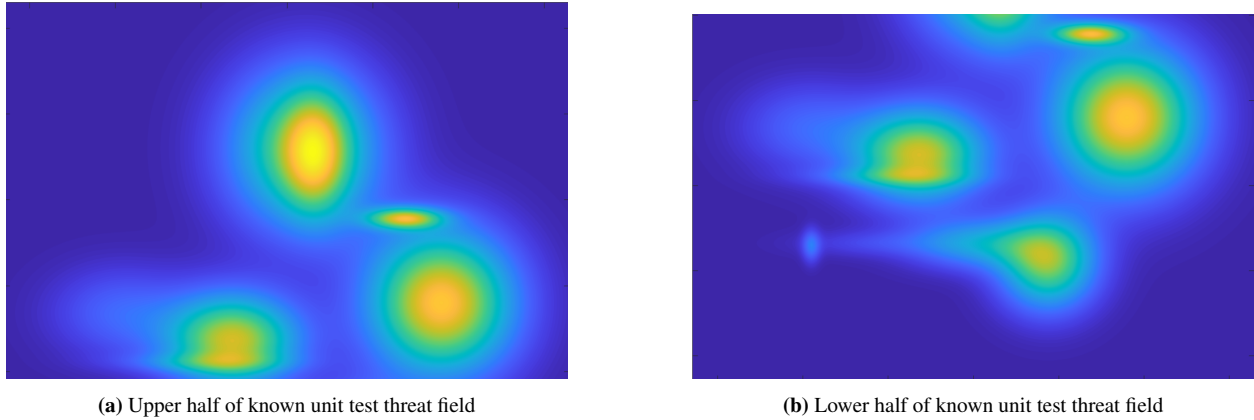


Figure 4.1: Upper and lower half of a known threat field

By assuming zero roll, pitch, and yaw, the simulated position required for the images to overlap correctly is calculated using basic trigonometry. Both images are sampled and mapped to a fully unknown environment, as in Figure 4.2.

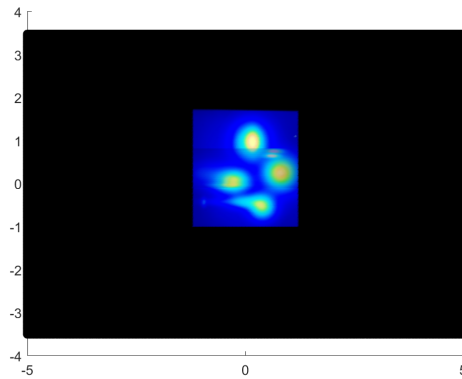
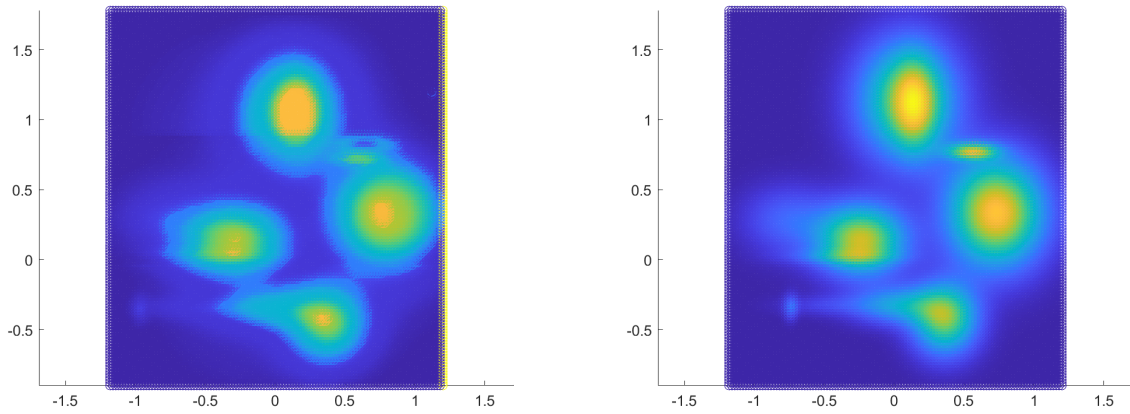


Figure 4.2: Mapping of known threat at simulated positions

This mapping is converted to the known threat field's color map, as is shown in Figure 4.3a. The known threat field is also discretized to the lab resolution in the mapping, as in Figure 4.3b.



(a) Color map-corrected mapping of known threat at simulated position

(b) Discretization of known threat

Figure 4.3: Initial processing of mapped threat field

To evaluate the error between the measured threat and the known threat, we plot the absolute value of the difference between the known and measured threats, shown in Figure 4.4.

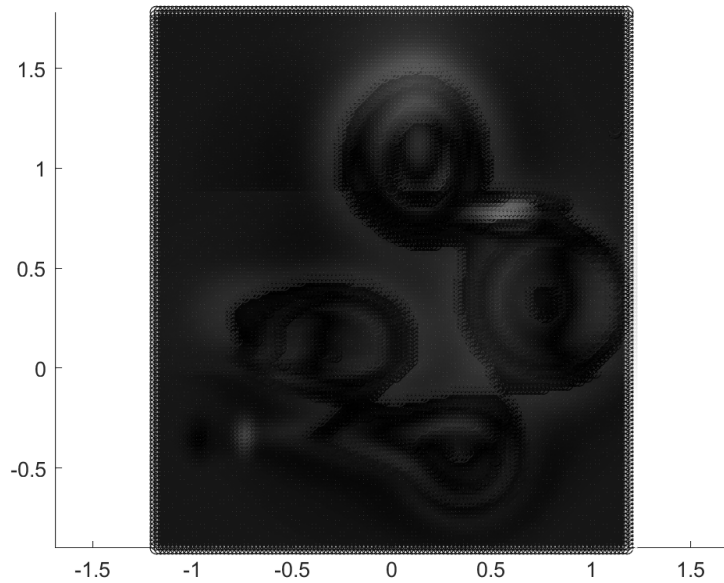


Figure 4.4: Normalized absolute error between known and measured threat

The mean error serves as a metric to evaluate the validity of the measured threat. In this trial, the mean threat value error of each pixel equals 0.0997. However, upon inspection, the greatest errors occur in areas of misalignment or areas exposure diffusion.

4.1.2 Threat Mapping with Non-Ideal Conditions

A threat field consisting of colored construction paper occupies a 8.73 square meter surface in the laboratory environment. A visual light Raspberry Pi Camera Module 2 is mounted to the bottom of one of the UAVs and faces downwards. As seen in Figure 4.5, we attach the UAV to a wooden stick and hold the UAV over the construction paper threat field.



Figure 4.5: Experimental setup for estimating construction paper threat field

We conduct 27 trials attempting to calibrate the measured pose of the UAV and the computer vision program. Although some trials have marginal error (such as in Figure 4.6), many have significant misalignment.

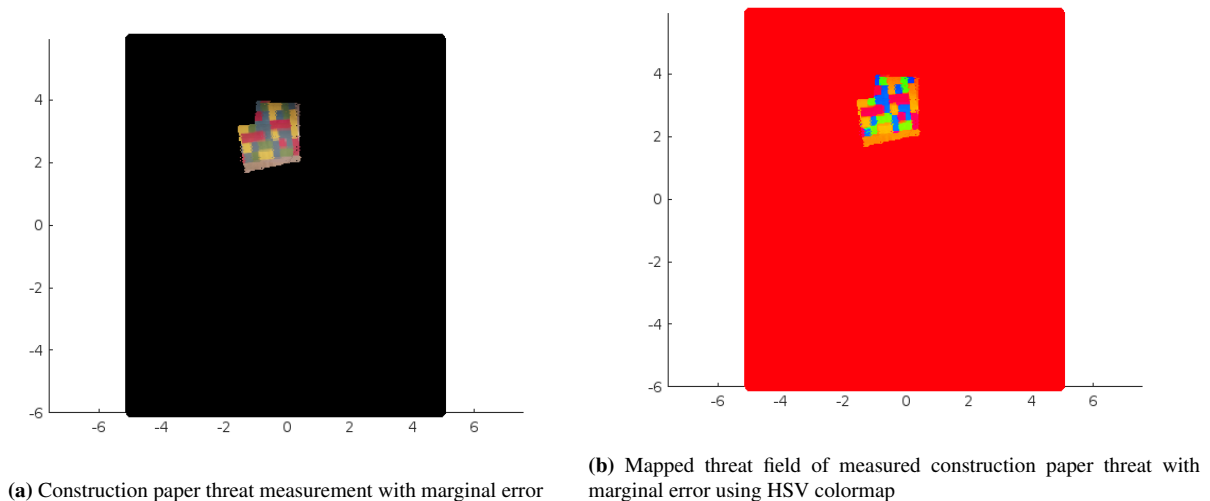


Figure 4.6: Threat estimate and mapping using a construction threat field

In all other trials, the misalignment in the projection of images into the threat plane introduces significant

error, shown in the lack of resolution in Figure 4.7. We unsuccessfully attempt to remedy this error by ensuring the global z-axis in VICON Tracker is properly level, applying a translation to account for the position of the camera with respect to the location of the measured UAV object in VICON Tracker, and modifying the computer vision program.

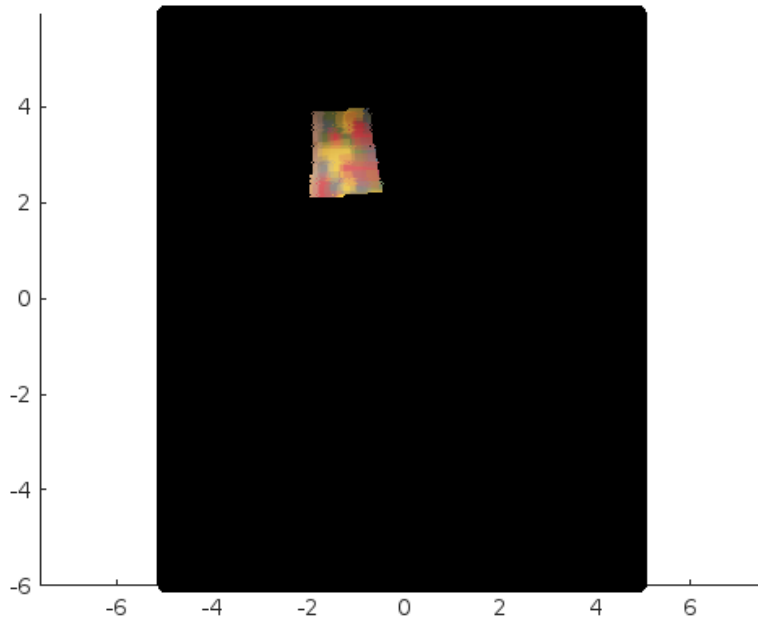


Figure 4.7: Construction paper threat measurement with misalignment

4.2 Experiment 2 Results

4.2.1 UGV Vertex Navigation using OpenCR1.0 Odometry

Initial tests rely on state estimates from the TurtleBot3's OpenCR1.0 odometry since real-time feedback from VICON Tracker is not yet configured. To demonstrate the functionality of the navigation package, we input five vertices in the local frame, track the TurtleBot3 using VICON Tracker for post-analysis, and log the onboard odometry. We transform the logged odometry data to the global frame in post-analysis. Results of this experiment are shown in Figure 4.8.

The TurtleBot3 converges to each vertex according to the onboard state approximations, but does not converge to the true position of each vertex. This is expected since we introduce error by relying on data from the TurtleBot3's odometry. We add additional error by approximating the angle of the z-rotation, but we disregard this error since it is not applicable while using feedback from VICON Tracker.

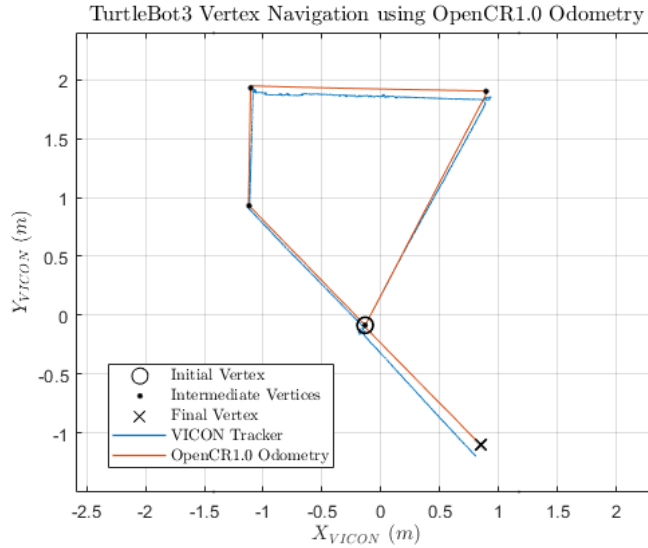


Figure 4.8: Single TurtleBot3 vertex navigation using OpenCR1.0 odometry compared with VICON truth

4.2.2 UGV Line Following in the VICON Environment

After establishing a connection between VICON Tracker and the ROS environment using the `vicon_bridge` package, we demonstrate real-time VICON feedback [19]. We simplify our initial tests in the VICON environment by creating a new line following package consisting of a calibration and a continuous translation. The angular velocity control input is calculated using the line following system model.

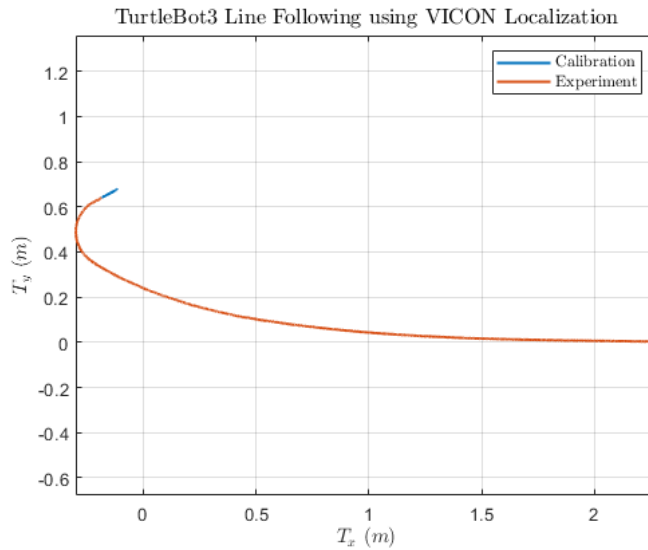


Figure 4.9: Single TurtleBot3 line following in the VICON environment

The TurtleBot3 converges to the positive x-axis in Figure 4.9 in the VICON environment, confirming the feedback works as intended and that the TurtleBot3 object is properly calibrated.

4.2.3 UGV Vertex Navigation in the VICON Environment

We update the navigation package to include a VICON callback function, a function to calibrate the object in VICON tracker, a function to calibrate the onboard odometry, and logic for when the onboard computer should use VICON localization versus odometry. We arrange shipping boxes into a makeshift city within the VICON environment consisting of multiple obstacles and two tunnels. We input eight vertices and test the updated navigation package.

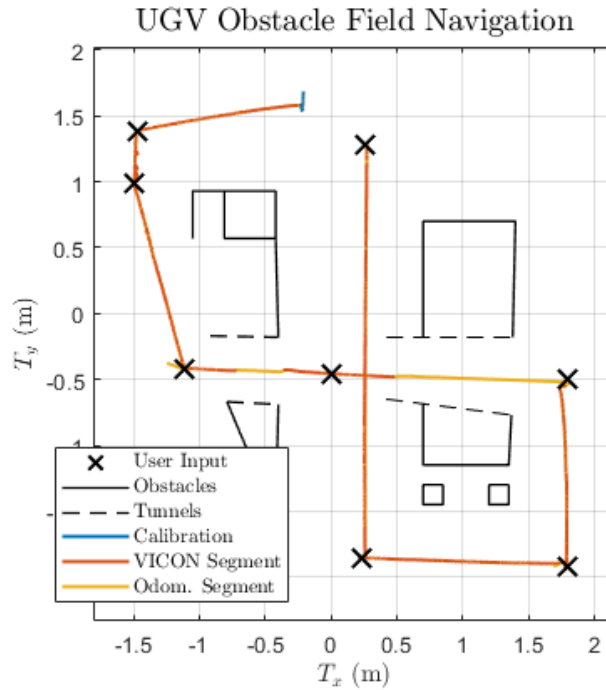


Figure 4.10: Trajectory of single TurtleBot3 vertex navigation in the VICON environment

Upon inspection of Figure 4.10, it appears that the TurtleBot3 converges to the lines from each initial vertex to each target vertex. Additionally, the onboard computer correctly switches to using odometry callbacks during periods of occlusion, such as going through tunnels. We simultaneously demonstrate our onboard data logging by comparing the measured odometry and VICON callbacks to the onboard estimated state and by examining the calibration coefficients.

The plot of the TurtleBot3's global states in Figure 4.11 further confirms that the onboard computer correctly detects when the TurtleBot3 is occluded and that the odometry calibrations are sufficient.

The calibration coefficients in Figure 4.12 also behave as expected. The first four calibration coefficients are constants calculated during the calibration of the TurtleBot3's VICON Tracker object, while the remaining three are calculated in real-time after each VICON callback. Spikes in the dynamic calibration coefficients are due to sensor noise.

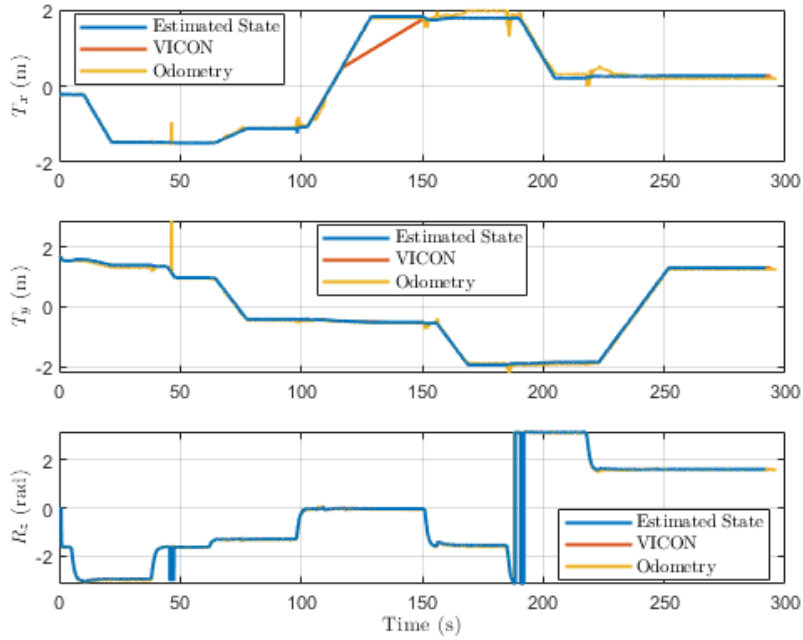


Figure 4.11: States of single TurtleBot3 vertex navigation in the VICON environment

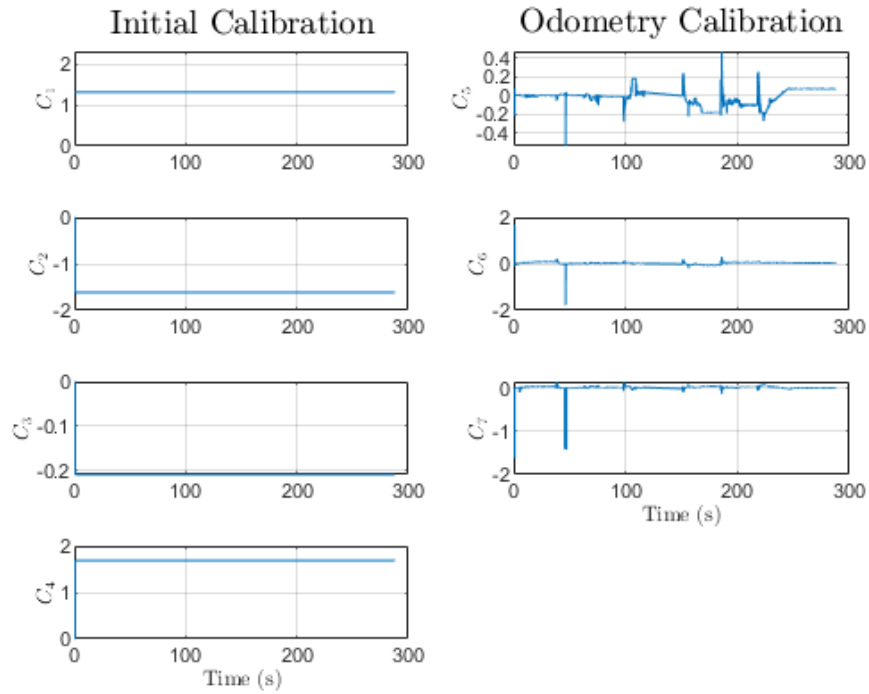
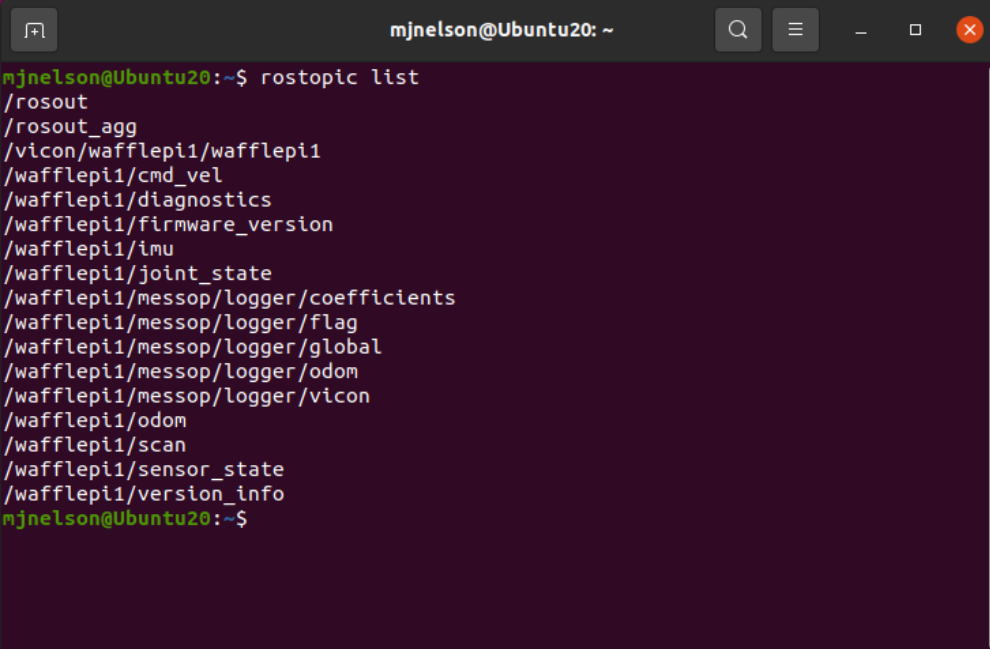


Figure 4.12: Calibration coefficients of single TurtleBot3 vertex navigation in the VICON environment

4.2.4 Multi-UGV Vertex Navigation in the VICON Environment

We receive four TurtleBot3 Waffle Pi to incorporate into the laboratory environment. We build all four and flash the appropriate software. To distinguish between vehicles in ROS, we first create a roslaunch file that includes the bringup launch file, messop node, and logger node within a group. We add a name space tag to the group that remaps all nodes and topics to include the vehicle name as a prefix, as shown in Figure 4.13.



```
mjnelson@Ubuntu20: ~  
mjnelson@Ubuntu20:~$ rostopic list  
/rosout  
/rosout_agg  
/vicon/wafflepi1/wafflepi1  
/wafflepi1/cmd_vel  
/wafflepi1/diagnostics  
/wafflepi1/firmware_version  
/wafflepi1/imu  
/wafflepi1/joint_state  
/wafflepi1/messop/logger/coefficients  
/wafflepi1/messop/logger/flag  
/wafflepi1/messop/logger/global  
/wafflepi1/messop/logger/odom  
/wafflepi1/messop/logger/vicon  
/wafflepi1/odom  
/wafflepi1/scan  
/wafflepi1/sensor_state  
/wafflepi1/version_info  
mjnelson@Ubuntu20:~$
```

Figure 4.13: Remapped TurtleBot3 bringup, messop, and logger topics with vehicle name prefix

We place three TurtleBot3 Waffle Pi in the VICON environment and successfully input vertices for each vehicle to transition to. However, there is time desynchronization between the core processes launched during bringup and the navigation and logger node, causes the vehicles to over rotate and over translate. We unsuccessfully try separating the launch file into two .launch files to launch the bringup separately from the custom nodes. We suspect that the name space tag introduces time delay due to the quantity of messages being remapped across multiple topics. We next try to hard-code the vehicle name into the bringup package and the navigation and logger nodes. The combination of modifying the TurtleBot3 diagnostics in the bringup package, using remap tags in the launch files, and decreasing the proportional gain experimentally corrects the issue.

4.3 Experiment 3 Results

4.3.1 Experiment Setup

This experiment phase focuses on collecting multi-modal data in a system with multiple aerial and ground vehicles, and fixed environmental sensors. Figure 4.14 shows the assets we use in this experiment.

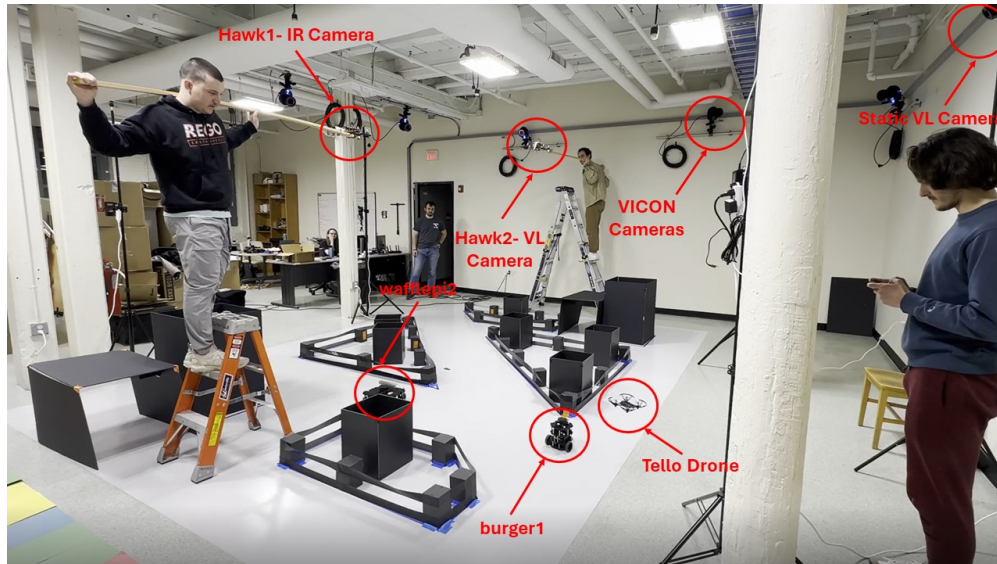


Figure 4.14: Phase 3 Experimental Setup

We use a TurtleBot 3 Waffle Pi (waffle) and a TurtleBot 3 Burger Pi (burger) as ground vehicles. We equip the waffle with a heating strip attached to a metal bar and command the vehicle to navigate to waypoints via the MESS. Due to time synchronization issues inherent to commanding multiple vehicles, we command the burger manually.

We equip the Hawk1 UAV with an infrared (IR) camera and Hawk2 with a visual light (VL) camera. Since we never achieve stable flight using mavros, we opt to attach the UAVs to wooden posts and move the posts to simulate flight. The Tello drone, which we control manually, has a forward-facing camera.

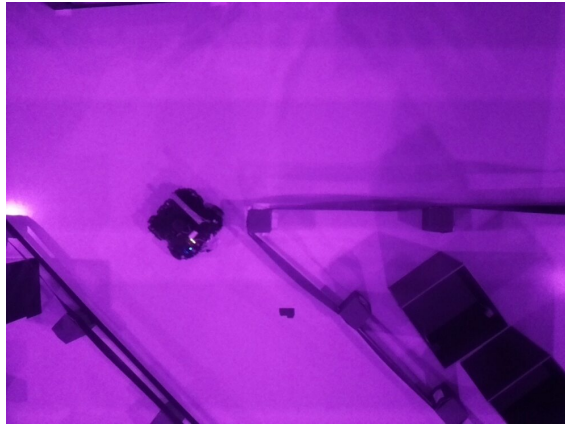
In addition to vehicle-fixed sensors, we use a 4K visual light camera mounted on a tripod. This camera captures the entire environment.

4.3.2 Sensor Data

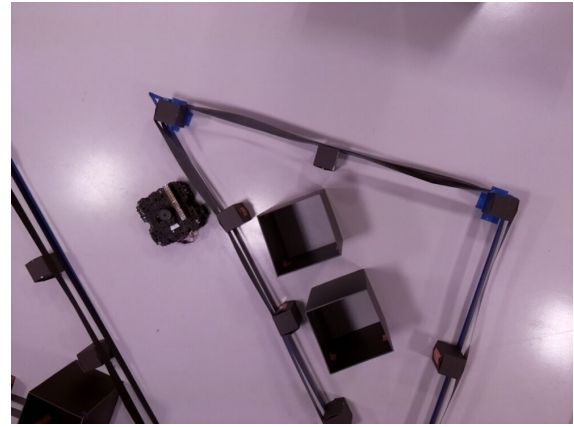
We capture 798 images over the course of about 6 minutes while running the experiment.

Figure 4.15a shows the waffle driving underneath the IR camera on Hawk1 with the lights in the lab shut off. The line visible on the waffle is the metal bar that houses the heating element. While the heating element is active, the IR camera does not detect a heat signature. This can be seen as the strip on the waffle is the same color as the background, and the floor of the lab is not the same temperature as the heating element.

Figure 4.15b shows the waffle driving underneath the IR camera on Hawk1 with the lights in the lab illuminated. Much like when the lights are off, the camera does not detect a heat signature coming from heating element.



(a) Image captured with an IR camera on the Hawk1 UAV with overhead lights off



(b) Image captured with an IR camera on the Hawk1 UAV with overhead lights on

Figure 4.15: IR images captured by Hawk1 with lights off and on

Figure 4.16 is an image of the waffle and burger passing each other. Over the course of the experiment, the Hawk2 VL camera captured 1882 images.

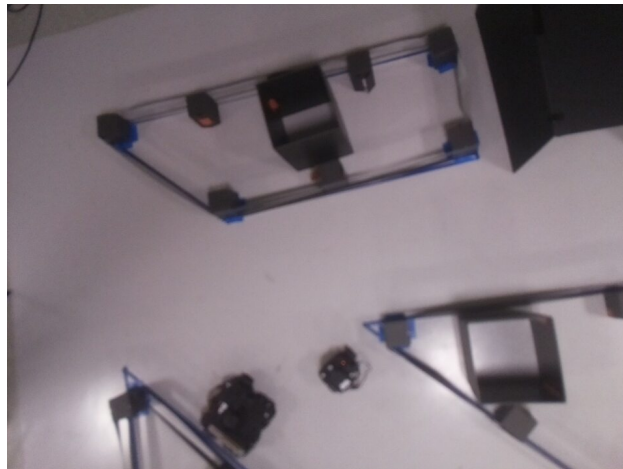


Figure 4.16: Image captured with a VL camera on the Hawk2 UAV

In addition to images from the IR and VL cameras, a static camera captures overhead video, which can be found here https://youtu.be/8_nLpJBp0Ls, and another camera captures a side view <https://youtu.be/optcQQL-fFo>. We also record the VICON position data for each vehicle in the experiment. Finally, we record and transfer the odometry and position logs from the waffle via MESS.

4.4 UAV Flight Testing

4.4.1 Indoor Flight

Upon switching to PX4, we achieve stable, level flight. We use the Manual/Stabilize option in QGroundControl, which levels the vehicle in flight. It does not maintain either altitude or position [24], so the vehicle may drift during flight. In this flight mode, we can hold a stable position, and take off in a level orientation. However, true to the documentation, the UAV is subject to drift in the indoor environment. Altitude control is completely subject to the operator, but is relatively easy to control by adjusting the throttle. A short flight with each of the UAVs can be viewed at the following links: <https://youtu.be/UrG0fQwiXDQ> and <https://youtu.be/FEtOMRbK9r0>.

We also perform basic roll, pitch, and yaw tests. We tip the UAV to either side, and return to the level position with no oscillations. We perform this test with both UAVs, with one equipped with the RPi and RPi camera. In both cases, the UAV is easily able to take off and stabilize itself, indicating that the build and components are correct. One thing that we notice is that one of the flight controllers loses barometer readings during initial flight testing. While we disable this reading and still achieve stable flight, the UAV with a working barometer is less sensitive to throttle changes. This may be due to the barometer reading, or may be due to the slight configuration differences of the two UAVs.

4.4.2 Autonomous flight (in SITL)

When first testing mavros, we use simulation software. We choose ArduPilot's Software in the Loop (SITL) capabilities to test our mavros connections and Python code. With this capability, we can arm the vehicle, takeoff, move to a waypoint, land, and disarm. These mission stages are easy to implement in mavros. To do so, we use a Python script that can publish and receive information from the mavros package. As a result, we can read the state of the flight controller at any point during the mission and direct the flight controller.

We choose to perform simulated tests to ensure that the code performs how we expect. Once we assemble the UAV, we then move to testing the code with a companion computer, in this case the Raspberry Pi 3 (RPi). Originally, with the ArduPilot software, we can connect to mavros easily. However, once we flash PX4 onto the flight controller, we begin running into processing issues. A key part of ROS is time synchronization, which allows the software to know when commands are sent and received. However, when we attempt to use a baud rate of 921,600 B/s, the RPi first begins to restart the UART port that is connected to the flight controller. If left long enough, the RPi crashes and shuts down. As a result, we never achieve a successful autonomous flight test. This is mainly because without proper time synchronization, we do not want to attempt to fly the UAV, which can cause substantial damage if control is lost from the RPi.

However, we do validate the physical connection and mavros by setting the baud rate to 38,400 B/s. With a low baud rate, we can send and receive information from the flight controller (for instance the charge of the battery). This is an important first step into connecting to the UAV with mavros. For future work, we recommend that the same software and physical connections are used. However, we recommend troubleshooting the RPi to either increase the baud rate it can handle, or to purchase an RPi with greater processing capabilities. Other considerations would be changing the physical connection or cable between the flight controller and RPi, switching software platforms (from mavros to either DroneKit, ROS 2, or MAVSDK), or testing mavros with ArduPilot. The last option would lead to unstable flight once again but would allow the user to test if the issues lies with PX4 or mavros. If the connection works between ArduPilot and the RPi via mavros, it is possible that certain configuration parameters could be set in PX4 to resolve the baud rate problems.

4.5 Wind Tunnel Results

4.5.1 Data Processing

The goal of wind tunnel data processing and analysis is to correlate motor speed data from the Ardupilot logs with thrust and torque data from the thrust stand. The main challenge with this methodology is time alignment of the data. We do not have time to develop a sophisticated method for conducting a wind tunnel test. Thus, we use separate logging software for the UAV and the thrust stand. Although this is simple to configure and operate, we are left with data lacking time synchronization.

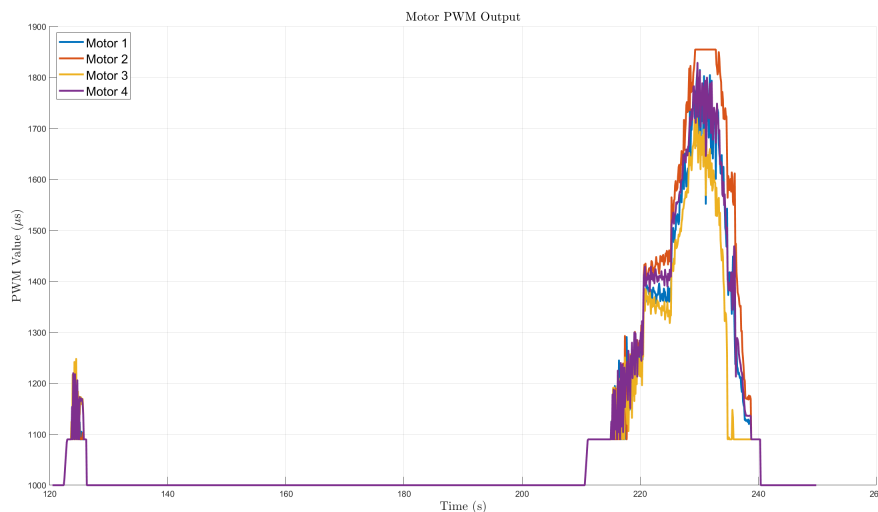


Figure 4.17: Raw motor PWM readings from Ardupilot log files

To remedy this issue, we must examine experimental data from both log files and manually align the time intervals. Figure 4.17 shows the raw motor PWM readings measured by the flight controller during one

test. The first step in the analysis is to isolate the time interval of the experiment. This is done manually for each run, and the resulting data should look similar to Figure 4.18, which shows one such reduced dataset.

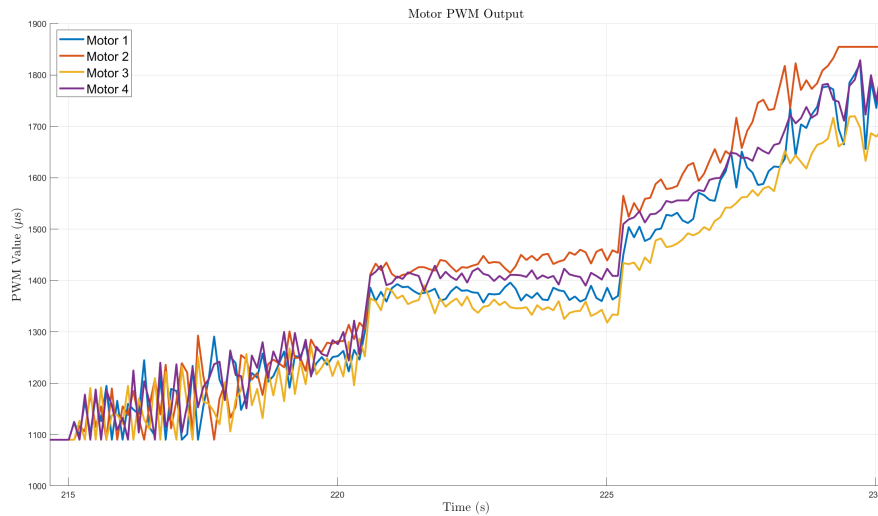


Figure 4.18: Motor PWM readings in time interval of interest

We process the thrust stand data similarly to the UAV log files. However, instead of defining a time interval, we only need to define a starting time. This is because we want the time interval for the log data and thrust stand data to be the same length. Thus we define only a start point for the thrust stand data and then enforce the same duration as the log files. Figure 4.19 shows a sample graph of the raw thrust stand data.

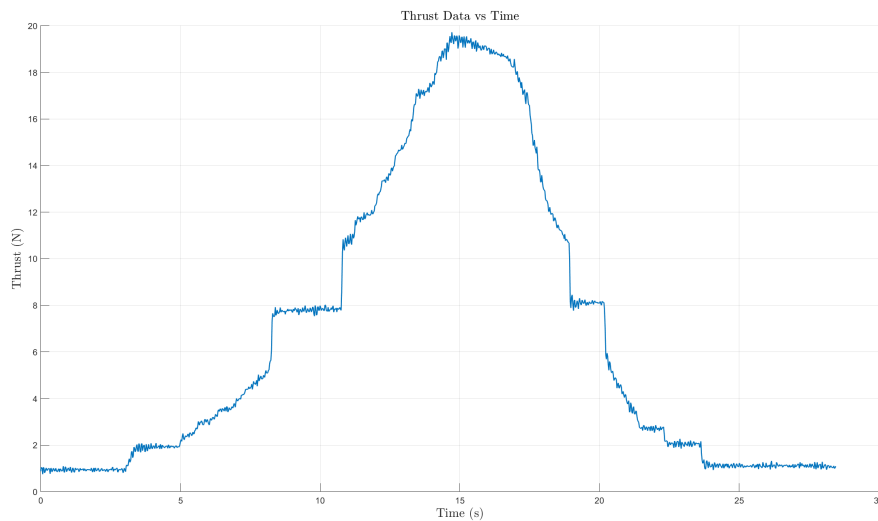


Figure 4.19: Raw thrust stand data collected during a wind tunnel experiment

After choosing time intervals, we analyse the resulting truncated dataset to determine if our data is properly aligned. At this stage we curve fit the data and interpolate the curve at fixed timestamps so that we

can then remove the time dependency. Figure 4.20 shows the results of this alignment for the thrust data. Note that at this point we have taken the average of motor PWM value at a given timestamp because we spin all the motors simultaneously.

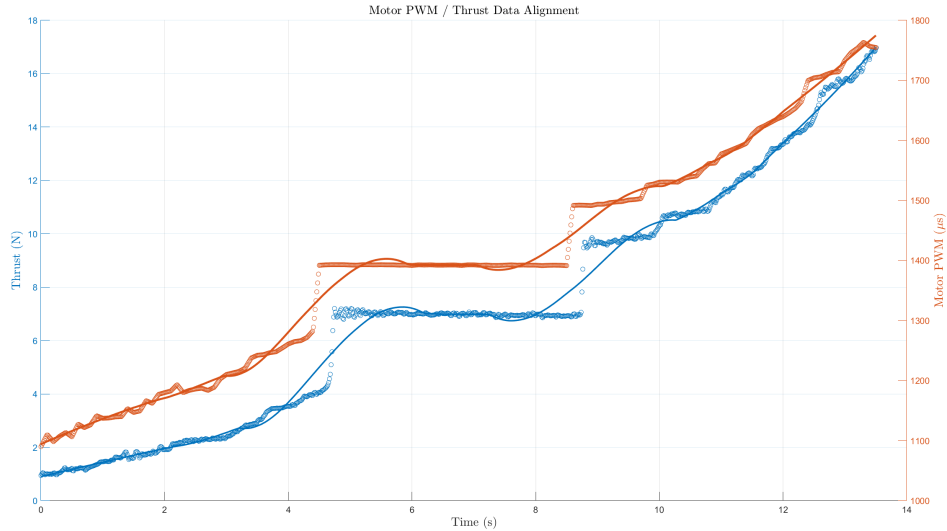


Figure 4.20: Motor PWM and thrust stand data alignment

After aligning the thrust stand data, we completely remove time dependency by associating motor PWM values with their respective thrust value. We can then perform a simple quadratic curve fit to find the quadratic coefficient. Figure 4.21 shows the results of this curve fitting as well as a 95% confidence interval for the accuracy of the curve fit. As seen in Figure 4.22 the torque data is more disordered, as evidenced by the larger confidence interval. We will discuss this in more detail in subsequent sections.

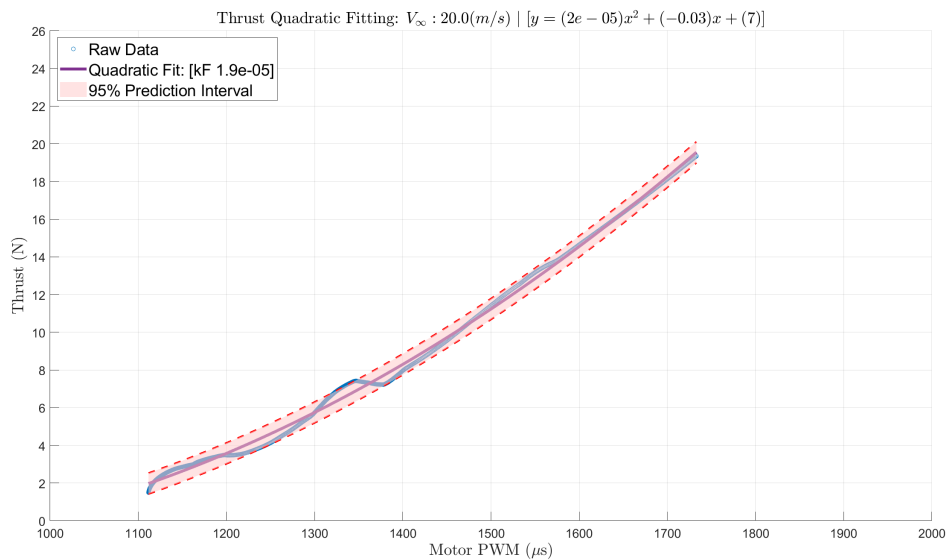


Figure 4.21: Quadratic curve fitting for thrust data

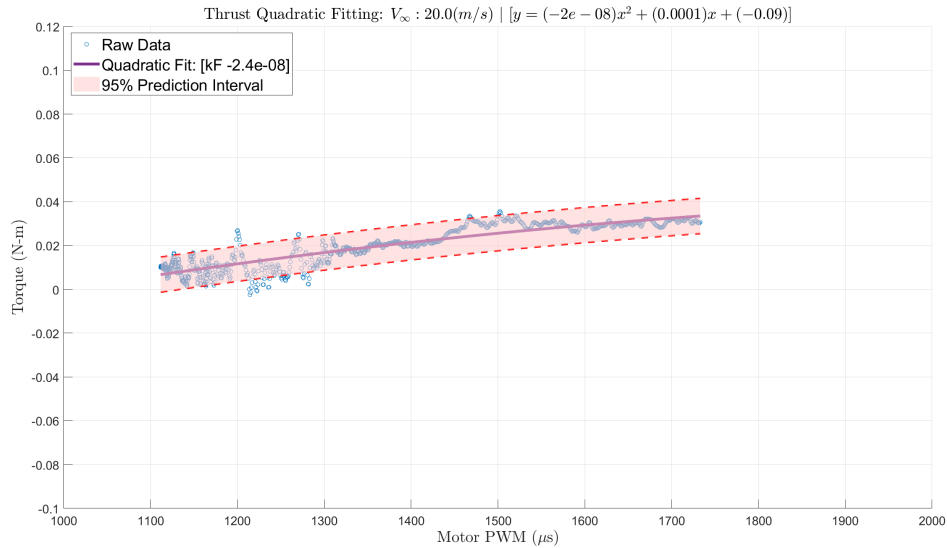


Figure 4.22: Quadratic curve fitting for torque data

4.5.2 Data Analysis

After processing all thrust and torque data, we are left with Figure 4.23 and Figure 4.24 which show the final quadratic curve fit for each wind speed. The title of each graph contains the final average value for both k_F and k_M , both of which are a simple average of each quadratic coefficient.

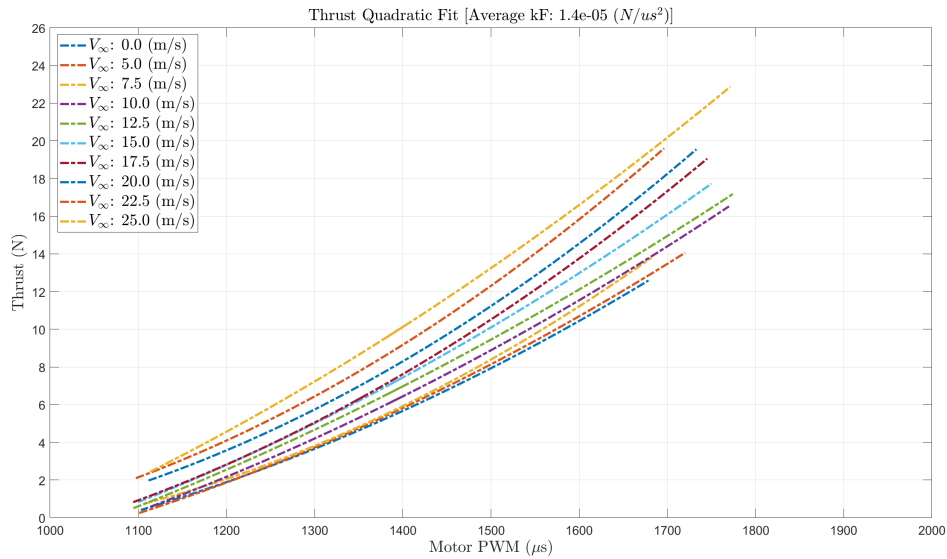


Figure 4.23: Thrust curve fit data for each wind tunnel condition

We consider enforcing some boundary conditions on the individual curve fits, such as enforcing that at 1000 PWM—which is equivalent to 0 rpm—there should be no thrust. We decide this approach is not necessary because the relationship between PWM and thrust is more nonlinear as the motors are closer to zero RPM. Also, since the steady state armed motor PWM is 1100, so we would never attempt to control

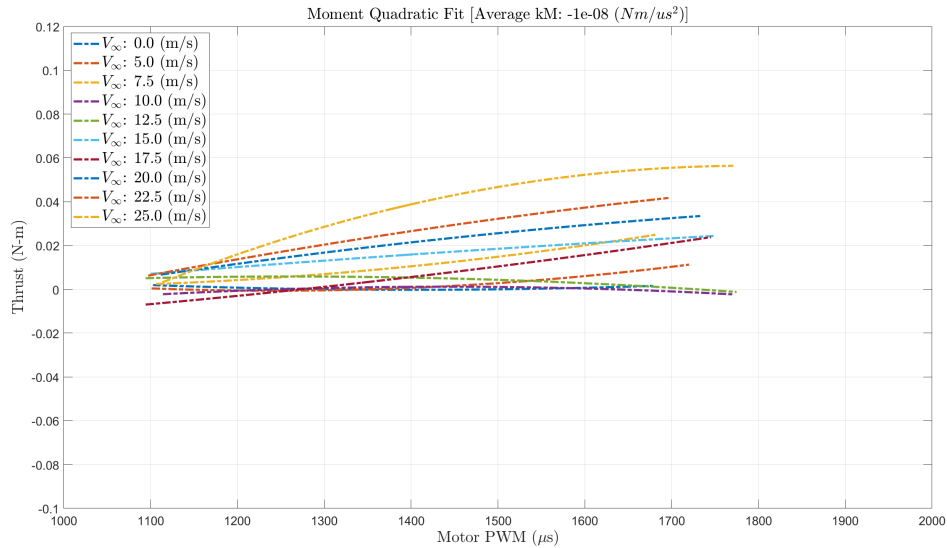


Figure 4.24: Moment curve fit data for each wind tunnel condition

the motors at a smaller PWM value.

The thrust curves are much more consistent with their concavity and magnitude compared to their torque counterparts. We conclude that this is most likely due to the lack of torque accuracy with our thrust stand at very low torque values.

4.5.3 Conclusions and Future Work

The wind tunnel testing proved an effective way of measuring aerodynamic coefficients for UAVs. The scope of this work did fall short of fully accomplishing the goal of directly relating motor rpm to UAV forces and moments. Since the motor rpm value was not a direct output of the Ardupilot log files, we could not easily associate rpm to thrust. However, through further testing, it is possible to convert the motor PWM output to rpm by finding the relationship between PWM and rpm for these specific motors.

Another potential avenue for future work would be to determine the moment of inertia about each Cartesian axis for our UAV design. This data is the final piece of information required to simulate the UAV's movement using a differential flatness model. Moment of inertia data can be found experimentally or by modeling the UAV in a CAD software such as Solidworks.

5 Conclusions

5.1 Conclusions

The main goal of this project is to design a modular lab set-up that can be used to collect small data sets for research in coupled sensor configuration and planning (CSCP). Our lab set-up requires minimal coding experience and knowledge on the part of the user and can accommodate multiple types of sensors. It supports both ground vehicles and UAVs.

Throughout the course of the project, the team on-boarded multiple robotic vehicles, sensors, and UAVs. A modular experimental software system (MESS) allows any user to specify the objectives of different vehicles in the experiment, save and load data, and save experimental configurations. Multiple UAVs were flight tested, with and without sensor payloads. Ultimately, the UAVs were not incorporated into the MESS due to late achievement of stable flight, as well as difficulties connecting to mavros.

The MESS was tested during a large experiment involving multiple robotic vehicles and sensors, proving the framework and data collection. The UAVs were flight tested to demonstrate stable flight, but were not included in the MESS, nor demonstrated autonomous flight. Overall, these tests demonstrated the feasibility and usability of the software, as well as compatibility between all components on the UAVs.

Through this MQP, the team learned about the importance of systematic testing and troubleshooting. While flying the UAV may sound easy, we found many setbacks and small challenges along the way. Many of these issues were fixed by testing each component individually to find any errors with the system. We also learned that we do not have to sustain our first decision. Throughout the project, we found that we had to switch components and change our vision. For instance, we re-built the UAV when the 2022 UAV presented too many challenges and broken components. We also switched coding languages partway through the project and evaluated multiple ground vehicles. While one choice may be the easiest, it may not be the best, so we found times when we needed to pivot quickly to avoid setbacks. Finally, we learned to always over-estimate the amount of time that things would take, especially regarding configuration and software.

5.2 Future Work

While we met many of the objectives set at the beginning of the project, or made significant progress towards them, there is still future work that can be done to refine and expand the capabilities of the current lab setup.

For the VICON motion capture system, future work should focus on increasing the field of view of the cameras. As it stands, the VICON cameras are all mounted higher than 7 feet above the ground, which allows the cameras to cover the airspace, but not the entire floor space. There still exist areas, especially in the corners of the lab space, where objects become occluded and lose useful position data. There are ten additional cameras available to us, but given the scope and time-frame of the project we did not install them. Our recommendation for future work is to mount the additional cameras closer to the floor to supplement the current setup. In doing this, the lower cameras can be configured to view all ground activity and the higher cameras can be configured to capture the entire airspace and some of the floor space as well. With all of the cameras mounted, future experiments should have very accurate coverage of the entire lab space and can mitigate interference from obstacles in the experiments.

For the UAV, future work should focus on achieving autonomous indoor flight. We were unable to achieve fully autonomous flight because of issues with the Raspberry Pi boards (both RPi 3 and 4 boards) crashing when communicating with the flight controller. Future work should investigate alternatives to the Raspberry Pi with better computing capabilities that can handle the large amounts of data output by the flight controller. Additionally, future work could investigate other software/protocols for communicating with the flight controller through the Raspberry Pi. We have identified MAVSDK and DroneKit as two potential alternatives, but ultimately these may not integrate as well with the MESS and ROS. PX4 also recommends using ROS2 instead of ROS, so future work could explore using ROS2 on the UAVs to achieve autonomous flight. This will unfortunately make it difficult to integrate the UAVs with the MESS, since the ground vehicles are based on ROS 1 not ROS2, but it could be worth pursuing autonomous flight independent of the MESS to prove its possible before then figuring way to incorporate with MESS.

For the computer vision, future work should focus on implementing point cloud registration. The current computer vision program relies purely on geometry to align images, and calibration issues results in critical misalignment. Future work should investigate updating the initial transformation of the sampled images using previously sampled images to estimate the threat in the environment. Additional work may include point cloud registration of obstacles in the laboratory environment if users want a threat field that is not relatively flat.

Within the MESS improvements can be made to the mission planning. MESS cannot perform actions outside of waypoint navigation. For example, there is no way to tell a sensor to activate after a vehicle reaches a destination. Additionally, safeguards should be implemented. These safeguards should include obstacle avoidance, ensuring the user can only command actions at the proper time, and should overall guide the user through the process of configuring an experiment. For example, MESS currently allows a user to

run an experiment before launching vehicles, when vehicles should be launched before running.

Bibliography

- [1] F. Ahmed, J. C. Mohanta, A. Keshari, and P. S. Yadav, “Recent Advances in Unmanned Aerial Vehicles: A Review,” *Arabian Journal for Science and Engineering*, vol. 47, pp. 7963–7984, July 2022.
- [2] Z. Zhang and L. Zhu, “A Review on Unmanned Aerial Vehicle Remote Sensing: Platforms, Sensors, Data Processing Methods, and Applications,” *Drones*, vol. 7, p. 398, June 2023.
- [3] A. Gupta and X. Fernando, “Simultaneous Localization and Mapping (SLAM) and Data Fusion in Unmanned Aerial Vehicles: Recent Advances and Challenges,” *Drones*, vol. 6, p. 85, Mar. 2022.
- [4] X. Yin and S. Lafortune, “A general approach for optimizing dynamic sensor activation for discrete event systems,” *Automatica*, vol. 105, pp. 376–383, July 2019.
- [5] K. Yousif, A. Bab-Hadiashar, and R. Hoseinnezhad, “An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics,” *Intelligent Industrial Systems*, vol. 1, pp. 289–311, Dec. 2015.
- [6] A. Bachrach, S. Prentice, R. He, P. Henry, A. S. Huang, M. Krainin, D. Maturana, D. Fox, and N. Roy, “Estimation, planning, and mapping for autonomous flight using an RGB-D camera in GPS-denied environments,” *The International Journal of Robotics Research*, vol. 31, pp. 1320–1343, Sept. 2012.
- [7] B. S. Cooper and R. V. Cowlagi, “Interactive planning and sensing in unknown static environments with task-driven sensor placement,” *Automatica*, vol. 105, pp. 391–398, July 2019.
- [8] C. Laurent and R. Cowlagi, “Coupled Sensor Configuration and Path-Planning in Unknown Environments with Adaptive Cluster Analysis,” (Atlanta), 2022.
- [9] P. Zulch, M. Distasio, T. Cushman, B. Wilson, B. Hart, and E. Blasch, “ESCAPE Data Collection for Multi-Modal Data Fusion Research,” in *2019 IEEE Aerospace Conference*, (Big Sky, MT, USA), pp. 1–10, IEEE, Mar. 2019.
- [10] H. Shakhathreh, A. H. Sawalmeh, A. Al-Fuqaha, Z. Dou, E. Almaita, I. Khalil, N. S. Othman, A. Khreishah, and M. Guizani, “Unmanned aerial vehicles (uavs): A survey on civil applications and key research challenges,” *Ieee Access*, vol. 7, pp. 48572–48634, 2019.

- [11] Z. Ameli, Y. Aremanda, W. A. Friess, and E. N. Landis, “Impact of uav hardware options on bridge inspection mission capabilities,” *Drones*, vol. 6, no. 3, p. 64, 2022.
- [12] “Zipline 2023 health impact report.” <https://www.flyzipline.com/newsroom/stories/2023-impact-report>.
- [13] M. Yaqot and B. Menezes, “The good, the bad, and the ugly: review on the social impacts of unmanned aerial vehicles (uavs),” in *International Conference of Reliable Information and Communication Technology*, pp. 413–422, Springer, 2021.
- [14] “Turtlebot3 quick start guide.” <https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/>.
- [15] “Mandatory Hardware Configuration.” <https://ardupilot.org/copter/docs/configuring-hardware.html>.
- [16] “Standard Configuration.” <https://docs.px4.io/main/en/config/>, Jan. 2024.
- [17] “Aws-deepracer.” <https://github.com/NESTLab/aws-deepracer>, 2023.
- [18] D. J. Balkcom and M. T. Mason, “Time optimal trajectories for bounded velocity differential drive vehicles,” *The International Journal of Robotics Research*, vol. 21, p. 199–217, Mar. 2022. DOI: 10.1177/027836402320556403.
- [19] “vicon_bridge.” https://github.com/ethz-asl/vicon_bridge.
- [20] “raspicam_node.” https://github.com/UbiquityRobotics/raspicam_node.
- [21] “turtlebot3_bringup.” <https://github.com/ROBOTIS-GIT/turtlebot3>.
- [22] R. Hartley and A. Zisserman, “Estimation - 2D Projective Transformations,” in *Multiple View Geometry in Computer Vision*, New York, NY, USA: Cambridge University Press, 2nd ed., 2004.
- [23] I. Gkioulekas, “Image Homographies.” Feb. 2020.
- [24] “Flight Modes (Multicopter).” https://docs.px4.io/main/en/flight_modes_mc/, Feb. 2024.
- [25] “Ubuntu Install of ROS Noetic.” <https://wiki.ros.org/noetic/Installation/Ubuntu>.
- [26] “Installing ROS.” <https://ardupilot.org/dev/docs/ros-install.html>.

A Raspberry Pi and Sensor Configuration

This section details our process for installing and using Ubuntu 20.04, ROS Noetic, and Ubiquity Robotics’s “raspicam_node” ROS Noetic package on a Raspberry Pi 3B. Unless stated otherwise, all of the following commands should be done in an Ubuntu terminal on the Raspberry Pi.

Ubuntu 20.04 Server Installation & Setup

Flash microSD

1. Install Raspberry Pi Imager onto your device.
2. Connect the microSD to your device using a card reader and open Raspberry Pi Imager.
3. Click “CHOOSE DEVICE” and select “Raspberry Pi 3”
4. Click “CHOOSE OS” and select “Other general-purpose OS” → “Ubuntu” → “Ubuntu Server 20.04.05 LTS (32-bit)”
5. Click “CHOOSE STORAGE” and select the microSD
6. Click “WRITE” to flash the Ubuntu 20.04 Server image onto the microSD.

Note: Raspberry Pi Imager may ask for a network SSID and password before writing. Please skip this step as Network information is manually configured after the first Ubuntu boot.

Boot Ubuntu

Connect a keyboard, mouse, and monitor to the Raspberry Pi and then connect a power supply. As Ubuntu boots for the first time, you will be asked to choose a new password. If you are prompted to login first, “ubuntu” is both the default username and password. After you set a new password, Ubuntu will continue to boot.

Configure Network Information

To connect your Raspberry Pi to a network, you must create a YAML configuration file. Run the following command in the terminal:

```
sudo nano /etc/netplan/01-network-manager-all.yaml
```

The file that you create will be blank initially. A sample YAML configuration file for a WiFi connection is provided below. For a wired connection, you will need to modify the sample YAML configuration file.

```
network:
  version: 2
  renderer: networkd
  wifis:
    wlan0:
      dhcp4: true
      access-points:
        {SSID}:
          password: {password}
```

Copy the sample YAML configuration file into the blank file you just created. Change the {SSID} and {password} text in the sample YAML configuration file to the SSID and password of your network (do not include the curly brackets).

Once you have entered all information into the YAML configuration file, save it and exit the text editor. Apply the changes to the network configuration by running the following command in the terminal:

```
sudo netplan apply
```

Edit Bash Script

To enable communication with the ROS master, you must edit the bash script to export the ROS master and the ROS host IPs. Using the terminal, open the bash script.

```
sudo nano ~/.bashrc
```

In the text editor, add the following code at the bottom of the bash script.

```
export ROS_MASTER_URI=http://{IP_OF_ROS_MASTER_DEVICE}:11311
export ROS_HOSTNAME={IP_OF_RASPBERRY_PI}
```

Reboot the Raspberry Pi.

```
sudo reboot
```

To test that your Raspberry Pi is connected to the network properly, use an SSH connection to access your Raspberry Pi from the ROS master. In a terminal on the ROS master device, run the following command.

```
ssh ubuntu@{IP_OF_RASPBERRY_PI}
```

If the ssh is successful, you will be prompted to log into the Raspberry Pi using the password you previously chose.

ROS Noetic Installation & Setup

Run the following commands to install ROS Noetic onto your Raspberry Pi [25].

Sources List and Keys Setup

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main"  
> /etc/apt/sources.list.d/ros-latest.list'
```

```
sudo apt install curl  
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc |  
sudo apt-key add -
```

Installation

```
sudo apt update
```

```
sudo apt install ros-noetic-ros-base
```

Environment Setup

```
source /opt/ros/noetic/setup.bash  
echo 'source /opt/ros/noetic/setup.bash' >> ~/.bashrc  
source ~/.bashrc
```

ROS Dependencies

```
sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator  
python3-wstool build-essential
```

```
sudo apt install python3-rosdep
```

```
sudo rosdep init  
rosdep update
```

Additional Steps

After completing the installation, run the following commands in the terminal.

```
cd  
mkdir catkin_ws && cd catkin_ws  
mkdir src  
catkin_make
```

Once the catkin workspace compiles, run the following command so that the setup.bash script is initiated every time a terminal is opened.

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

Install raspicam_node

To build the raspicam_node from Ubiquity Robotics’s GitHub repository [20], first clone the node.

```
cd ~/catkin_ws/src
git clone https://github.com/UbiquityRobotics/raspicam_node.git
```

Create a new YAML configuration file for some missing dependencies.

```
sudo nano /etc/ros/rosdep/sources.list.d/30-ubiquity.list
```

In the text editor, add the following code to the blank YAML file.

```
yaml https://raw.githubusercontent.com/UbiquityRobotics/rosdep/master/raspberry-pi.yaml
```

Install additional ROS dependencies and then compile the node.

```
rosdep update
cd ~/catkin_ws
rosdep install --from-paths src --ignore-src --rosdistro=$ROS_DISTRO -y
catkin_make
```

Run raspicam_node

Start the ROS master in a terminal on the master device.

```
roscore
```

Depending on the version of your Raspberry Pi Camera, SSH into the Raspberry Pi and run the following command in a new Ubuntu terminal. The below command is an example for a v2 camera with 1280x960 resolution.

```
roslaunch raspicam_node camerav2_1280x960.launch
```

To view the camera output, open a new terminal on the ROS master device and run the following command.

```
rqt_image_view
```

To view your image feed from the Raspberry Pi Camera, select your device from the dropdown menu.

Troubleshooting

If you get an error stating “Failed to create camera component” when attempting to launch the raspicam_node with roslaunch, Ubiquity Robotics advises to check that the camera cable is seated properly on both ends. In case that was not the issue, complete the following steps to manually install the camera_info YAML file and update the config.

Update config.txt

Navigate to the root directory on your Raspberry Pi device. Ensure that you are in the root directory by entering `pwd` into the terminal and verifying that `/` is the output. Then, navigate to the firmware directory.

```
cd /boot/firmware
```

Open `config.txt` in the text editor.

```
sudo nano config.txt
```

At the bottom of `config.txt`, add the following information.

```
[all]

start_x=1
gpu_mem={MEMORY_ALLOCATED_TO_CAMERA_MODULE}
```

Reboot the Raspberry Pi, and then reattempt to launch the `raspicam_node`.

Install camera_info.txt

If you get an error that `camera_info.txt` is missing, navigate to the root directory on your Raspberry Pi device. Then, navigate to the `camera_info` directory.

```
cd /home/ubuntu/.ros/camera_info
```

If any of the directories do not exist, you can create them by entering the following command in the terminal.

```
mkdir {DIRECTORY_NAME}
```

Copy the YAML files from the installation to the new path.

```
cp -r ~/catkin_ws/src/raspicam_node/camera_info /home/ubuntu/.ros/
```

B Raspberry Pi and Flight Controller Configuration

This section provides a basic overview of configuring the PX4 autopilot on an ARKV6X board [16]. We refer the reader to PX4 documentation and community forums for further documentation if needed.

Flight Controller Software Configuration

Flash PX4

This section assumes the user has downloaded QGroundControl. However, any ground station that can flash autopilot software will work.

1. Open QGroundControl, and using the logo in the upper left corner, navigate to “Vehicle Setup.”
2. Select “Firmware” from the menu on the left.
3. Connect the autopilot (in this case the ARKV6X) to your computer.
4. Confirm the autopilot software to download (PX4), the autopilot version, and the make and model of the flight controller.
5. Select “OK” to load the firmware to the board.

Select the Frame Type

Connect the flight controller to QGroundControl. From the “Vehicle Setup” menu, select “Airframe.” For our configuration and frame, choose “HolyBro QAV250” under the “Quadrotor X” category.

Calibrate the Sensors

Select the “Sensors” tab under “Vehicle Setup.” Then,

1. Select “Orientations.” By default, PX4 assumes the flight controller is on the top of the frame, with the x-axis toward the front of the vehicle. If this is not the case, select the applicable rotation.
2. Calibrate the compass, gyro, and accelerometer by selecting each option and following the relevant instructions in QGroundControl.

If needed, the user can disable some or all of the onboard sensors. We choose to disable the compass (due to interference from flying inside) and one UAV’s barometer (due to a loss in functionality). The relevant parameters and values are:

1. Set "CAL_BAROx_PRIO" to 0 to disable all barometer input.
2. Set "CAL_MAGx_PRIO" to 0 to disable all compass input.

Battery & Power Module

Select the "Power" tab under "Vehicle Setup." At this point, enter the relevant characteristics of the connected battery. At minimum, input the number of cells, maximum, and minimum voltage. The maximum voltage should be slightly lower than the actual maximum voltage of the battery, for instance 4.05V per cell if the full capacity is 4.2V.

Motor Configuration

Select the "Actuators" tab under "Vehicle Setup." Specify the coordinates of each motor from the center of mass. This can be an approximate value if needed. Then, select appropriate ESC (in our case the OneShot125 ESCs). Ensure that the motors are assigned to the correct AUX channel. Typical configuration and spin direction are shown in Fig. B.1. To change the location of each motor, the easiest way is to change

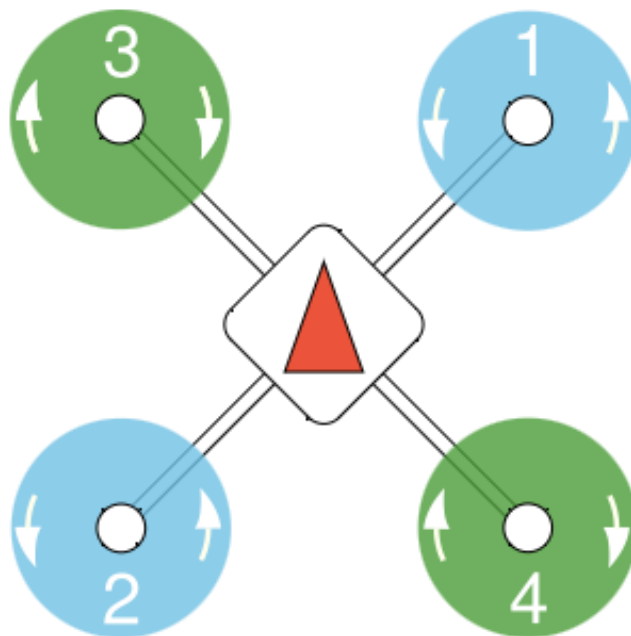


Figure B.1: Proper motor orientation and direction

the AUX input in QGroundControl. However, to change the spin direction, the easiest way is to reverse the positive and negative wires. To ensure that the motor are in the correct places and spinning properly, use the "Actuator Testing" option. First, remove the propellers to avoid injuries or unpredictable behavior. Then, each motor can be tested by moving the respective slider.

ESC Calibration

Once the motor configuration is complete, return to the "Power" tab. Ensure the propellers are not installed. Then, select "Calibrate" under "ESC PWM Minimum and Maximum Calibration." Follow the instruction in QGroundControl.

Joystick Calibration

We choose to use a joystick to control the UAV. To configure this, navigate to "Parameters" under "Vehicle Setup." Set "COM_RC_IN_MODE"=1. Then, pair a controller to the computer. We use an Xbox controller. Once the controller is paired, navigate to "Joystick." Then, perform the stick calibration and assign buttons. Note that only one button should be assigned to each function. Further, important functions include: arm, disarm, emergency stop, and any useful flight modes.

Tuning

PX4 provides two options for tuning. A user may opt for either automatic tuning or manual tuning. PX4 recommends autotuning, but this does not work for us. For future teams, we would recommend following PX4 autotune documentation before resorting to manual tuning.

Log Files

QGroundControl provides method to view logged data from the flight controller. Under "Analyze Tools" the user can navigate to "Log Download" to download the .bin files or to "MAVLink Inspector" to view real-time data. We use this functionality to diagnose issues and to gather data during wind tunnel testing.

MAVLink Communication

The default port for MAVLink communication is TELEM2. We configure the following parameters on the flight controller to use the TELEM2 port for MAVLink communication to a Raspberry Pi, which runs mavros.

1. Set "MAV_1_CONFIG" to 102 (TELEM2).
2. Set "MAV_1_RATE" to the appropriate baud rate (recommended: 921600).
3. Set "SER_TEL2_BAUD" to the appropriate baud rate (must be the same as "MAV_1_RATE").

VICON Input

PX4 recommends using "VISION_POSITION_ESTIMATE" for motion capture localization. This is because the EKF2 subscribes to this topic, but does not subscribe to the motion specific topics. As a result, we recommend using the corresponding ROS topics to publish VICON position data to the UAV. A few parameters must be set on the UAV to allow the EKF2 to take in these inputs.

1. Set "EKF2_EV_CTRL" horizontal, vertical, velocity, and yaw fusion according to model.
2. Set "EKF2_HGT_REF" to Vision to use VICON for height estimation.
3. Set "EKF2_EV_DELAY" to the network delay in publishing and receiving VICON position data.
4. Set "EKF2_EV_POS_X", "EKF2_EV_POS_Y", and "EKF2_EV_POS_Z" to the offset between the VICON and the UAV's frame of reference.

After setting these parameters, reboot the flight controller.

Raspberry Pi Configuration

mavros Installation

Assuming that ROS is installed (see Appendix A), run the following commands in terminal to install mavros [26].

```
sudo apt-get install ros-noetic-mavros ros-noetic-mavros-extras
wget https://raw.githubusercontent.com/mavlink/mavros/master/mavros/scripts/
  install_geographiclib_datasets.sh
chmod a+x install_geographiclib_datasets.sh
./install_geographiclib_datasets.sh
```

Launch File

On the Raspberry Pi, create and open the launch file in `~/catkin_ws/src`.

```
touch px4.launch
nano px4.launch
```

Copy and paste the following code into the new file. This launch file assumed a baud rate of 921,600 and that the flight controller is connected to the Raspberry Pi on pins 8 and 10. Note that the baud rate must match the baud rate on the flight controller.

```
<launch>
  <!-- vim: set ft=xml noet : -->

  <arg name="fcu_url" default="/dev/ttyAMA0:921600" />
  <arg name="gcs_url" default="" />
  <arg name="tgt_system" default="1" />
  <arg name="tgt_component" default="1" />
  <arg name="log_output" default="screen" />
  <arg name="fcu_protocol" default="v2.0" />
```

```

<arg name="respawn_mavros" default="false" />

<include file="$(find mavros)/launch/node.launch">
  <arg name="pluginlists_yaml" value="$(find mavros)/launch/
px4_pluginlists.yaml" />
  <arg name="config_yaml" value="$(find mavros)/launch/
px4_config.yaml" />

  <arg name="fcu_url" value="$(arg fcu_url)" />
  <arg name="gcs_url" value="$(arg gcs_url)" />
  <arg name="tgt_system" value="$(arg tgt_system)" />
  <arg name="tgt_component" value="$(arg tgt_component)" />
  <arg name="log_output" value="$(arg log_output)" />
  <arg name="fcu_protocol" value="$(arg fcu_protocol)" />
  <arg name="respawn_mavros" value="$(arg respawn_mavros)" />
</include>
</launch>

```

Start the ROS master in terminal on the master device.

```
roscore
```

Launch mavros on the Raspberry Pi.

```
roslaunch px4.launch
```

Troubleshooting

If mavros crashes due to permission issues, assign the current user as the owner of the relevant serial port.

```
sudo chown {user} /dev/ttyAMA0
```

Replace {user} with the name of relevant profile on the Raspberry Pi.

If the connection is crashing, check the baud rate of the port. It should be greater than or equal to the desired baud rate.

```
stty -F /dev/ttyAMA0
```

If the flight controller is not connecting to mavros, ensure that /dev/ttyAMA0 is properly assigned. Open the config file.

```
sudo nano /boot/firmware/config.txt
```

Append the following lines the end of the file.

```
enable_uart=1
dtoverlay=disable-bt
```

By default on the Raspberry Pi 3, this port is assigned to Bluetooth and `/dev/ttyS0` is the port assigned to pins 8 and 10. However, `/dev/ttyS0` is the miniUART and is less powerful, so this is not desired.

If the baud rate is not high enough on the Raspberry Pi, try to change the clock of the port by appending the following to the end of the `/boot/firmware/config.txt` file.

```
init_uart_clock=16*{desired baud rate}
```

If mavros prints "RT too high for timesync" increase the baud rate on the Raspberry Pi and the flight controller.