UNIVERSITY OF CALIFORNIA
SANTA CRUZ

AN OPEN SOURCE REAL-TIME CONTROLLER FOR RESOURCE-CONSTRAINED AUTONOMOUS VEHICLES AND SYSTEMS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Aaron Hunter

March 2023

The Dissertation of Aaron Hunter
is approved:

Gabriel Elkaim, Chair

Patrick Mantey

Renwick Curry

James Whitehead

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

An Open Source Real-time Controller for Resource-constrained Autonomous Vehicles and Systems

by

Aaron Hunter

The use of autonomous systems is burgeoning in the world for applications in many fields from scientific, industrial, to military. At the same time, advances in semiconductor technology have enabled ever smaller, complex, and use-specific microprocessors and microcontrollers. This work details the design and implementation of an open source real-time hardware controller for resource-constrained autonomous vehicles and systems. It is intended to be integrated inside a distributed control architecture consisting of the real-time hardware controller, a guidance and navigation computer, and an edge tensor processing unit for machine learning inferences. While the latter two processors are commercially available, a dedicated, modular real-time controller is not, providing the motivation for this work. To demonstrate the versatility of our open source real-time controller we present several use cases including a ground vehicle, marine vessel, quadcopter, and fixed-wing aircraft. The power of the distributed architecture is the ability to solve complex sensing, guidance, navigation, and control challenges even in resource-constrained systems. One such challenge is the simultaneous localization of an autonomous system while mapping an environment. In this work we develop the components of a novel hybrid sensor combining a visual camera and LiDAR sensor that is mounted on the ground vehicle. This sensor is trained to recognize landmarks in the environment using object detection frameworks and deployed on the edge tensor processing unit. At the same time, the LiDAR sensor provides range and bearing information for objects within its field of view. By combining the two we can get fast detections of arbitrary landmarks in the environment as well as determine their position relative to the sensor, thus enabling simultaneous localization and mapping functionality.

To my father,

Stephen Hunter,

for providing me encouragement when I needed it the most.

Acknowledgments

I would like to recognize my fellow lab members, in particular, Carlos Espinosa and Pavlo Vlastos, for the excellent conversations and the mutual support.

Thanks to Ren Curry for his insight and his excellent editing skills and to Gabe Elkaim for initiating this project.

I would like to acknowledge the Center for Research in Open Source Software for their support of this work.

Chapter 1

Introduction

1.1 Motivation

The civilian use of autonomous craft for scientific as well as commercial purposes has grown significantly in the last few years. A 2018 report from the National Oceanic and Atmospheric Administration detailed the use of un-crewed systems, UxS[1], for the agency [26]. The number of types of un-crewed aerial vehicles (UAVs) employed by the agency doubled in a one year period and the number of total vehicles deployed increased by 38% for the same year. Use of other UxS by the agency show a similar trend. In the same report they detail the use of UxS from space (in the form of satellites) to the ocean floor (remotely operated vehicles) and nearly every environment in between.

UxS are also used to monitor dangerous environments. A New York Times article from 2021 [38] demonstrated the use of a novel autonomous surface vessel (ASV) that collected video imagery from within the eye of Hurricane Sam. There are examples in the literature of the use of UAVs to monitor wildfires [33] and indeed commercial products exist for that purpose [30].

UxS are in use in commercial sectors as well. In agriculture, for example, many types of systems exist for crop health monitoring [1], water use monitoring [16], crop phenotyping [45], and even autonomous weeding for organic crop production [5]. Other

[1]The 'x' in UxS indicates multiple vehicle types are in use, e.g., aerial, surface, and ground vehicles.

commercial applications exist in filmmaking, sports broadcasting, real estate sales, house cleaning, personal use, etc.

UxS are even deployed on other planets. NASA currently has an autonomous ground vehicle (AGV) on Mars, the "Perseverance," but also a UAV specifically designed for the thin atmosphere of the planet [24].

Clearly the use of these systems is widespread and growing. At the same time the autonomy requirements for the systems are growing more challenging. The ASV cited in the New York Times article, for example, can be deployed on missions up to one year in duration and the Mars Rover mission duration is likely longer. The implication for these longer duration missions is that vehicles must have onboard path finding capabilities, automated data capture and sample analysis, failure detection algorithms, data storage, and communication abilities to name just a few.

Processing power requirements for autonomous systems are increasing as well, driven by the need for computationally expensive operations such as onboard computer vision, LiDAR sensor processing, and for evaluating machine learning (ML) models for object detection, image classification, and natural language processing. Longer duration missions and navigating complex environments place additional demands on the onboard processors. Finally, interacting in multi-agent environments, that is, environments with multiple autonomous systems adds dramatically to the complexity of the control system.

Designing a modern autonomous system involves significant engineering effort and requires expertise in multiple areas. The resulting system may involve complex interactions with many specialized processing systems.

1.2 Problem Statement

Given the widespread interest and use of autonomous systems there exists a need for a vehicle-agnostic controller—or autopilot—to enable the research and development efforts for new vehicles and new vehicle types. This autopilot should be capable of real-

time computation (that is, temporally deterministic); easily modifiable in order to adapt to different vehicle configurations; and open-sourced (both firmware and hardware) in order to be accessible to a wide audience of users and contributors.

Furthermore, the autopilot may form the real-time core of a larger, distributed control system that also includes a single board computer (SBC) for non-real-time tasks and a tensor processing unit (TPU) to enable onboard ML capabilities. This architecture allows a developer to take advantage of the advancements in computing for SBCs and TPUs as well as to solve the issue of handling real-time tasks and non-real-time tasks simultaneously. To enable a distributed control system the autopilot must integrate with the external modules using a standard interface and a lightweight, extensible binary communication protocol.

This architecture offers unique capabilities for autonomous vehicles. There are several sub-problems that can be specifically addressed with this architecture. The first is the means to develop and integrate real-time algorithms such as state estimation and vehicle control onto the autopilot. The embedded firmware operates using the simplest structure possible, with a hardware timer to dictate sensor and control update intervals. This structure allows an algorithm or sensor measurement to be encapsulated into a single function (with a standard input and output) and therefore replaced or modified with minimal disruption to the remainder of the firmware. The controller can be adapted to any type of vehicle with relative ease, for example, or to evaluate different state estimation algorithms by changing one function and recompiling the firmware.

Second, the SBC enables new capabilities for autonomous vehicles. Typically, the SBC has a Linux operating system (OS) providing access to standard functionality such as file storage, internet access, camera integration, USB devices, and running WiFi access points. These capabilities allow storage of sensor and vehicle state information into files from the vehicle, run mission planning software, or connect to USB peripherals such as the TPU. The OS also opens up extensive capabilities provided by open source software. For example, scientific software such as SciPy, computer vision software (e.g., OpenCV), ML packages (e.g., Tensor Flow), or even convex optimization

software (e.g., CVXPY) are all readily available to download onto the SBC. These advanced computational packages enable capabilities such as optimal trajectory generation, mapping of landmarks, and advanced vision sensors.

Finally, the TPU is a specialized processor that is designed to speed up ML inferencing tasks. The increased use of machine vision has made the need for onboard ML capabilities an attractive feature for many types of autonomous systems. It enables fast object detection or image classification without placing a computational burden on the SBC. For example, object detection allows for identification of landmarks or obstacles. The TPU integrates seamlessly with the SBC via a USB interface and as in the case of the SBC runs open source software if possible.

In summary, the increasing demands on autonomous systems and the technological advances in SBCs, TPUs, and sensing technology, suggest that a modular, distributed control system that includes a dedicated real-time autopilot is needed for the development of new autonomous vehicle classes. Small, resource-constrained systems benefit the most from this architecture and motivate this work.

1.3 Related Work

Given the proliferation of autonomous systems it is somewhat surprising that real-time controllers are not more ubiquitous, few of the ones that do exist use open source firmware, and fewer still have open source hardware.

The most promising one is a vehicle-agnostic control system: the modular rapid prototyping system, R2P [4]. The project hardware and software are both open source. The system is designed so that each sensor or actuator is a standalone module called a node. The authors have developed a publisher-subscriber middleware to connect the nodes, a protocol called RTCAN. A series of connected nodes form the robot architecture. One downside to this approach is that the developer is restricted to the modules that have been developed for the system, that is, the system doesn't have the ability to communicate directly with off-the-shelf sensors, nor can they easily be created by the

user. Furthermore, each module is separately controlled with its own microcontroller and has an RJ45 wired connector so a complex system rapidly becomes unwieldy with many small boards wired together. Another downside for optimal real-time performance is the use of a real-time operating system (RTOS), specifically the ChibiOS/RT [7]. This is a powerful, lightweight and widely-used RTOS with a hardware abstraction layer (HAL) that allows for re-use of peripheral drivers across different hardware platforms. However, modifying the firmware of a given module for a custom algorithm may be difficult to debug for real-time performance. For example, the inertial measurement unit (IMU) module has its own attitude estimation algorithm encoded in it. If a developer needs a different attitude estimation algorithm it may be challenging to implement it and still guarantee the latency needed for the application. Nevertheless this system allows robot developers a path to quickly build and test a prototype robot, particularly if autonomy isn't required and size of the control system is not a factor.

An open source autonomous vehicle platform is found in the literature, the F1/10 platform [27]. Although the project labels itself as an autonomous cyber-physical system platform, it is a 1/10 scale race car, employing an NVIDIA Jetson TX2 SBC running the Robot Operating System (ROS) middleware. While this is a powerful SBC and ROS is a widely adopted middleware for robots, it is not real-time. In fact, the recent launch of ROS2 in part is due to the need for lower latency operations—a closer approximation to real-time, although it isn't strictly real-time as it continues to operate on the Linux OS. In any case, this project is dedicated to a specific racing platform and therefore not easily adapted for other purposes.

Aerial vehicles provide the richest source for real-time controllers, some of which have been adapted to different classes of vehicles. Table 1.1, adapted from a recent survey of open source UAV hardware controllers[11] list the ones active as recently as 2018. Most of the platforms listed that use the STM microcontroller are designed to operate with an RTOS as the middleware, with the autopilot firmware loaded over it. The exceptions to this is the Chimera, which uses the Paparazzi autopilot firmware

which has both an RTOS based firmware as well as a 'bare metal' implementation[2]. The most prevalent autopilot firmware packages are PX4, Ardupilot, and LibrePilot (formerly OpenPilot). The other devices in the table, FlyMaple, APM2.8, Erle-Brain and PXFmini are no longer available.

Both PX4 and Ardupilot support non-aerial vehicles to some extent. However, modifying the firmware within the RTOS is notoriously challenging due to the complexity of the code base. It is particularly difficult to implement a new vehicle type but also challenging to guarantee that latency requirements are met when modifying the estimation or control algorithms. These firmware packages are better suited to projects where modifying any of the underlying algorithms isn't desired.

The topic of an RTOS versus a bare metal application merits further discussion and will be covered in a later chapter.

Platform	MCU	Sensors	License	Interfaces
Pixhawk	STM32F427	b, m	BSD	c, s, a, pp, sb, ds
Pixhawk 2	STM32F427	b, m	CC-BYSA-3.0	c, s, a, pp, sb, ds
PixRacer	STM32F427	b, m	CC-BY 4.0	c, pp, sb, ds
Pixhawk 3 Pro	STM32F427	b, m	CC BY 4.0	c, s, pp, sb, ds
PX4 FMUv5 and v6	STM32F427	b, m	CC BY 4.0	c, s, a, pp, sb, ds
Sparky2	STM32F405	b, m	CC BY-NC-SA 4.0	c, pp, sb, ds, da
Chimera	STM32F767	b, m, p	GPLv2	c, s, a, da, pp, sb, ds, x, au
CC3D	STM32F103	None	GPLv3	pp, ds, sb
Atom	STM32F103	None	GPLv3	pp, ds, sb
APM 2.8	ATmega2560	b	GPLv3	pp, a
FlyMaple	STM32F103	b, m	GPLv3	None
Erle-Brain 3	Raspberry Pi	b, m	CC BY-NC-SA	a
PXFmini	Raspberry Pi	b, m	CC BY-NC-SA	a

Table 1.1: Comparison of open source hardware UAV controllers from a recent survey [11]. Abbreviations are b: barometer, m: magnetometer, p: pitot tube sensor, c: CAN, s: SPI, a: ADC, pp: PPM, sb: S.BUS, ds: DSM, da: DAC, x: XBEE, au: AUX.

[2]Bare metal refers to programming the microcontroller with no underlying operating system.

There are many papers on self-driving cars that describe vehicle control systems but very few discuss ones suitable for resource-constrained vehicles. [17], for example, demonstrates a distributed architecture of a full size autonomous racecar that requires computational resources far beyond those available to resource-constrained systems. One exception to that, however, comes from [3], which reviews four different architectures with a focus on how resource-constrained systems can inform design choices for full size vehicles. All the architectures they review are modular with the low level (i.e., real-time) control functions deployed on a microcontroller and higher level tasks performed by various systems—either ROS or a custom-designed middleware. The real-time control in two cases use a HAL, the others do not.

Within the Autonomous Systems Laboratory (ASL) there is an important related work called the Santa Cruz Low-Cost UAV GNC Subsystem (SLUGS)[21], which was an early autopilot intended for UAVs. It was programmed using the embedded code generation capability of Matlab and Simulink. This project, though successful, was never widely adopted. One reason is that Matlab and Simulink are expensive commercial software packages and come with a significant learning curve. Another is that the project wasn't open source limiting access to vehicle developers or potential contributors. These limitations motivate this work which is open source, both hardware and firmware.

1.4 Contributions

The contributions described in this work are summarized in the following list:

- An open source design of a real-time autopilot that is vehicle agnostic, that is, easily adaptable to many different types of craft. It fits within a modular system architecture of our design suitable for resource-constrained autonomous systems along with open source libraries for many common functions.

- An open source repository including hardware design files, microcontroller initialization code, sensor and actuator libraries for common sensors and outputs,

attitude estimation and other navigation algorithms, and vehicle control algorithms.

- A benchmark algorithm to evaluate real-time hardware performance and an evaluation of the performance of four different processors (three microcontrollers and one SBC) using the benchmark.
- A ground vehicle platform suitable for demonstration of the controller capabilities presented in detail here. Three additional use cases using the real-time controller within the distributed architecture that are either completed or under development.
- A custom object detection algorithm trained to identify landmarks in the environment and deployed to a TPU.
- A sensor module consisting of a camera, a lightweight LiDAR, and a panning servo designed to be used in conjunction with the object detection algorithm for mapping landmarks.

1.5 Dissertation Organization

This dissertation is organized as follows. Chapter 2 is a review of the requirements for the real-time controller. Chapter 3 details the hardware development process, architecture, electrical design, and the final controller. Chapter 4 details the software design, architecture, and development process. In Chapter 5 we propose a benchmark algorithm to quantify the performance of the real-time controllers. Chapter 6 shows the results of using the benchmark on four different processors, three real-time systems using microcontrollers and one SBC running Debian Linux. Chapter 7 details the development of an AGV from the chassis to autonomous navigation used as the main test platform for the controller. Chapter 8 covers three new cases for different autonomous systems (a USV, a quadcopter, and a fixed-wing aircraft) that use this architecture or are

in the process of deploying it. Chapter 9 demonstrates several areas where this work can be extended. Finally, Chapter 10 provides the conclusions.

There are two supplementary appendices.Appendix A is a derivation of the complementary filter, the basic algorithm behind the benchmark. Appendix B details the application of the autoregression with external inputs method for system identification using a simple motor model as an example.

Chapter 2

Requirements

In this chapter we discuss the requirements for a real-time, vehicle agnostic autopilot, starting with the intended use case, a taxonomy of real-time tasks, hardware and peripherals, interface requirements, and finally firmware.

2.1 Intended Use

The primary motivation for this work is to provide a means for an autonomous vehicle developer to rapidly prototype the real-time guidance, navigation and control system of a new vehicle. To enable this we decided at the inception of the project to make the design and firmware open source and named it the Open Source Autonomous Vehicle Controller, or OSAVC for short. The advantages provided by open source methodologies to a would-be vehicle developer are that the designs and algorithms are freely accessible, they are easily modifiable (e.g., for a custom application), and can be sourced from numerous vendors. From the project side open sourcing allows contributors from all over the globe to add new functionality, to test and provide feedback, and iterate on the designs. Indeed, during the development of this project we were able to participate in the Google Summer of Code program which hires student interns to contribute to open source projects. Over the course of two summer sessions, the OSAVC project (under the guidance of the Center for Research in Open Source Software)

received over 45 applications from students all over the globe, and was able to provide six paid internships. Additionally, the OSAVC prototype was used by two other graduate researchers for an ASV and a quadcopter which was an early demonstration of the capability and modularity of the project.

2.2 Real-time Task Taxonomy

The OSAVC provides real-time control of the vehicle actuators and measurement of the sensors. In the context of control systems, real-time refers to a fixed output period of the controller, i.e., the period between updating the vehicle actuators (or other algorithms such as attitude estimation) is deterministic and constant. All modern control systems are digital, that is they do not provide continuous outputs, instead they update the outputs on a fixed clock cycle. The systems under control, however, exist in the continuous real world. If the controller clock cycle, τ_c, is short compared to the dynamics of the vehicle, then the digital controller approximates a continuous controller. In other words, if the rise time[1] of a system in response to a step change in input is τ_v, then the digital control system approximates the continuous case when $\tau_c << \tau_v$. In the frequency domain a usual figure of merit is that the update rate is 1/20th-1/30th the bandwidth of the system [12]. A finite clock cycle introduces a lag in the response of the controlled system. In addition to having as small a lag as is practical, it is also important to minimize the variation in the clock cycle. Small variations are inevitable but larger variations can lead to poor control or, in the worst case, instability.

Not all tasks in an autonomous system require real-time control. For example, logging of telemetry data for post-mission use does not require a fixed clock—the system can buffer the data until it is convenient to store it. Thus, when designing a distributed control system it is natural to divide tasks into real-time and non-real-time categories. Real-time tasks are the domain of the real-time controller and all others belong to a processor with an OS. Note that some control systems attempt to handle all tasks within

[1]Rise time is defined as the time for a system to change from 10% to 90% of its steady state value after a step change in input.

the same processor by using an RTOS. We discuss the tradeoffs of this type of OS in Section 2.6.1.

Strictly real-time tasks are ones that support the control of the vehicle. These comprise the measurement of all the inputs and computation of the algorithms necessary to provide the actuator output. These include any sensor that is used to estimate the vehicle's state (typically position, velocity, and attitude). A non-comprehensive list of these sensors include wheel encoders, airspeed sensors, GPS, gyroscopes, accelerometers, magnetometers, barometers, etc. Accordingly, the navigation tasks, which include state estimation, are also real-time as they provide the input to the controller. Although mapping of the environment isn't a strictly real-time task, simultaneous localization and mapping (SLAM) [9] is, as it provides vehicle state information (namely position and orientation). Communications generally do not have real-time requirements with the exception of remote control, when the system is operated manually using a radio. Finally, the controller itself must operate on a fixed cycle.

All other tasks are non-real-time. Note that this doesn't imply that they are not time-sensitive. For example, a range sensor might be used for obstacle avoidance, or a camera may be used to identify landmarks in the environment. In both cases, these data are needed to support guidance of the vehicle. Guidance is the process of determining the vehicle trajectory and therefore must be done quickly, but not necessarily on a fixed clock. Generally, guidance tasks (including mapping) are not real-time but are typically time sensitive. Other tasks, such as logging of data, measurement of sensors not used for state estimation, and general communications (e.g., communicating to a ground control station) are non-real-time.

A generic taxonomy of the tasks of an autonomous system is summarized in Table 2.1 showing the classification of real-time versus non-real-time as discussed above.

Real-time	Non-real-time
Sensors (state measurement)	Sensors (all others)
Communication (remote control)	Communications (all others)
Navigation	Guidance
SLAM	Mapping
Control	File system tasks
	Data logging
	User interface

Table 2.1: A control system taxonomy of tasks required in an autonomous system.

2.3 Overall Requirements

From a high level, the OSAVC should support the main hardware and communication interfaces used by commercially-available components. The components needed for a vehicle will include sensors, radios (both for remote control and general communication, e.g, with a ground control station), and actuators (e.g., motors, solenoids, indicators, etc.). Additionally, as a real-time subsystem, the OSAVC module should communicate efficiently and quickly to the larger vehicle control system. For example, a vehicle may have an onboard guidance and navigation SBC that requires periodic communication of the vehicle state variables in order to plan a route. The module should support the real-time requirements of latency and speed for a vehicle. That is, it should be able to operate as quickly as needed by the vehicle and meet the deterministic latency requirement for the control algorithms. A final requirement is that the OSAVC provide power, signal conditioning, and pwer management for the sensors and processor, and optionally for any external peripherals or modules (e.g., the SBC).

2.4 Peripherals

The most common hardware interfaces are the inter-integrated circuit (I2C) interface, the serial peripheral interface (SPI), the universal serial bus (USB), and the universal asynchronous receiver-transmitter (UART)—commonly known as a serial port. Less common but used extensively in automobiles and robotics is the controller area network bus (CAN bus).

Other peripherals needed for the OSAVC are hardware timers, input capture modules (IC)—used for decoding incoming digital signals, and output compare modules (OC) used for generating pulse-width modulated (PWM) signals needed for motor and servo actuator control. Finally, general purpose input-output pins (GPIO) are useful for turning on LEDs, setting switches, and other requirements.

The OSAVC should have most or all of these hardware peripherals in order to support the broadest range of applications; while tradeoffs can be made with respect to capability versus size, weight, and power, we typically fall on the side of greater capability in this work.

2.5 Hardware

2.5.1 Printed Circuit Board

The printed circuit board (PCB) requirements are that it be as small and light as practical while also allowing for hand assembly—as many users may not have reflow ovens to solder the components. These requirements therefore drive the use of surface-mount components to minimize the size of the assembly but limits some component types which are difficult to solder by hand (e.g., ball grid arrays). The PCB is limited to four layers to minimize cost.

2.5.2 Connectors

Many vehicles experience high levels of vibration during operation. It is therefore imperative that the connectors used be robust to vibration while minimizing board area use. They must also be easily sourced, not too expensive, and capable of hand assembly.

2.6 Firmware

The firmware is written entirely in the C programming language for speed and efficiency. The most important requirement of the firmware is modularity. Each sensor driver, estimation algorithm, and control algorithm is modular. Each module has a header file which provides the interfaces to public methods and a source file containing the code; each source file has a built-in test harness to allow for module troubleshooting outside of an application.

To maximize efficient utilization of the processor each sensor module is non-blocking. This requirement forces the use of vectored interrupts and each sensor has a unique interrupt priority assigned to it.

The firmware library contains a selection of commonly used sensors needed for vehicle autonomy both to provide a minimum of functionality for a new developer as well as a template for new sensor drivers.

The application itself should be as simple as possible to achieve the desired control. An example application for an AGV will be provided as a template for other applications. The application is a bare metal application consisting of an infinite loop with a hardware timer for task scheduling. This avoids the complexity and overhead of an RTOS. This requirement merit some discussion and is covered in Section 2.6.1.

2.6.1 RTOS vs bare metal

An RTOS is a piece of software that brings in real-time and non-real-time tasks into the same controller. It uses a HAL that allows for programmers with no expertise

in embedded programming to communicate with various sensors without any understanding of the underlying hardware. It also provides for standard OS features like data logging and directory structures. The difficulty in using an RTOS lies in modifying the real-time tasks. Even with well-structured code, debugging the latency of a new real-time algorithm can be challenging. Autopilot firmware running on an RTOS typically implements the algorithms with tuneable parameters (e.g., the extended Kalman filter in Ardupilot). This may work well for many cases, but modifying an algorithm for faster operation or for different parameters is challenging. Developing a new algorithm is even more difficult. Of course any OS increases the complexity of an application as well using more computational resources.

A bare metal application, on the other hand, allows for strict control over the sensing and control elements of an autonomous system. Measuring the latency is straightforward using a hardware timer of the processor, as is managing processor bandwidth. Debugging the function is similarly simplified. The challenge in programming on bare metal is the need for an understanding of the hardware (sensors, peripherals, and processor performance) in order to configure it properly, although device manufacturers often offer configuration apps or software development kits (SDK) to ease this process. Another benefit of bare metal programming is efficiency due to the much lower overhead. The main downside for bare metal applications is the challenge in duplicating the common features of a standard OS. For this reason, a bare metal application is suited better for a distributed architecture where the real-time and non-real-time tasks are strictly segregated between the real-time core and the SBC as discussed earlier.

Chapter 3

Hardware Design, Assembly and Testing

3.1 Development Process

The design of the OSAVC PCB consisted of two development efforts. As we had no prior experience designing a PCB we decided to start with a 'daughter board'—a simpler I/O PCB that could interface with a commercially available development board[1]. The first daughter board was hand-wired to determine an initial layout and to kickstart the development process. This allowed us to develop PCB layout and assembly skills and at the same time acted as a platform to test sensor driver modules. The I/O board—a simple two layer PCB—required two revisions before its performance was satisfactory.

We also selected a vehicle platform for the AGV test bed on which to validate OSAVC. To complement the AGV we purchased a suite of sensors that are widely used for small autonomous vehicles and proceeded to develop a library of sensor drivers and integrated the sensors onto the vehicle. The discussion of the software design is found in Chapter 4. Once a complete set of sensors were developed and tested, we implemented a basic remote control application to verify the functionality of the Max32 as well as the AGV platform. Details on the AGV are presented in Chapter 7.

[1]The chipKit Max32 by Digilent, unfortunately no longer offered for sale.

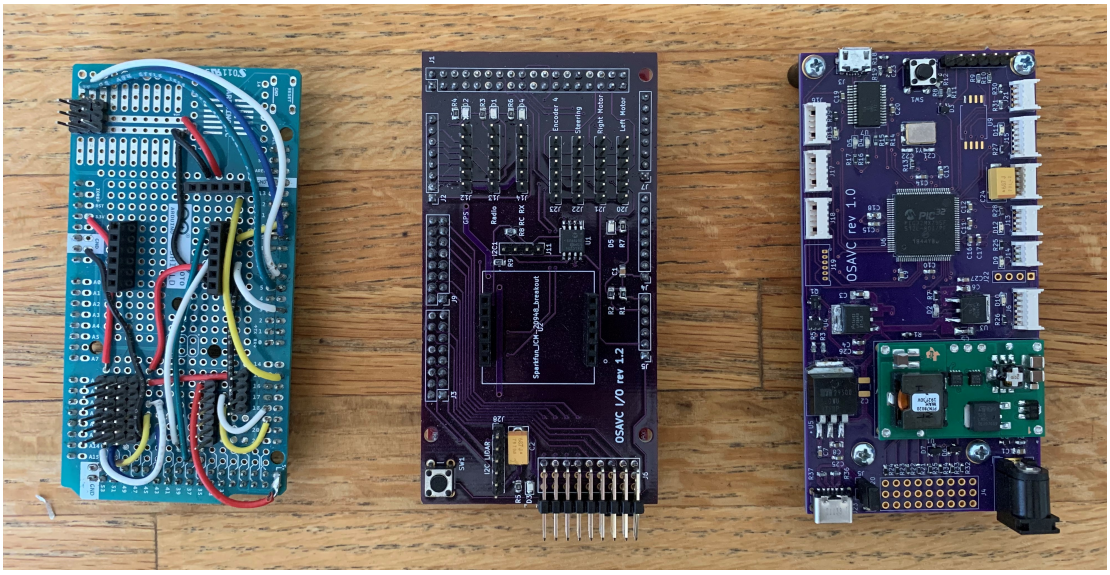

Figure 3.1: The complete evolution of the OSAVC PCB designs. The board on the left is the original hand-wired prototype daughter board. The middle image shows the final design of the I/O daughter board. The image on the right hand side is the first version of the OSAVC.

The second phase of development was the design of the OSAVC PCB itself. This was a more difficult effort beginning with the electrical design, followed by the PCB layout, unit testing of the assembly, and integration with the vehicle. A single revision of the design provided a working PCB. Fig. 3.1 shows the evolution of the designs, beginning with the first hand-wired daughter board, the completed daughter board PCB, and the initial OSAVC design.

3.2 Architecture

We designed the OSAVC with a few goals in mind. The most important of which being that the controller be vehicle agnostic—that is, adaptable to a large class of vehicle types. This translates into the hardware supporting as wide a variety of interface and output protocols as practical. Fig. 3.2 shows a more detailed view of the controller. The firmware architecture was designed to be as simple and as temporally deterministic as

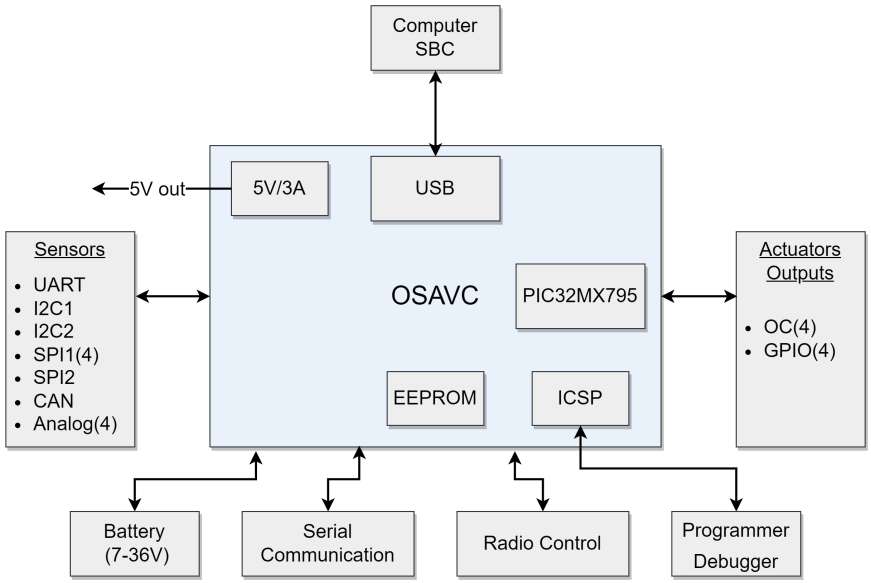

Figure 3.2: Block diagram of the OSAVC. The heart of the OSAVC is the PIC32MX795 microcontroller which controls all the I/O and the hardware peripherals. The OSAVC accepts a wide variety of interface protocols for sensing and many output protocols for vehicle control. It includes a USB interface for debugging with an external computer or for use with an SBC. It has a separate 5V power supply to provide power to external components. It also has capability of sending sensor and state information to a remote ground station and accepts remote control manual input.

possible. To this end the dynamic control algorithm and the state estimation algorithms are modular. This allows these algorithms to be designed and tested in control-specific software (e.g., Matlab and Simulink) and ported out as C functions which can then be compiled with the vehicle firmware directly[2]. We demonstrate this approach in Chapter 5. We also required the sensor drivers to be non-blocking and the control loop to operate using a hardware timer to minimize timing jitter. The last goal is to enable as large a community as possible. To support this goal the hardware design and firmware are open source and hosted online [28].

[2]Note that this can be done by hand or by the automatic C-code generation capability of the Matlab embedded coder.

3.3 Sensors, Peripherals and Connectors

The sensors we selected are ones typically used in resource-constrained autonomous vehicles. Possibly the most common one is the inertial measurement unit, or IMU. This sensor consists of three triaxial sensors in a single package: gyroscopes to measure angular rotation rates, accelerometers to measure specific force, and magnetometers to measure the earth's magnetic field strength. The gyroscopes are used to compute the dynamics of the vehicle, the accelerometers and magnetometers are used to determine the vehicles attitude relative to known inertial vectors—gravity and the magnetic field.

Almost as ubiquitous as the IMU is the GPS (global positioning system) receiver. This sensor calculates the vehicle's location in global coordinates by measuring the range of the sensor to four or more satellites at known orbital locations and computing the optimal fit of those ranges. GPS sensors can also provide other useful data, such as the current time, vehicle heading and speed in inertial coordinates.

Another commonly used sensor is an angular encoder. This device measures the angle of rotation around an axis. In the AGV it is used to calculate the speed of the motors as well as the angle of the steering servo. In other areas of robotics they measure joint or actuator angles.

The microcontroller itself has built in analog to digital converters (ADCs). Some sensors communicate their output using analog signals. A typical application for analog signals is battery voltage monitoring.

The last sensor we selected is a LiDAR sensor (light detection and ranging). LiDAR is commonly used to detect local obstacles in the environment. It operates by measuring the time of flight of a laser beam on a round trip path from the vehicle to an obstacle.

In addition to the sensors decribed above, there are several peripherals commonly needed for autonomous vehicles. Electronic speed controllers (ESCs) are used to provide control signals to brushless DC motors (BLDCs). ESCs require a specific form of PWM signals to command the motor velocity. Servo motors use the same PWM signal but instead control the angle of a motor and are used for various types of actuators, such as robot arms or steering systems. Finally, it is useful to have onboard data storage for

vehicle parameter information. A common inexpensive (but low capacity) non-volatile memory device is the EEPROM (electrically erasable programmable read-only memory), typically available in the Kbit range.

Many autonomous systems rely on remote manual control through serial radio receivers. Other forms of radio control are possible but most modern systems utilize serial data streams. Other radios are used to communicate from the vehicle to ground control systems. These radios typically use a serial protocol. Another form of communication is supported through a serial to USB converter. This can be used as a debug port or to transmit data to an external computer. Finally, the microprocessor is programmed using the Microchip PICkit 3/4 serial programmer. A summary of the basic sensors and peripherals supported by the OSAVC and their respective hardware interfaces is found in Table 3.1.

Every sensor has a physical connector and there are nearly as many connector types as there are sensor types. After a review of the most common connectors we selected the Molex Picoblade for the OSAVC. It is a good compromise between footprint and mechanical reliability. It has 1mm pitch between conductors to minimize size and uses through hole pins for connection to the PCB, making it more reliable than surface mount equivalents, and easier to assemble.

3.4 Electrical

Because this research is primarily open source, the tools used for development need to be open source as well; we used the KiCad electrical design automation suite[3] for the electrical and PCB layout design.

The first design decision before embarking on the electrical design was the choice of microcontroller. we selected a microcontroller that satisfied the requirements listed in the Chapter 2, the Microchip PIC32MX795 series microcontroller, a 32 bit MIPS-based processor. This device includes a wide suite of hardware peripherals including

[3] https://www.kicad.org/

Device	Model	Interface	Note
IMU	TDK ICM20948	SPI1	
GPS	u-blox NEO M8N	UART2	
Encoder	AS 5047D	SPI2	12 bit, up to four devices
Battery voltage	N/A	AN0	Scaled 1:8
GPIO	N/A	AN1-4/RB2-5	Analog or digital
LiDAR	Garmin V3HP	I2C2	
ESC	generic	OC2-5	Bidirectional/unidirectional
Servo	generic	OC2-5	Shared with ESC
RC receiver	FrSky serial	UART5	SBUS protocol
Radio	MRO	UART4	915 MHz, ASCII
EEPROM	Microchip 25LC256	I2C1	256 Kbit
USB	FTDI 2232RL	UART1	Serial-USB converter
Programmer	Microchip PICkit3/4	ICSP	In-circuit serial programmer
User I2C	N/A	I2C1	Connector provided
User CAN	N/A	CAN1	For external transceiver

Table 3.1: List of devices (sensors and peripherals) supported by the OSAVC.

numerous UARTs, SPI ports, I2C ports, IC blocks, OC blocks, and timers. It allows for CAN bus integration with the inclusion of an external CAN transceiver. It operates at a reasonably fast 80 MHz clock frequency, has 128 Kb SRAM and 512 Kb flash memory, is a mature design, and is well supported by documentation and a functional integrated development environment (IDE) called MPLabX. The PIC32MX795 was offered in a development board (the aforementioned Max32) which we acquired to begin developing the code base for the controller and for use as a reference design for the OSAVC.

The basic requirements for the microcontroller operation are decoupling capacitors on all the voltage input and reference pins, a reset connection to the MCLR pin, connections to the programming interface, and the oscillator pins [32]. Although the microcontroller may operate without an external oscillator for precise and stable timing it is recommended to use a crystal oscillator. We selected an 8 MHz oscillator with 30 pF capacitors, and a 0 Ohm resistor to allow for attenuation of the oscillator signal if required. To reset the microcontroller, we implemented a manual pushbutton switch, as well as following the guidelines for the programmer reset function. This is shown in Fig. 3.3. In addition to the basic requirements an FTDI serial to USB converter chip is connected to the first UART port of the microcontroller to allow for communication to an external device. This feature is primarily used for debugging purposes during development, as well as communication to a single board computer (SBC) or ground control station (GCS) during normal operation. This is detailed in the microprocessor sheet of the schematics[4].

The microcontroller requires a stable 3.3V power supply, provided by the Texas Instruments LP38690DT-3.3 low dropout (LDO) voltage regulator. Many sensors and peripherals require 5V operation, and the LP38690DTX-5.0 LDO regulator provides this capability. Because the OSAVC may provide power to an external SBC as well as some high current devices such as servo motors, a Microchip 29300-5.0 5V/3A LDO regulator is used. Finally, the TI PTN78020W switching regulator accepts battery volt-

[4]see: https://github.com/uccross/open-source-autonomous-vehicle-controller

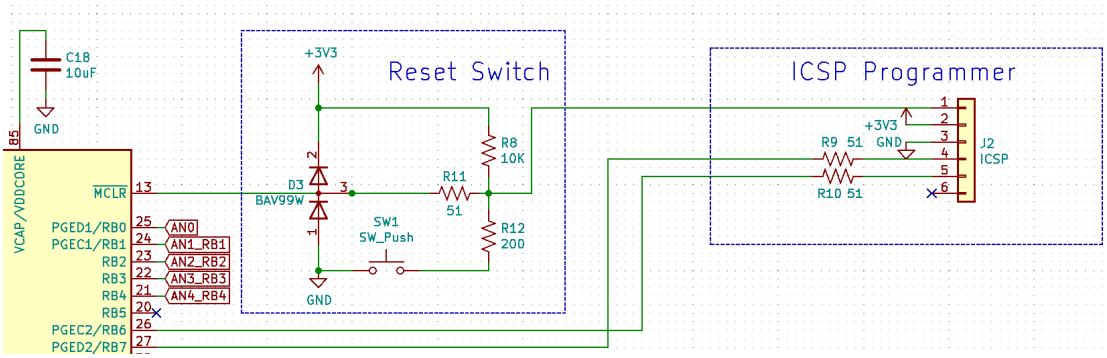

Figure 3.3: Detail of the reset switch and ICSP programmer circuit. The reset switch allows the user to manually reset the microcontroller to start the program execution. The ICSP also resets the microcontroller after new code is loaded.

ages in the range of 7-36V and regulates down to 7V/6A, and feeds the two 5V LDO regulators. The switching regular is used because it is very efficient—above 90%—and allows for reducing high battery voltages without wasting energy in the form of heat. The output of the LP38690DTX-5 in turn provides power to the 3.3V LDO. Alternatively, the USB port can power the LP38690DTX-5 and LP38690DT-3.3, typically used when developing and testing firmware. This is switched by a P-channel MOSFET transistor using a voltage divider and comparator circuit when USB power is present (and no other voltage source) detailed in Fig. 3.4. It can also be found in the power section of the schematics.

The peripherals supported by the OSAVC are listed in Table 3.1. LEDs indicating communication were placed on signal lines for most sensors[5] as well as to indicate the operation of the various voltage sources. A jumper is provided to power the OC modules with the 29300-5.0 and the SBC with up to 3A current at 5V when operated with a battery. The GPIO pins have Schottky diodes to protect the microcontroller from electrostatic discharge events. The detailed design is located in the I/O sheet of the schematics.

[5]Some sensors, such as the ones connected through the SPI ports, operate at too high a frequency, or for too short duration for LEDs to be useful.

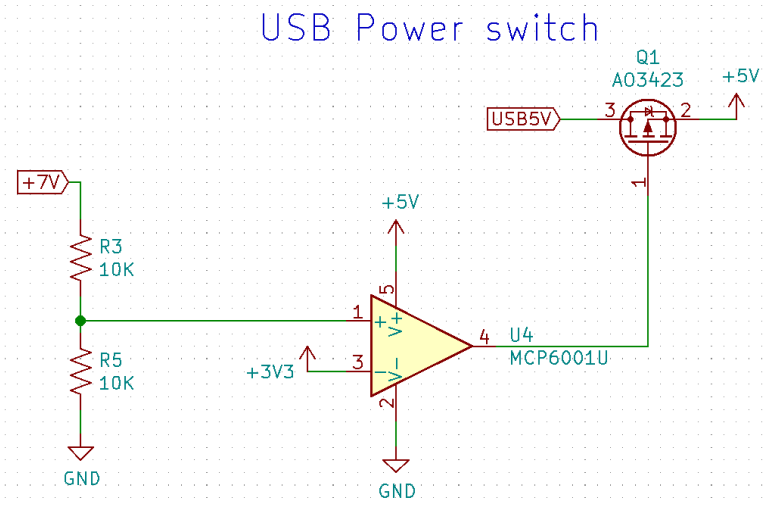

Figure 3.4: Detail of the USB power switch. When no battery voltage is present, the transistor Q1 connects the 5V power net to the USB 5 V input.

3.5 PCB

The guiding principles for the PCB layout balanced phsyical size with manufacturability. Because the design is open source as opposed to a commercial product, we cannot guarantee that an adopter will have access to a reflow oven for soldering components to the PCB. Therefore the design has to support manual soldering while also minimizing board area. Thus, we selected surface mount devices (SMD)—in order to save board space—that were still possible to solder by hand. As a result, the smallest discrete devices in the design come in the 0603 package, with a footprint of approximately 1.55 by 0.85 mm. For the microcontroller, we selected the TQFP-100 package, which has leads on a 0.5 mm pitch and occupies a 14 mm by 14 mm footprint. we made similar choices for the other components. To ease the assembly all the components are located on the top side of the PCB.

For the PCB itself, we also balanced cost vs size by choosing a four layer board. The PCB has two signal planes (the top and bottom layers), an internal ground plane, and an internal 3.3V plane. This design provides some noise immunity by having these mostly complete internal copper planes.

we chose to mount the switching power supply (the TI PTN78020W) on the PCB despite its large size (approximately 39 by 23 mm) in order to minimize the mechanical complexity. The board could have been made much smaller by placing the power supply separately, but we deemed the convenience of a single PCB worth the tradeoff.

The top signal plane and silkscreen of the PCB is shown in Fig. 3.5. For the following discussion refer to the physical board of Fig. 3.6 where the silkscreen designations are easier to see. For the component layout, we chose to have the battery connection enter the PCB from a pair of mounting holes sized for 16 AWG wires on the lower left hand side (J1). Mounted directly adjacent is the PTN78020W (U1). The three LDOs are located along the top and right hand sides of the PTN78020W (U2, U3, U5), and a USB-C power output (for SBC power) is located on the upper left hand corner of the PCB (J23). Also on the left hand edge are the motor outputs and GPIO pins located on a 19 by 3 row header(J4). On the top and bottom edges of the PCB are the sensor and peripheral connectors (J6, J13-21). The microcontroller is located near the center of the PCB (U6). The oscillator (Y1) is located adjacent to the microcontroller to keep the signal traces short and of nearly equal path length. The EEPROM is located on the lower right of the PCB (U9). On the upper right is the FTDI serial-USB converter (U7) which connects to the outside world on the right hand edge of the PCB (J3). The reset button (SW1) and the in-circuit serial programming (ICSP) port (J2) are also located on the right hand edge of the PCB. Refer to the schematics for the location of the various discrete components.

For most signals we selected 0.25 mm trace widths matching the lead width of the microcontroller. For higher current carrying traces, such as those from the voltage regulators or battery, we chose traces of at least 1 mm in width—well above the minimum width recommended by the KiCad calculator—in order to minimize resistive losses and to keep the traces as cool as possible. Signal path crossings between the top signal and bottom signal layers were kept as close to perpendicular as was possible to eliminate coupling between signals.

To keep the LDOs cool, we created large ground planes and sunk vias through the PCB to transfer heat to the interior ground plane. These vias are the arrays of circles seen in the three ground pads of the LDOs.

Finally, the PCB is mounted by the four 3 mm holes (H1-4) and connected to the internal ground plane.

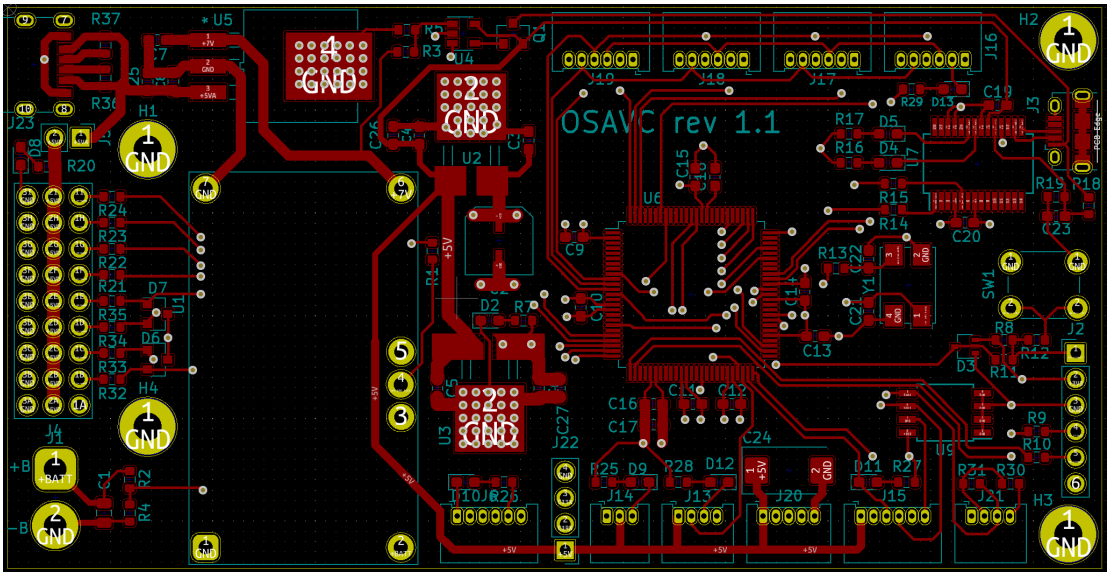

Figure 3.5: The top signal plane and silkscreen of the OSAVC PCB.

For manufacturing the PCB, we chose to produce prototypes using the OSHPark manufacturer[6]. The front and back side of a bare PCB is shown in Fig 3.6

3.6 Assembly

We evaluated two different methods to solder the components onto the PCB. For the first method, we hand placed the components and soldered the leads or pads individually under 10× magnification. For the large ground planes of the LDOs we first melted a thin layer of solder over the pad, placed the LDO, then soldered the leads, then applied heat and solder to the edge of the ground planes. It was evident when the

[6]https://oshpark.com/

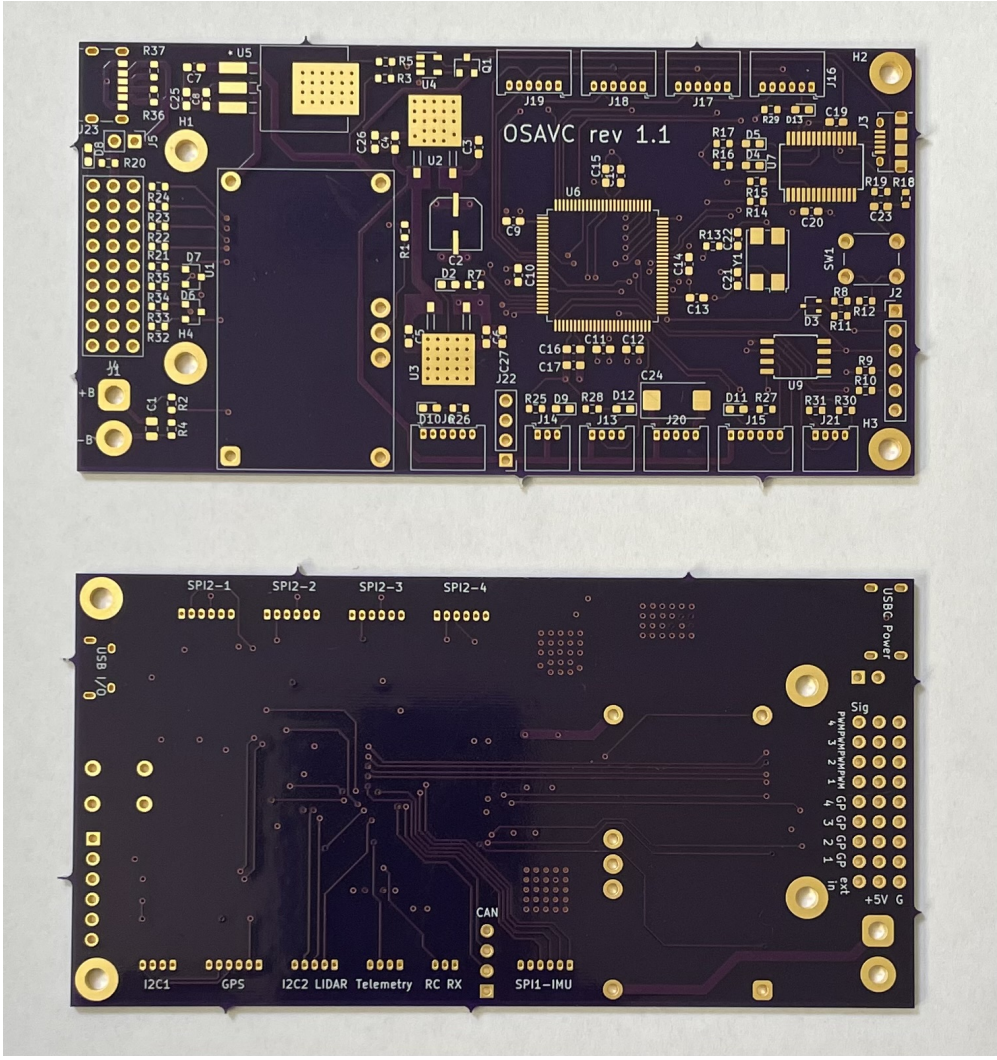

Figure 3.6: The front and back of the manufactured PCB. Note the bottom image is flipped left to right with respect to the top image.

solder on the ground plane melted as it was visible pulled under the LDO. More difficult was to solder the ground planes under the USB-B micro connector as there is no exposed edge on which to apply heat. In this case we used a hot air rework station from underneath the component to ensure a solid solder joint, but this method is not ideal. A completed PCB assembly is shown in Fig. 3.7.

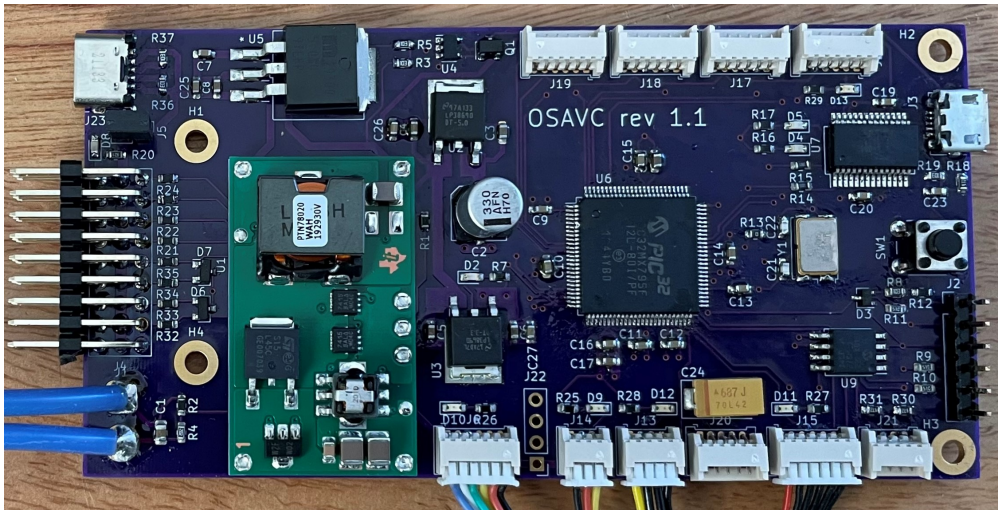

Figure 3.7: The completed PCB assembly. This prototype is hand-soldered. It is easy to see the that the solder has flowed underneath the three LDOs.

For the second assembly method, we purchased a stainless steel stencil and solder paste. We first carefully aligned the stencil to the PCB, then used a stiff card to evenly spread solder paste into the exposed openings of the stencil. After removing the stencil, we placed the components onto the PCB, then used a toaster oven to attempt to flow the solder under the components. This method produced a prototype with many bad connnections and had to be extensively reworked before the PCB was operational. Probably a better approach is to heat the PCB from underneath on a well controlled heater and monitor the solder joints from above, but we have not tested this method.

3.7 Testing

There are several tests to perform prior to attempting to load code onto the OSAVC. These tests ensure that the assembly was performed correctly.

The first test is to plug a USB-B cable from a computer to the matching receptacle on the OSAVC. This should provide power to the board and the LED from the 3.3V LDO should light up. This indicates that the USB power is succesfully delivered to the regulator and therefore the USB connnector is connected. The next step is to verify

With a multimeter that 5V is found on all 5V traces or leads. The LEDs on the FT232RL should also light briefly as the IC negotiates a connection to the computer.

The next test is to unplug the USB power and provide a voltage between 7-36V to the battery connector (XT60). If the LED on the 3.3V LDO lights up, that indicates that all the regulators for the PCB are functional. Again, confirm with a multimeter the voltage at the output of each regulator.

With the power sources and the FT232RL validated, the next test is to load the Serial.X project using the MPLabX IDE and the serial programmer (ChipKit 3 or 4). The IDE will indicate whether the microcontroller has been successfully identified and the code loaded. If so, verify the 3.3V sine wave at both outputs of the oscillator. Finally, the Serial.X project test harness is a simple loopback to an external computer. Communicating over a terminal program over the correct COM port should echo the input back to the user.

From this point on, each peripheral in use is validated with the appropriate driver project loaded and the associated test harness specified in the pre-processor macro configuration of the IDE[7].

Evaluating every sensor and peripheral is a time consuming process. A good area for future development is the design of an automated board testing station. Given the future deployment possibilities, such a testing station would be useful for diagnosing and troubleshooting errors or post-crash analysis.

[7]Using the pre-processor allows us to define a macro which builds the test harness when the sensor module is compiled on its own, i.e., when not included in a larger project

Chapter 4

Software Design

4.1 Introduction

Writing applications for microcontrollers is known as embedded programming as opposed to general programming for larger scale systems. Embedded programming has unique constraints and challenges. The main constraint is the limited memory available for the program and data—typically on the order of hundreds of kilobytes for most microcontrollers. A main challenge is the hardware-specific nomenclature and configuration of the peripherals. For example, to enable a UART on the OSAVC microcontroller one first has to configure the peripheral by changing the register bits corresponding to the memory-mapped IO that control its function[1]. Fig. 4.1 shows a snippet of the initialization of UART1. Although the logic of the routine is clear, the register names and values are cryptic. Determining the correct settings and hardware registers requires careful study of the microcontroller datasheet.

Perhaps the main challenge of writing embedded firmware, however, is ensuring that code is fast and repeatable. The need for speed results from the vehicle control requirement. Most vehicles require control algorithms operating in the tens to hundreds of Hz. Furthermore, the control period itself needs to be repeatable, that is the latency

[1]This is common to almost all microcontroller designs, however, there is no consistancy in naming or control register configuration—thus each peripheral must have its hardware specified carefully

and jitter of the control loop needs to be predictable and small. This arises from the fact that while controller design on a microcontroller is necessarily digital, the vehicle itself operates in the continuous domain. Variations in the period of the control loop introduce extra noise in the form of timing jitter which can make the control less precise in the best case or unstable in the worst case.

These constraints and challenges are a significant barrier to adoption of the OSAVC. In particular, to ensure efficiency and low latency, all the peripheral drivers[2] have to be non-blocking, that is, they cannot allow the processor to stall while waiting for data. This requires the drivers to use interrupts to communicate to the peripheral. To minimize some of the difficulty for an adopter of the OSAVC, we have written drivers for many common sensors and devices, developed control application code that can be adapted to a specific vehicle implementation, and incorporated several useful utilities common to most autonomous vehicles. This work is detailed in the next sections.

4.2 Peripheral Drivers

The peripheral drivers are located in libraries in the OSAVC repository. Each driver consists of two files: a header file that specifies the module public methods and a source file. Inside the source file the private variables and functions are declared static to limit their scope to the module. In this manner the code base is modular and similar to object-oriented programming. All interactions with the peripheral are handled through the use of interrupts.

An interrupt is a process whereby the microcontroller receives a signal (known as an interrupt request, or IRQ) from a hardware peripheral informing it that a peripheral needs attention, for example, when a new piece of data is received. When the microcontroller receives the IRQ it pauses execution of the main program and jumps to an address in memory pointed to by the interrupt. At this memory location is a pointer to the appropriate interrupt service routine (ISR). The ISR is a small piece of code which

[2] A driver is a piece of code that configures and operates a hardware device of the microcontroller, e.g., a serial port.

handles the particular hardware need and clears a flag in the peripheral register to let it know it has been handled. Each interrupt has a priority (and subpriority if necessary) to deal with the case when multiple interrupts occur simultaneously.

To make process this more concrete, the code in Fig. 4.1 is configured to interrupt the processor when a character is received (U1RXIE) or transmitted (U1TXIE). When a new byte is received from the UART it is stored in a register. The UART sets a flag in a control register indicating what caused the interrupt. The microcontroller then pauses code execution and jumps to the ISR. Inside the ISR, the character is read from the register and stored in a buffer. The flag is cleared and the microcontroller restarts the main code. A snippet from the UART1 ISR is shown in Fig. 4.2.

Many vehicle developers will never need to write a peripheral driver if they utilize the existing sensors and devices listed in Table 3.1 from the previous chapter. However, if they need a specific device the existing code base can be used as a template.

```
U1MODEbits.BRGH = 0;
/* set baud rate */
U1BRG = ((Board_get_PB_clock() / BAUD_RATE) / 16) - 1;
/* configure the RX and TX pins */
U1STAbits.UTXEN = 1;
U1STAbits.URXEN = 1;
/* clear overflow */
if (U1STAbits.OERR == 1) {
    U1STAbits.OERR = 0;
}
/* configure using software flow control, if set to 2 it would be CTS
   /RTS */
U1MODEbits.UEN = 0;
/* configure UART interrupts */
/* interrupt when buffer is not empty */
U1STAbits.URXISEL = 0x0;
/* int when all characters are sent (TRMT == TRUE) */
U1STAbits.UTXISEL = 0x01;
/*clear interrupt flags */
IFS0bits.U1RXIF = 0;
IFS0bits.U1TXIF = 0;
/* set interrupt priority to 1 */
IPC6bits.U1IP = 1;
/* enable interrupt on RX & TX */
IEC0bits.U1RXIE = 1;
IEC0bits.U1TXIE = 1;
/* turn on UART */
U1MODEbits.ON = 1;
```

Figure 4.1: Code snippet for configuring a UART for the OSAVC. The cryptic register names and settings are a challenge when writing embedded firmware. The sequence of configurations are: set the baud rate, turn on the receiver and transmitter, clear the buffers, set up the flow mode (hardware or software control), configure the interrupt, clear interrupt flags, define the interrupt priority, enable the interrupts, and turn on the UART.

```
void __ISR(_UART_1_VECTOR, IPL1SOFT) IntUart1Handler(void) {
    if (IFS0bits.U1RXIF) { //check for received data flag
        if (is_buffer_full(rxp) == FALSE) {
            if (reading_rx_buffer == FALSE) {
                write_buffer(rxp, U1RXREG);
            } else {
                /*a collision occurred need to disable interrupts to
                   exit ISR*/
                /*it will be re-enabled in get_char() */
                rx_collision = TRUE;
                IEC0bits.U1RXIE = 0;
            }
        }
IFS0bits.U1RXIF = 0; // clear the flag
...
```

Figure 4.2: Code snippet handling a character received from the UART1.

4.3 Other Libraries

In addition to the peripheral drivers, there are a few other libraries that are useful for autonomous vehicles. One is the MAVLink subrepository. MAVLink is an open source lightweight binary communication protocol[3] originally designed for micro aerial vehicles to communicate with ground control stations but now used widely in autonomous vehicle communication. A second library is a linear algebra library of many common computations for vector, matrix, and quaternion operations. Another useful library is an attitude estimation algorithm used to orient a vehicle in three dimensional space. We present this algorithm in some detail later in Chapter 5. Finally, a general PID controller library is also available.

4.4 Application Architecture

Finally we have written sample control applications using a state machine approach to simplify the algorithms to the minimum complexity required. A typical architecture for an application is shown in the state machine diagram in Fig. 4.3. In this example all the initializations are performed first, then a simple while loop is entered. At the start of the loop the current time is queried (one of the libraries in the repository is a hardware timer) after which the state transition occurs to the next state. The Check Timers state checks to see if a timer has expired. These are the synchronous events we define. For example, we could configure a timer for 10 msec (100 Hz) to compute a new control command. If the control period has elapsed the main loop calls the controller update in the Service Timer state. Once the control update completes the state checks the next timer. The Check Timers state is exited once all the timers have been evaluated. Using this structure we can define as many timers as needed, all based on one hardware timer.

After the synchronous events have been evaluated, the next state is Check Peripheral Events. These events are represented by boolean values indicating, for example, that a sensor has new data to be processed. These events are asynchronous, meaning

[3] https://mavlink.io/

that they may occur at any time. If an event has occurred, the state transitions to the Service Peripheral state—a function which services the event. For example, the radio control module receives data over UART5. If a new command string from the radio controller is available, the application calls a function which decodes and stores the radio controller data. These data consist of the various switch and gimbal settings of the controller. After decoding and storing the data, the state transitions back to Check Peripheral Events, which continues until all events have been checked.

At any point in the main loop the microcontroller may be interrupted by the hardware peripherals. This is seen as the arrow in the lower left of the figure labeled 'IRQ' and representing an interrupt request. When one is detected, the main loop execution is paused and the ISR is performed. Once complete the main loop continues.

This architecture is used for both manual and autonomous control of a few different vehicles presented in Chapters 7 and 8.

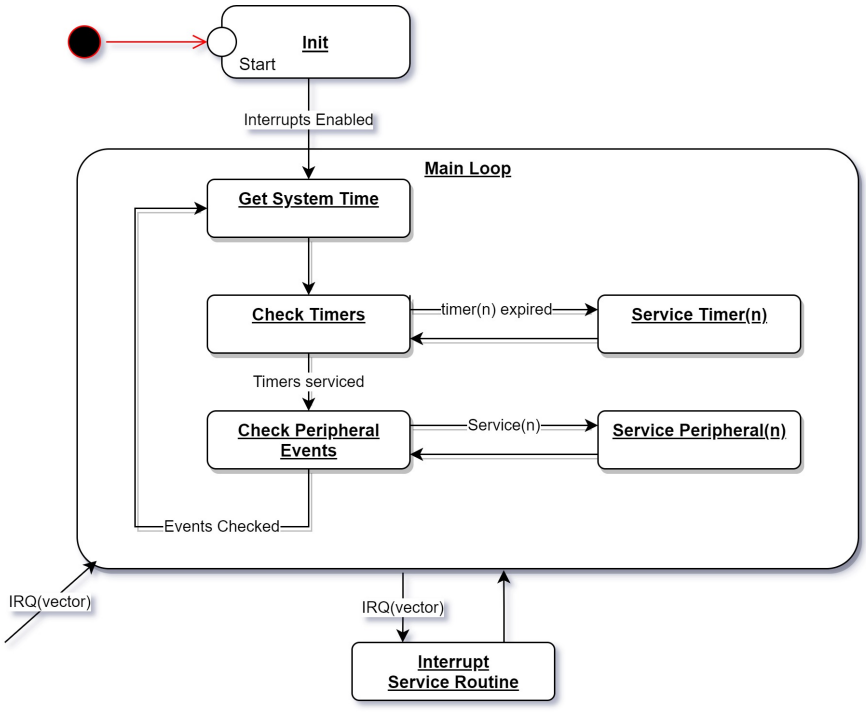

Figure 4.3: A simple pattern for a control application on the OSAVC.

4.5 Developing Control Algorithms on the OSAVC

The architecture presented in the previous section allows for modular controller update functions. A general controller update flowchart is shown in Fig. 4.4. Once the control timer has expired, the update sequence is called from the Check Timer state. The first action is to reset the timer. Next a function is called to estimate the current state of the vehicle. A state vector consists of all the parameters that define the vehicle's dynamics (position, velocity, angular velocity, attitude, etc.). The state vector is sent to the controller which uses it and the desired state to calculate the outputs (generally motors or servos). These outputs are sent to the actuators and the execution flow returns to the calling state.

This architecture demonstrates three great advantages of using the OSAVC. The first is that both the state estimation function(s) and the controller itself consist of two files: a header file and a source file. Thus testing a new algorithm is as simple as replacing two files and recompiling the source code. The second is that it is easy to evalute the performance of the new algorithms with respect to latency by making use of the system hardware timer. We use this technique to evaluate benchmark results in the next chapter. Finally, this architecture allows algorithms to be developed and tested in Matlab before deploying to the vehicle using the automatic code generation capability of the software. We used this capability to develop two of the benchmark algorithms in the next chapter as a demonstration.

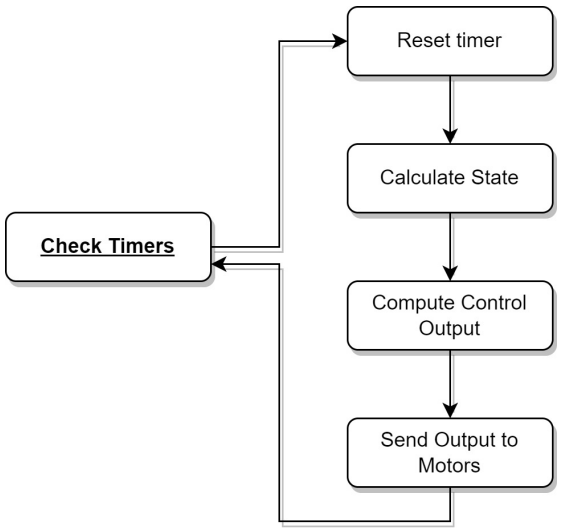

Figure 4.4: A sample update sequency for a vehicle controller.

Chapter 5

Benchmark Description

5.1 Introduction

In this chapter we discuss a method for characterizing the performance of the OSAVC. The inspiration for this comes from the computer industry which often employs benchmark tests designed to compare the performance of various systems using a common algorithm. Ideally, the benchmark study would compare the performance of the OSAVC against a commercially-available autopilot, for example. However, the difficulty of implementing a novel algorithm onto such an autopilot is one of the main motivations for developing the OSAVC in the first place, namely that it is challenging to modify the source code of these existing devices. Instead, we designed a study that compares four different hardware configurations: the OSAVC, the Digilent UC32 development board, the Raspberry Pi Pico RP2040 microcontroller, and the Raspbery Pi 4b computer hosting a (non-real-time) standard Linux OS. More detail regarding these processors is found in Chapter 6.1 where we present the results.

The benchmark uses an attitude estimation algorithm developed by Mahoney [22] to evaluate the hardware systems. Attitude estimation is a method to determine the orientation of a vehicle in three dimensional space and is used by most autonomous systems in some form. Attitude estimation algorithms are known as attitude heading reference systems, or AHRS. This particular algorithm uses two inertial sensors—a

triaxial magnetometer and a triaxial accelerometer—as well as a triaxial gyroscope sensor to provide the input to a complementary filter. This filter is discussed in detail in a following section, however, fundamentally it measures the orientation of two inertial vectors (gravity and the local magnetic field vector) along with the angular velocity in the vehicle frame to determine the attitude of the vehicle in the inertial frame. This approach is similar to the Kalman filter [14] which is optimal when the dynamics are linear and the noise sources are normally distributed. The complementary filter has been shown to match the performance of the Kalman filter when tuned properly [6], moreover it is much less computationally expensive making it suitable for embedded systems.

The parameters of interest in the benchmark are the mean and distribution[1] of the latency of the filter. Latency is an important parameter because it dictates the speed of the update rate. For challenging applications, e.g., the stabilization of a quadcopter or similar UAV, the update rate must be as fast as possible to provide the greatest margin of stability. The distribution of the latency is also important because variation in the latency can also affect the stability of the craft.

We implemented four different variations of this filter to provide deeper understanding of the performance of these hardware systems. Two implementations use quaternions to represent the attitude in the estimation, one with single precision floating point numbers, and one with double precision. The other two implementations use direction cosine matrices (DCMs) to represent the attitude in the algorithm, again one using single precision floats and one using double precision. These are discussed in greater detail in Section 5.6.

[1] If latency were a random variable we would be concerned with the variance, however, the latency of a non-real-time system is not normally distributed. Instead we will examine the distribution of the latency values.

5.2 Complementary Filter

A complementary filter is a feedback filter designed to combine two or more dynamic inputs to provide the best estimate of a given parameter. The two different inputs have different dynamics, in particular, they are accurate in different frequency domains. A typical application is one where two sensors estimate the same quantity, but one sensor is accurate in the low frequency domain but has high frequency noise and the second sensor has good high frequency dynamics but is inaccurate at low frequencies. A general block diagram for the complementary filter is shown in Fig. 5.1 below and a derivation can be found in Appendix A. Note that this diagram is specifical to the case where Y_u represents a time derivative of the variable we wish to estimate. If it were a direct measurement of the variable it would enter the block diagram after the integration term. Referring to the figure, $\hat{X}$ is the estimate of the parameter X. Y_x is a measurement of X with good low frequency response and Y_u is a measurement of $\dot{X}$, the rate of change of X with good high frequency response. $C(s)$ is the transfer function of the filter and determines the frequency crossover point between the two signals. Following the signal path from left to right, the difference between Y_x and $\hat{X}$, form the error signal, $e(s)$, which is multipled by $C(s)$ and added to Y_v. The summand is integrated to determine $\hat{X}$, which is fed back to the input.

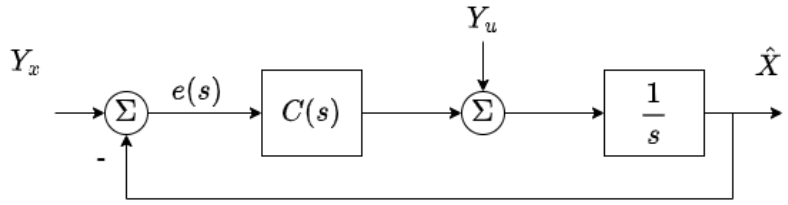

Figure 5.1: The complementary filter block diagram.

5.3 Attitude Estimation

In the AHRS application, we consider an IMU attached to the body frame of a vehicle. The magnetometers and accelerometers of the IMU both have good low frequency accuracy but suffer from high frequency noise. Gyroscopes, on the other hand, have excellent high frequency accuracy but suffer from a slowly varying bias. In simple terms the attitude estimation filter compares the current estimate of the attitude relative to a known inertial vector, e.g., gravity, acting on the sensor. The difference between the estimate and the known value creates an error vector, which can be expressed as an error rate by the transfer function $C(s)$. This vector is added to the gyroscopic measurement of the vehicle angular velocity. The resulting vector is the total error rate which is integrated into a rotation applied to the prior attitude estimate to determine the new attitude estimate.

The filter used for the benchmark algorithm is adapted from [22]. It computes two different error terms (one using gravity as a reference vector, and one using the local magnetic field) that are weighted according to their frequency responses and added together to form a composite error term before including the vehicle angular velocity and integrated numerically. For purposes of the benchmark, we chose to use the same weighting of the magnetic and inertial to be the same value, $k_p = 2.5$, and we chose the integral term $K_I = .05$ as these terms provided good tracking and noise performance. Note, however, that these terms can be tuned for desired performance using frequency domain analysis of the filter transfer function. The block diagram of the filter is shown in Fig. 5.2. It is similar to the one of Fig. 5.1 but differs in implementation. Although this block diagram demonstrates the DCM implementation of the filter, the quaternion implementation follows the same logic. Note that this figure only shows a single inertial vector forming the error term but it should be clear from the discussion above that two or more inertial vectors can be added to improve the filter estimate. Referring to the figure, $\mathbf{\Omega_y}$ is the vector of the measured gyroscope rates. These rates are put into the cross product operator $(\Omega)_\times$. This is the (skew symmetric) matrix form of the cross product, i.e., $(\Omega)_\times \mathbf{v} = \mathbf{\Omega} \times \mathbf{v}$, and represents the measured vehicle angular velocity.

R_y is an inertial vector measurement expressed as a DCM, that is, a the inertial quantity expressed as a rotation in matrix form. The block operation $\hat{R}^T R_y$ is the matrix method to form an error term. This is evident if we realize that if $\hat{R}$ is equal to R_y then the block yields the $\mathbb{I}^3$ identity, which represents zero rotation. The output of this block is rotation error, $\tilde{R}$. This error is mapped by the skew-symmetric matrix projection operator $\mathbb{P}_a(\tilde{R})$ into an error *rate* in the same form as $(\Omega)_\times$, that is, it can be considered an error rate once scaled appropriately. This scaling is performed by the gain block k. The output of the gain block and the $(\Omega)_\times$ block are summed together to provide the total error dynamics A. The multiplication of the previous attitude estimate and A yields the estimate error rate, $\hat{R}A = \dot{\hat{R}}$, integration of which yields the new attitude estimate.

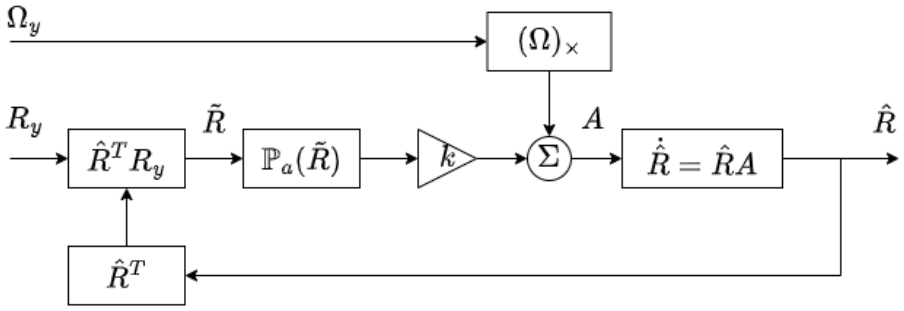

Figure 5.2: The attitude estimation filter block diagram expressed for the DCM implementation, adapted from [22].

5.4 DCM Implementation

The DCM attitude estimation filter is implemented with explicit gyroscope bias correction estimated in parallel with attitude and uses multiple inertial vector measurements (rather than the rotation matrices, R_y). This is called explicit attitude estimation and is described by the following equations. Each reference inertial vector, v_{0i} is given in the inertial frame and is rotated into the body frame using the prior attitude estimate as given in Eq. 5.1. The cross product of the estimated and the scaled measurements of

the inertial vectors, v_i, are summed together as described by Eq. 5.2 to provide the inertial error rate, ω_{mes}. Separately, the bias rate estimate of the gyroscopes is determined from Eq. 5.3. Finally, the bias-corrected gyroscope measurements are combined with the inertial error rate and multiplied by the prior attitude estimate as given in Eq. 5.5. This quantity is integrated to provide the new attitude estimate using the matrix exponential acting on $A\Delta t$ as shown in Eq. 5.6, where Δt is the integration time. for clarity, we introduce the notation $\hat{R}^+$ and $\hat{R}^-$ as the new attitude estimate and prior attitude estimate respectively. The integration is carried out using the Euler-Rodrigues method as formulated in [31], given in Eq. 5.7. Finally, the new bias estimate $\hat{b}^+$ is obtained through forward Euler integration as shown in Eq. 5.8. Again for clarity we represent $\hat{b}^+$ and $\hat{b}^-$ to represent the new and prior bias estimates, respectively.

$$\hat{v}_i = \hat{R}^T v_{0i} \tag{5.1}$$

$$\omega_{mes} := \sum_{i=1}^{n} k_i v_i \times \hat{v}_i, \quad k_i > 0 \tag{5.2}$$

$$\dot{\hat{b}} = -k_I \omega_{mes} \tag{5.3}$$

$$A = (\Omega^y - \hat{b})_\times + k_P(\omega_{mes})_\times \tag{5.4}$$

$$\dot{\hat{R}} = \hat{R}A, \quad \hat{R}(0) = \hat{R}_0 \tag{5.5}$$

$$\hat{R}^+ = \hat{R}^- \exp(\|A\|\Delta t(A)_\times) \tag{5.6}$$

$$= \hat{R}^- \left(\mathbb{I} + \frac{(\sin \|A\|\Delta t)(A)_\times}{\|A\|} + \frac{(1 - \cos \|A\|\Delta t)(A)_\times^2}{\|A\|^2} \right) \tag{5.7}$$

$$\hat{b}^+ = \hat{b}^- + \dot{\hat{b}}\Delta t \tag{5.8}$$

The use of the trignometric functions in the integration and the redundancy of nine terms to represent attitude introduces significant computational burden. Thus we consider another method to implement the algorithm in the next section using quaternions and quaternion integration.

5.5 Quaternion Implementation

The quaternion implementation of the attitude estimation filter follows the same steps as the DCM form. The differences lie in the rotation, calculation of the total error derivative, and integration steps. The rotation of the inertial vectors, v_{0i}, into the body frame uses quaternion rotations as given in Eq. 5.9. Note the $\mathbf{p}()$ operator which converts a vector quantity to a pure quaternion[2] and the $\mathbf{vex}()$ operator which converts a pure quaternion to a vector. The attitude error rate, $\dot{\hat{q}}$ is computed in Eq. 5.12 and is the quaternion analog to the DCM error rate $\dot{\hat{R}}$ shown in Eq. 5.5. Using the $\hat{q}^+$ and $\hat{q}^-$ notation to mean the new and prior attitude estimates, the quaternion integration is shown in Eq. 5.13.

$$\hat{v}_i = \mathbf{vex}(q \otimes \mathbf{p}(v_{0i}) \otimes q^*) \tag{5.9}$$

$$\omega_{mes} := \sum_{i=1}^{n} k_i v_i \times \hat{v}_i, \quad k_i > 0 \tag{5.10}$$

$$\dot{\hat{b}} = -k_I \omega_{mes} \tag{5.11}$$

$$\dot{\hat{q}} = \frac{1}{2}\hat{q} \otimes \mathbf{p}(\Omega^y - \hat{b} + k_P \omega_{mes}) \tag{5.12}$$

$$\hat{q}^+ = \hat{q}^- + \dot{\hat{q}}\Delta t \tag{5.13}$$

$$\hat{b}^+ = \hat{b}^- + \dot{\hat{b}}\Delta t \tag{5.14}$$

5.6 Implemented Filters

The motivation to evalute two attitude estimation methods with two floating point precision values is to find the most efficient (with respect to latency) algorithm that doesn't compromise accuracy of the result. Trigonometric functions (used to perform the matrix explonential integration in the DCM version of the filter) in embedded systems are expensive despite heavily optimized libraries. Floating point operations are

[2] A pure quaternion is one where the scalar value is zero and can represent an angular velocity. This is the analogous to the matrix projection operator $\mathbb{P}_a()$ in matrix algebra.

similarly expensive particularly when operating on double precision values which require 64 bits per number. Since most embedded processors use 32 bit registers, double precision floating point operations approximately double the latency of a given operation.

Code efficiency impacts latency as well. One objective of this study is to determine how automatic code generation compares to hand-written code. It is convenient to develop new algorithms in software suited for that purpose, such as Matlab, and then to use automatic code generation to translate the algorithm into C, rather than transcribe the algorithm manually. The advantage to this technique is that it is fast and doesn't introduce bugs into the resulting code. The disadvantage is that the code is not optimized for speed. Handwritten code can minimize the number of floating point operations, for example, by precomputing factors used more than once in a given computation. However, the code must be debugged and tested rigorously, which increases development time.

Four different versions of the AHRS filters were implemented. The first version (version 'SQM') uses single precision floating point numbers and a hand-coded quaternion-based representation of attitude. Version 'DQM' is the same as 'SQM', except using double precision floating point numbers. Version 'DQA' is the same as 'DQM' except the code is translated from Matlab using automatic code generation. Finally, version 'DDA' is the Matlab-coded double-precision DCM implementation. the four verions were chosen to highlight how the numerical precision requirements, the representation method, and how automatic code generation affect the latency of the algorithm. The versions are summarized in Table 5.1

Version	Numerical Precision	Implementation Method	Code Creation
SQM	Single	Quaternion	Manual
DQM	Double	Quaternion	Manual
DQA	Double	Quaternion	Automatic
DDA	Double	DCM	Automatic

Table 5.1: The four versions of AHRS filters evaluated in this work. Manual code creation means the algorithms were directly coded, automatic refers to the Matlab code generation of C functions.

Chapter 6

Benchmark Results

6.1 Systems Under Evaluation

In this chapter we present the performance of four systems against the benchmark developed in the prior chapter. The systems are the Digilent UC32, the OSAVC, the Raspberry Pi Pico and the Raspberry Pi 4b SBC.

The properties of the PIC32 microcontroller used in the OSAVC were discussed previously, however, we reiterate some of the salient features of it here. It has a single core MIPS microcontroller operating at 80 MHz. It has 128 Kb of SRAM and 512 Kb of flash memory. Unlike many microcontrollers it has an extensive list of dedicated hardware peripherals.

The Digilent UC32 is a development board for the Microchip PIC32MX340 microcontroller. It has similar specs to the PIC32MX795, but with fewer dedicated hardware peripherals.

The processor of the Raspberry Pi Pico is the RP2040 microcontroller (designed by the Raspberry Pi Foundation). It contains dual ARM Cortex M0+ processors operating at 120 MHz (capable of up to 132 MHz), with 264 Kb SRAM and 2 MB of flash memory. It has a limited number of dedicated hardware peripherals[1], however, it does contain eight distinct hardware 'slices' of programmable IO that can mimic many

[1] This is the main reason this microcontroller is inadequate for a general purpose autopilot.

different peripherals. Although the RP2040 can support an RTOS, it is tested here with a light HAL in the form of the software development kit (SDK). The SDK allows for flexible assignments of the IO pins, lightweight abstractions for the supported protocols, and an optimized math library stored in the ROM that allows for fast floating point operation.

The last system evaluated in this study is the Raspberry Pi 4b SBC running a Linux OS (Debian Buster). Its processsor is the Broadcom BCM2711, quad core Cortex-A72 (ARM v8) 64-bit system on chip (SoC). It operates at 1.5 GHz and has 4 GB RAM. It has 40 general purpose IO pins and supports many common digital protocols.

All experiments used the same sensor for input, the BNO055 IMU communicating via the I2C interface. This sensor contains two inertial sensors—a triaxial magnetometer and a triaxial accelerometer—as well as a triaxial gyroscope sensor and can operate at 100 Hz repetition rate, although the magnetometer is only updated at 20 Hz.

The sensor was placed on a small prototype board and powered by the system under evaluation. The OSAVC, UC32, and the Pi connected to the sensor with wires, the RP2040 was small enough to be mounted on the prototype board itself. This is shown in Fig. 6.1.

6.2 Procedure

Attitude estimation fundamentally measures a dynamic state of a sensor, that is, the orientation and rotation of the sensor in the inertial frame. Ideally, each benchmark evaluation would involve the sensor undergoing an identical sequence of rotations while running the estimation filter. One method to accomplish this is to mount three sensors on a platform and direct the signals to the hardware systems under evaluation. This is cumbersome, however, and requires more components than were available. Instead we repeated a sequence of rotations in a total time of approximately 30 seconds of the sensor for each evaluation. The sequence of rotations is a $\pm 90^{\circ}$ rotation in the yaw axis,

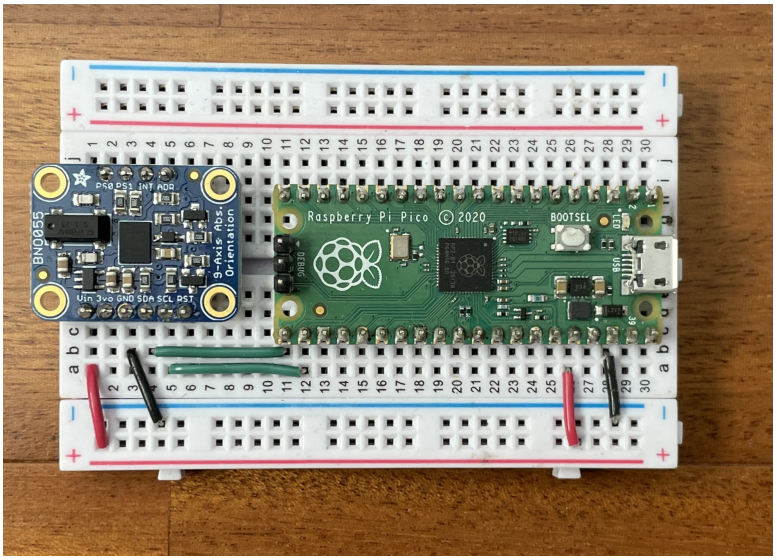

Figure 6.1: The Bosch BNO055 IMU mounted on a prototype board along with the Raspberry Pi Pico.

a $\pm 90^{\circ}$ rotation in the roll axis, and a $\pm 90^{\circ}$ rotation in the pitch axis. A sample rotation sequence is shown in Fig. 6.2.

Inside the algorithm the duration of every iteration of the algorithm was timed with the internal hardware timers available in the three systems. The algorithm updated every 20 msec, or at 50 Hz. The resulting update duration and the attitude estimate were transmitted to a computer that stored the data.

6.3 Results and Discussion

The data for all systems are summarized in Table 6.1 and plots are provided for more insight into how each system operates with respect to the others.

Despite having similar specifications for the OSAVC microcontroller, the UC32 development board performed the worst of all the sytems tested. Fig. 6.3 shows the distribution of the latency for three AHRS algorithms (version 'DQM' was implemented after these tests were performed). From this figure, we can see that the filter version SQM (refer to Table 5.1 for the description of each version) performed the best with a

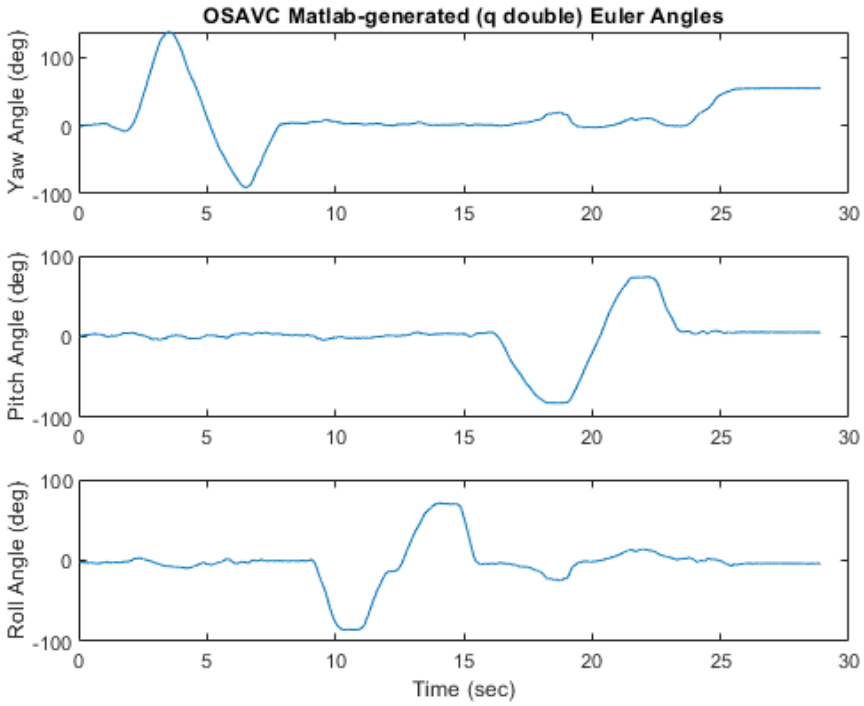

Figure 6.2: The rotation sequence used for all benchmark tests. The sensor is first rotated in the positive and negative direction in the yaw axis, then in the roll axis, and finally in the pitch axis. All tests were 30 seconds long.

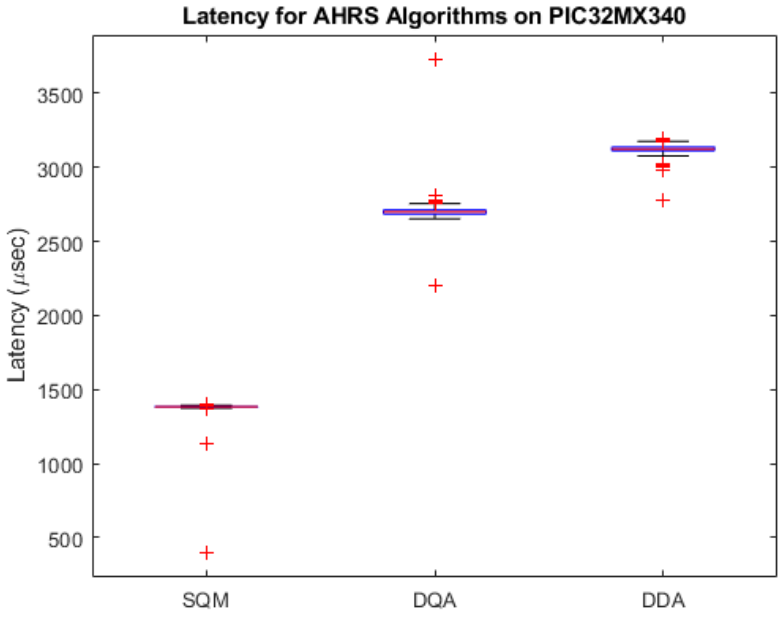

Figure 6.3: AHRS algorithm latency distributions for the UC32 development board. This development board performed worse than the other systems evaluated.

mean latency of about 1.37 msec. Version DQA performs significantly worse at 2.70 msec. Finally version DDA has the longest latency of all at 3.12 msec.

The OSAVC performed significantly better than the UC32. Fig. 6.4 shows the latency distributions for four versions of the AHRS algorithm. Not surprisingly, the version SQM performs the best with a mean latency of 256 μ sec, follwed by version DQM at 357 μsec, DQA at 446 μsec, and DDA at 508 μsec.

The Raspberry Pi Pico performed marginally better than the OSAVC for versions SQM (200 vs 250 μsec) and DQM (approximately 330 vs 357 μsec), but similarly or worse for DQA and DDA. Plots for the four AHRS filter version are shown in Fig. 6.5. Interestingly, the histograms, shown in Fig. 6.6, indicate a bimodal distribution of values around two distinct means rather than the apparently normal distributions for the UC32 and OSAVC. One possible explanation is that the optimized math libraries resident in the boot ROM of the Pico have different latencies depending on the conditions of the algorithm, for example when the angles are small between rotations.

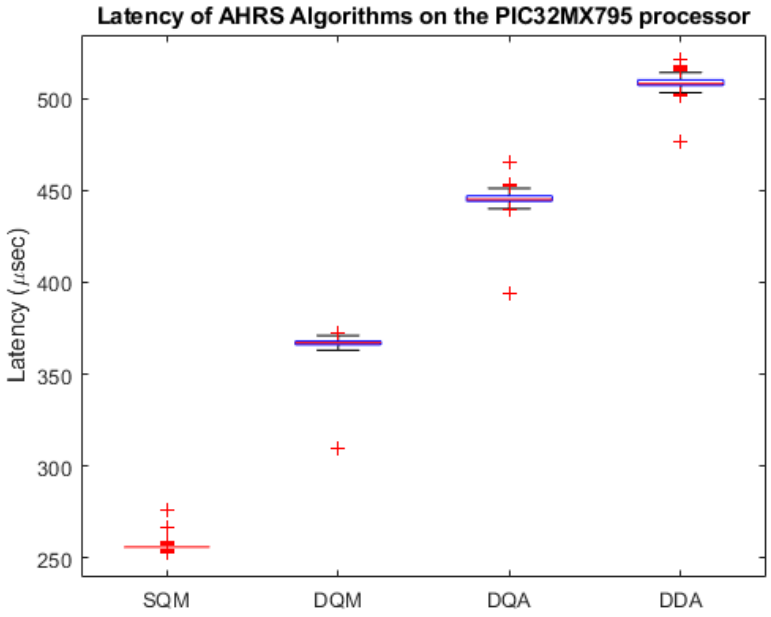

Figure 6.4: AHRS algorithm latency distributions for the OSAVC.

The results for the Raspberry Pi are shown in Fig. 6.7. These distributions are clearly distinct from the other systems. First, the latencies are significantly shorter (2-10 μsec vs 230-550 μsec) but the distributions are clearly not normal. Recall that the difference between these two versions is just numerical precision, suggesting that the Pi OS automatically computes at full precision, regardless of the data type. What isn't clear in the distributions is the full width of the distributions. While the arithmetic mean of the version SQM, for example, is 1.8 μsec, the difference between the maximum value and minimum value is 50.0 μsec.

Another striking difference between the Pi performance versus the other systems is that the latency depends upon the method of data collection. Early on during the benchmarking experiments we observed that if the data was streamed from the Pi over a serial port, the latency was nearly double the value for data stored directly to a file on the Pi. This is displayed clearly in Fig. 6.8. This effect may be due to the OS running

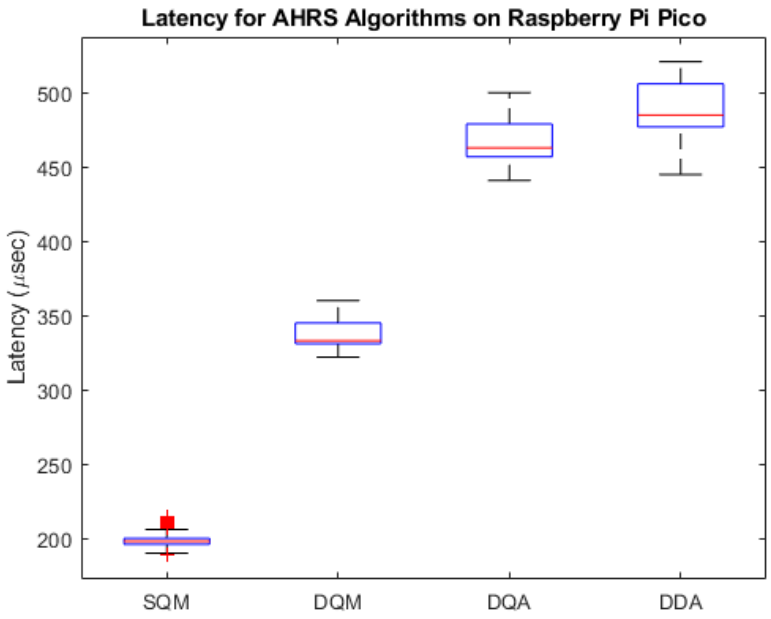

Figure 6.5: AHRS algorithm latency distributions for the Raspberry Pi Pico development board.

other tasks while computing the estimate, in this case operating the serial port—even though the data is streamed after the filter completes an update.

6.4 Conclusions

Several conclusions can be drawn from this benchmark study. For the microcontroller systems (that is, excluding for the moment the Raspberry Pi) the order of the four AHRS algorithms from lowest latency to highest was consistent: SQM,DQM,DQA,DDA. Comparing SQM vs DQM, we can see that single precision calculations are faster than double precision ones, all else being equal. This is unsurprising for 32 bit microcontrollers. We would expect that the double precision calculations would take twice as long as single precision ones. Because the bulk of the latency is driven by floating point operations we should expect an increase the total latency by as much as a factor

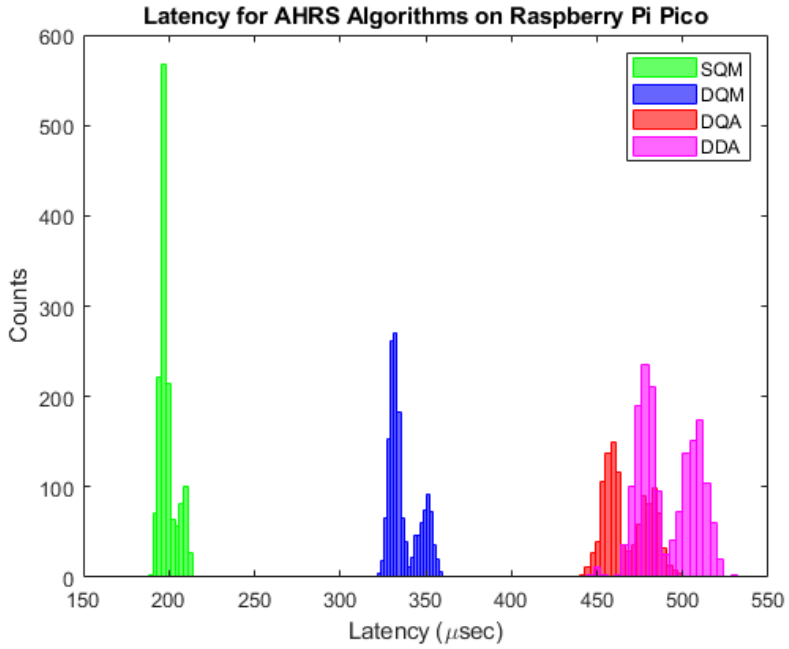

Figure 6.6: AHRS algorithm latency histograms for the Raspberry Pi Pico development board. Note the bimodal distributions, possibly due to optimized math libraries.

of two. The results bear this out with an increase of 95% for the UC32, 43% for the OSAVC, and 65% for the Pico.

Another observation is that automatic code generation introduces significant latency increases. Comparing version DQM and DQA on the OSAVC we see an increase of 78 μsec (21%) when using the Matlab-generated functions. The comparison is even more striking on the Pico, which exhibits an increase of 130 μsec (39%) on average. These differences are significant, however, but not so large as to discourage the use of automatic code generation. This points to a feasible path for developing algorithms on the computer using Matlab and compiling them for the microcontroller without the introduction of bugs and with a manageable increase in latency. This is important because it demonstrates a means for vehicle developers to implement real-time control or estimation algorithms without needing to be specialists in embedded firmware.

Finally, when comparing quaternions to DCMs, we see that DCMs require longer computation times, ranging from roughly 5% (Pico) to 15% (UC32 and OSAVC). This

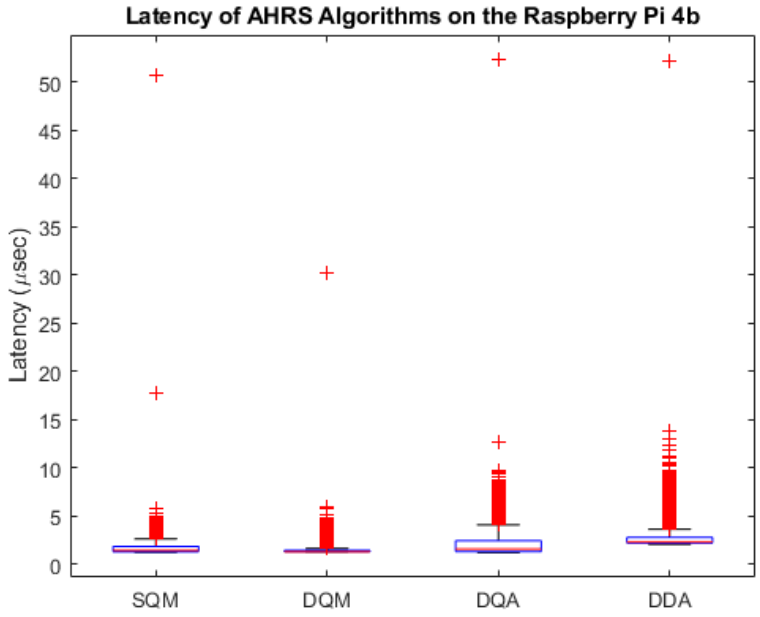

Figure 6.7: AHRS algorithm latency distributions for the Raspberry Pi SBC. The distributions are not normal but are much faster than the microcontroller-based systems.

is due to the larger number of floating point operations due to the redundant information carried in the DCM as well as the more expensive method to integrate the result at each time step. This last is a subtle point. DCMs require the use of the matrix exponential for integration to ensure numerical accuracy of the result, that is, to ensure that the resulting attitude from the DCM algorithm is orthonormal. For quaternion integration, it is possible to integrate using Euler integration and then re-normalize the result if necessary.

There are a few conclusions we can draw regarding the microcontroller systems themselves. Clearly, the Digilent UC32 development board performs much worse than either the OSAVC or the Pico. This is mysterious given the similar microcontroller specifications and architecture of the UC32 to the OSAVC.

The OSAVC performed slightly worse than the Pico when evaluating the hand-coded algorithms. In all likelihood this is due to the faster clock speed of the Pico—120 MHz vs 80 MHz. Interestingly, that result is reversed upon comparing the automatic

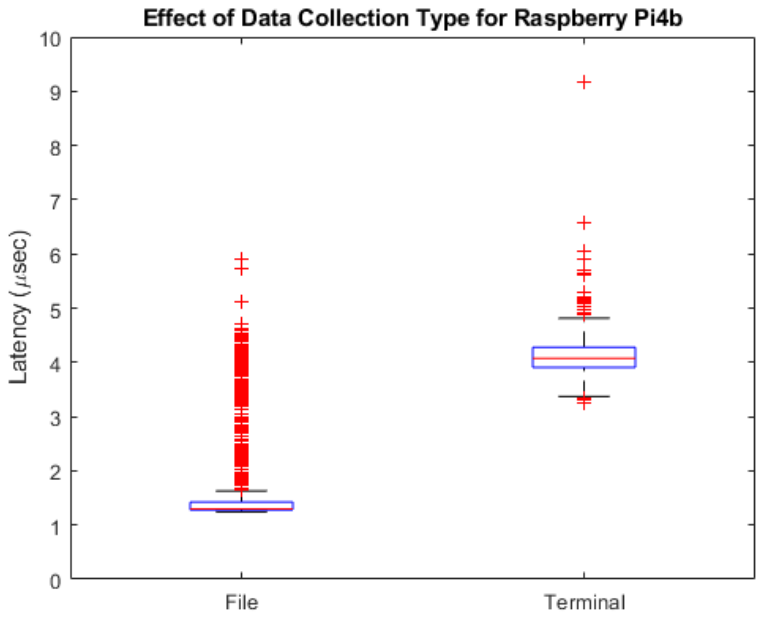

Figure 6.8: The latency of AHRS filter version DQM on the Raspberry Pi using two different methods of data collection: stored directly to a file or streamed over the serial port. The latencies appear to depend on which method is used. This behavior is likely the effect of the OS interacting with the filter operation.

System	AHRS Version	Mean $\pm$ σ (μsec)	Max-Min
UC32	SQM	1383 $\pm$ 32	1002
UC32	DQA	2704 $\pm$ 41	1529
UC32	DDA	3122 $\pm$ 26	417
OSAVC	SQM	256 $\pm$ 0.9	23
OSAVC	DQM	367 $\pm$ 1.9	62
OSAVC	DQA	445 $\pm$ 2.7	71
OSAVC	DDA	508 $\pm$ 3.1	45
Pico	SQM	200 $\pm$ 5	25
Pico	DQM	337 $\pm$ 9	38
Pico	DQA	467 $\pm$ 13	59
Pico	DDA	490 $\pm$ 16	90
Pi	SQM	1.8 $\pm$ 1.6	50
Pi	DQM	1.6 $\pm$ 1.1	29
Pi	DQA	2.4 $\pm$ 2.3	51
Pi	DDA	3.1 $\pm$ 2.2	50

Table 6.1: The mean $\pm$ standard deviation and maximum difference of latency for the systems under test and each AHRS algorithm.

code generation in which case the OSAVC outperformed the Pico. It isn't clear why this is the case but may be due to an increase in latency due to the use of the SDK as opposed to bare-metal programming. When evaluating the variation in latency, however, the OSAVC performed better than the Pico for all AHRS versions. The OSAVC had less than 0.6% standard deviation over mean for all AHRS variants. The Pico on the other hand, performed four to five times worse than this for all variants. This appears to be due to the bimodal distribution of latency for the Pico, possibly as a result of its optimized math libraries. Although 2.5% latency variation is small, it translates into additional noise in the attitude estimate due to inaccuracy in the integration time constant inside the algorithm.

If we consider that most vehicles will have an update period on the order of milliseconds to tens of milliseconds, both the Pico and the OSAVC conceivably can perform the attitude estimation without consuming the bulk of the time budget allocated to real-time control. This is an encouraging result.

The last system to discuss is the Raspberry Pi. Its behavior is different than the microcontroller systems. The benchmark experiments demonstrate that floating point precision doesn't affect the latency of the algorithms at least within the bounds of the experiments. This is likely due to the fact that the Raspberry Pi is running a 64 bit OS, so all calculations are computed using 64 bit registers regardless of the specified precision (up to 64 bits, of course).

Although the Raspberry Pi is much faster than any of the microcontroller systems—as one would expect due to the difference in clock speed and register width—its latency is not normally distributed. This demonstrates the issue of using a system without a real-time OS. During these studies, the Raspberry Pi was not running other processes besides the OS in the background. If, for example, the Raspberry Pi was performing a background task that was delayed, this would affect the calculation of the attitude in a detrimental way. This is in contrast to the OSAVC, with its deterministic code execution. Put another way, while the Pi has much more computational power and speed than any of the microcontrollers, the OS can't guarantee a fixed latency, whereas bare-metal programming does if designed appropriately *because* there is no OS.

In summary, the OSAVC performed nearly well in terms of latency as the fastest microcontroller studied, the Raspberry Pi Pico, but had smaller latency variation. The UC32 had the worst latency of the all the systems. The Raspberry Pi performed the best by far in terms of latency but the latency distribution when considered relative to the mean was the worst—understandably, as it is not a real-time processor.

Chapter 7

OSAVC Testbed: AGV

In this chapter we discuss the development of a test vehicle used to validate the performance of the OSAVC. we chose to develop an AGV because it can support a wide range of sensors and actuators with only minor impact on vehicle performance. Similarly, it has enough payload capacity to add an SBC and TPU. Finally, a ground vehicle is easier to test without fear of damaging the vehicle or endangering others, when the inevitable mishap occurs.

The other main research objective for the AGV is to evaluate a hybrid LiDAR-vision mapping sensor. This sensor uses a camera and a convolutional neural network (CNN) model that has been trained to recognize specific landmarks in the environment. Once a landmark is detected, a LiDAR sensor is swept across the camera field of view (FOV) to identify the relative location of the landmark to the vehicle. This sensor can be used for SLAM operations (e.g., for use in GPS denied environments) or for autonomous guidance (e.g., to navigate a closed circuit course using the landmarks to delineate the track boundaries). Although this research isn't complete, much of the development is presented here. The steps necessary to complete the research are in Chapter 9.

7.1 Distributed Control Architecture

In [15], we proposed a distributed control architecture for resource constrained autonomous vehicles. In particular, we focused on vehicles constrained by payload capacity, energy storage or both. These constraints place unique requirements on a vehicle. The architecture is demonstrated in Fig. 7.1 and consists of an SBC for guidance and asynchronous tasks, an edge TPU[1] to perform machine learning computations, and the OSAVC for real-time control.

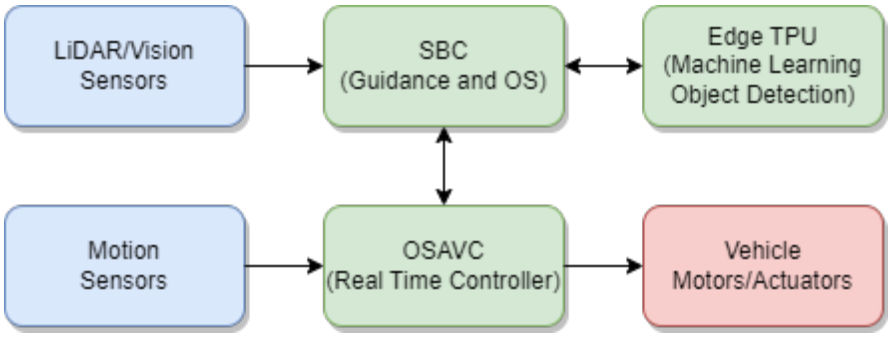

Figure 7.1: The distributed control system we proposed in [15] consisting of the OSAVC (the real-time controller) responsible for navigation, state estimation and control, a single board computer for guidance and other higher order computations as well as hosting an OS, and an edge TPU for machine learning computation.

7.1.1 Single Board Computers

A recent article benchmarked the performance of commercially available SBCs [23]. The results are summarized in Table 7.1. These SBCs have different target markets. The Raspberry Pi 4b for example, is targeted to the educational market, whereas Nvidia models use a GPU to target ML mobile applications. Most SBCs reported in the study have an ARM core for their processor, the notable exception being the Odroid H2+ with an x86 core. The Nvidia TX2 has an ARM core in addition to a 256-core

[1]Edge processing refers to a distributed computing paradigm where processors are located closer to the source of the data.

GPU to enable ML applications. The results from the study were particular to a specific set of video operations and therefore the conclusions are not broadly applicable to general use. Of more interest is the wide range of power consumption (15 to 60 W), weight (45 to 185 g), and cost ($49 to $479).

SBC Model	Clock Speed (GHz)	CPU Cores	CPU	RAM (GB)	Storage Media	Cost ($)	Power (W)	Weight (g)
Odroid XU4	2	8	ARM 32-bit	2	eMMC or microSD	49	20(5V, 4A)	60
Odroid H2+	2.5	4	x86 64-bit	32	eMMC, microSD or SATA	119	60(15V, 4A)	185
Raspberry Pi 4B	1.5	4	ARM 64-bit	8	Micro-SD	75	15(5V, 3A)	45
Nvidia TX2	2	4	ARM 64-bit	8	eMMC	479	57(19V, 3A)	85

Table 7.1: A comparison of SBCs from a video processing benchmarking study [23]. Note the Nvidia TX2 also has a 256-core GPU in addition to its 4-core CPU.

We selected the Raspberry Pi 4b for the architecture because of its powerful CPU—a 1.5 GHz quad-core ARM processor, its large community of users, its Linux OS (and the associated open source software ecosystem), and its affordability. A vehicle adopter is given some latitude with this choice, however. The library we use to communicate between the OSAVC (MAVLink) is compatible with most Linux distributions, so other SBCs with a compatible Linux distribution will likely work with some modification, however, many of the SBC applications we developed use Raspberry Pi specific hardware and configurations. These will need to be ported or modified to work on a different SBC.

7.1.2 Tensor Processing Units

Edge TPUs for ML applications have recently become widely available to the consumer market. There are two notable models: the Intel Neural Compute Stick 2 (NCS2) [25] and the Coral USB Accelerator [8]. Both connect to a host computer using a USB interface, cost less than $100, and are small (65 $\times$ 30 $\times$ 8mm for the Coral USB Accelerator and similar in size to the NCS2). These attributes allow them to be used with compatible SBCs. Again, our choice of the Raspberry Pi 4b does not constrain a new vehicle developer to this SBC as the Coral communicates over USB and the object detection models are compatible with many different Linux distributions. The main difference between the two TPUs is the software interface—the Coral unit is manufactured for Google and has native support for their Tensor Flow Lite ML applications, whereas the NCS2 uses a translation software to convert ML models to a compatible form.

For the proposed architecture we chose the Coral because of its native support of TensorFlow Lite image classification and object detection models.

What follows is a discussion of some of the interesting capabilities enabled by this architecture. The first is using the TPU to help identify landmarks in the environment and the CPU to provide guidance based on that information. The second is onboard optimal sensor calibration. A third capability that we have developed is presented in the next chapter when discussing optimal guidance strategies.

7.1.3 Visual Object Detection

The AGV uses a camera to identify markers that determine the boundaries of a closed circuit. The course markers are cones of different colors to differentiate between the left boundary and the right boundary of the circuit. Through a research project funded by the Google Summer of Code, we developed and trained several machine learning models to identify the cones, including the MobileNet V2 [39] model, the

YOLOv5[2] model [36], and the EfficientDet model [40]. After successfully running these on standard computers, we ported them to the AGV. We first implemented them on the SBC where we were able to get object detection at about 1 frame per second. Fig. 7.2 shows the results of the YOLOv5 model running on the SBC. This model was accurate in identifying cones at a useful inference rate even in shadow. As seen in the figure the model returns the bounding box coordinates and confidence level of the object detection inference. We have been able to compile and test the Efficient Det model on the TPU, offloading the object detection entirely from the CPU and achieving frame rates in excess of 10 frames per second.

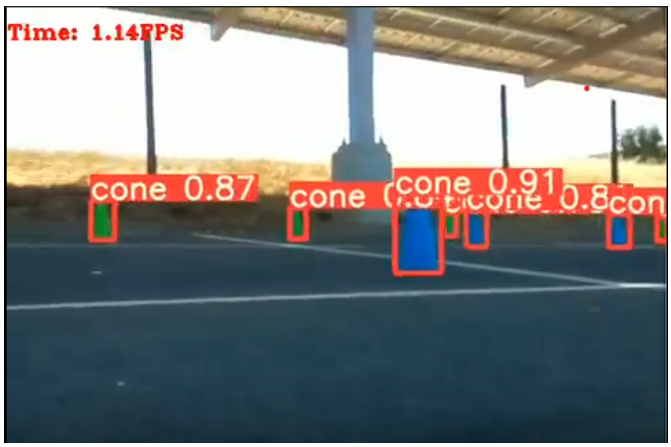

Figure 7.2: Cone detection using the YOLOv5 object detection mode. The images are taken from the AGV. The model returns the bounding box coordinates and confidence level of the detection. The inference frame rate is displayed at the upper left corner of the image. This model was run on the SBC—we have successfully deployed the EfficientDet model to the TPU with significantly faster inference times.

7.1.4 IMU Calibration

A key requirement for autonomous navigation is sensor calibration, in particular the calibration of magnetometer and accelerometer scaling, offset, and alignment variables.

[2]There is some ambiguity whether this is a new model at all or just a revision of the YOLOv4 model because the model is unpublished.

There are numerous algorithms for inertial sensor calibration most relying on fitting a set of data taken while rotating the sensor around all three axes to an ellipsoid and determining the calibration parameters through a least squares fit [18]. Because inertial sensors like magnetometers and accelerometers experience a constant force regardless of orientation when at rest, a set of data sampled slowly—to minimize accelerations—and uniformly during rotations around all three axes should describe a sphere centered at the origin. Rather than perform a direct fit to an ellipsoid, instead we demonstrate in Fig. 7.3 a simulated calibration using a method that iteratively fits the data to a unit sphere [10]. This algorithm is written in python and runs easily on the AGV SBC and computes the calibration parameters without assuming ellipsoidal constraints to the data. We have employed this algorithm successfully on real data taken on the vehicle while rotating it through a random series of orientations. We used the distributed control architecture to sample and store the data, as well as calculate the calibration parameters. Implementing this algorithm as part of vehicle calibration suite is a rich area of future research on the platform.

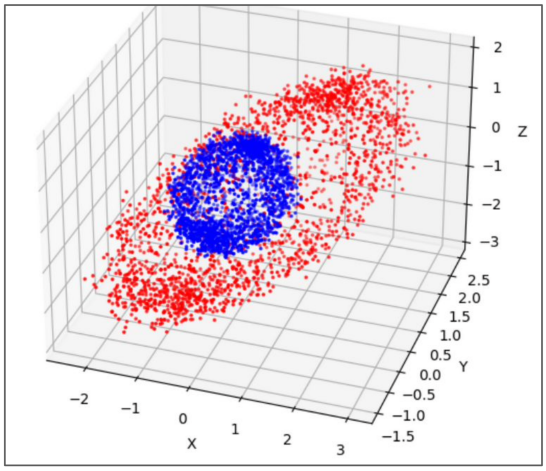

Figure 7.3: IMU sensor calibration simulation. The red dots indicate randomly oriented, mis-calibrated data from an inertial sensor. The blue dots are the calibrated data after performing an iterative fit to a unit sphere.

7.2 Hardware

The platform used as a development testbed is the DFRobot Asurada GPX. This model has been replaced with a substantially similar one, the DFRobot ROB0170 NXP Cup Race Car Chassis [37]. Sold as a kit, it consists of a small (approximately 34 cm long) aluminum chassis that allows for extensive modification due to numerous mounting points. It is driven by dual BLDC rear motors rated for 12 V operation at 930 K_v. A 30 A ESC with BLHeli firmware in bi-directional mode conrols each motor. This means that the motor can be operated both forward and in reverse. A 13:50 gear reduction is applied to increase the torque at the wheels. It uses an Ackerman steering mechanism controlled by a servo motor. It has a wheelbase of approximately 174 mm, and four 65 mm diameter hollow (non-inflated) rubber wheels. It is powered by up to a three cell lithium polymer or equivalent 12V (nominal) battery. The battery and ESCs are located on the lower chassis plane in front of the motors. Fig. 7.4 shows the lower chassis with the steering assembly and the motors installed.

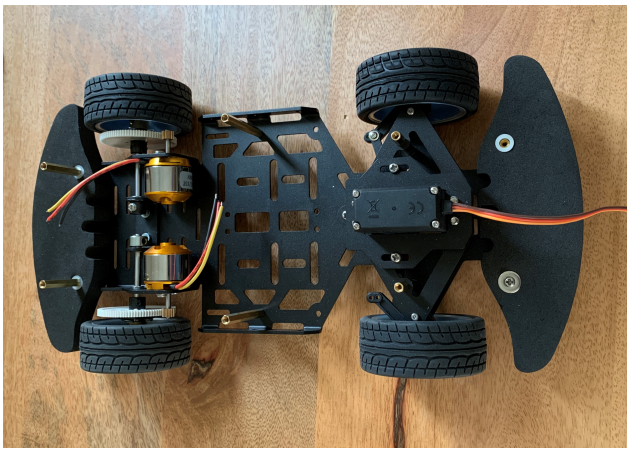

Figure 7.4: View of the GPX Asurada lower chassis. It has independent rear wheel drive with two BLDC motors. The Ackerman steering assembly, driven by a servo motor, is mounted in the front of the chassis.

We designed mounts to house Hall-effect rotary encoders to measure the rear wheel rotation angle and angular velocity and incorporated them into this platform. These sensors sense the angle of magnets mounted on the end of the driveshafts. The steering

rotation is measured by an additional encoder with the magnet mounted concentric to the servo output gear. We also fabricated a small platform to house the OSAVC and a Raspberry Pi4b SBC and one for the sensors and radio control receiver over that. A third platform mounts over the steering servo for a object-detection and mapping module discussed later. An image of the fully assembled vehicle is shown in Fig 7.5.

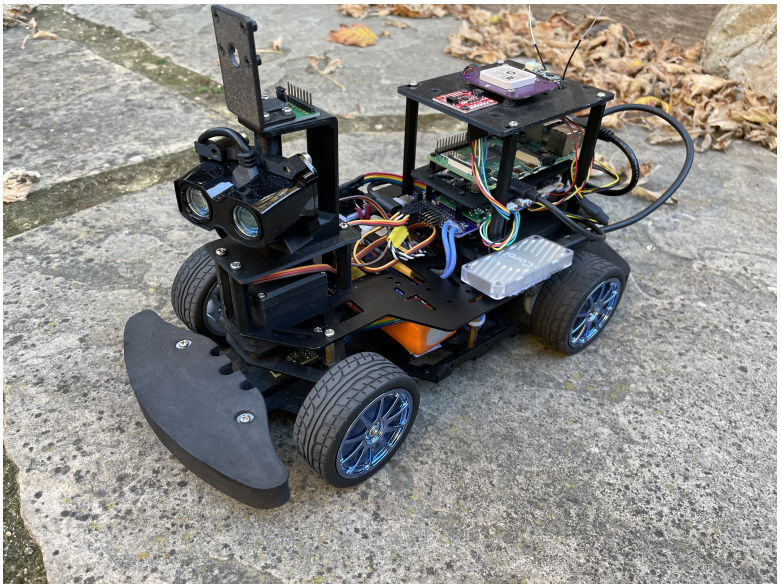

Figure 7.5: AGV ('Sparky')—is a test platform for the OSAVC. Sparky is equipped with a LiDAR and camera sensor (not mounted) used for landmark detection at the front of the vehicle. It has a Coral TPU to perform object identification (silver component mounted on upper chassis plate) connected via USB to the SBC.

The vehicle is equipped with a GPS sensor, IMU, four rotary encoders, and a LiDAR (refer to Table 3.1 for a complete description of these sensors). The vehicle has four outputs controlling the two ESCs for the drive motors, the steering servo, and a servo to control the pointing direction of the LiDAR. The motors are powered from the battery directly; the servo motors are powered by the onboard 5V LDO. The vehicle is also equipped with a serial radio for telemetry communication to a remote ground station, and a radio control receiver for manual control.

The OSAVC is the real-time controller of the overall system, but it is part of a larger distributed control system architecture discussed in more detail in the subsequent

section. The OSAVC communicates over the USB port to a Raspberry Pi4b SBC using the MAVLink protocol. The SBC connects to the Raspberry Pi V2 camera, used to record video and detect obstacles or landmarks. It also connects to the Coral TPU for ML inferencing. The complete block diagram of the AGV system is shown in Fig. 7.6.

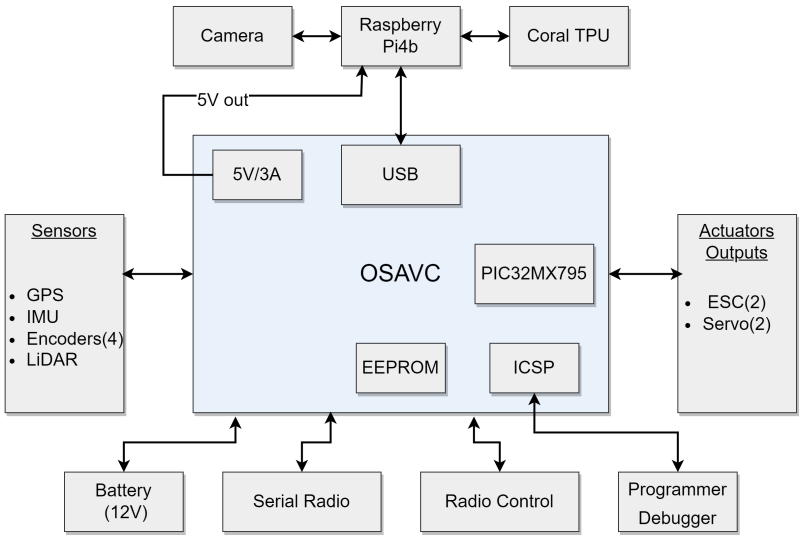

Figure 7.6: AGV hardware block diagram. The PIC32 microcontroller is the heart of the OSAVC and provides all hardware interfaces and peripherals. To avoid complicating the diagram we simply state that it is connected to every element of the OSAVC.

7.3 Vehicle Model

In this section we present the models for the longitudinal and lateral dynamics separately in order to separate the steering control and the motor (velocity) control into distinct controllers. The yaw (steering) rate depends on the rear wheel speed, so first we control the motor velocity to the desired value, and then determine steering rate from the vehicle speed and the lateral kinematic model.

7.3.1 Longitudinal Model

The complete longitudinal model contains all the forces acting in the x axis of the vehicle. These are represented in Fig. 7.7, where:

- m is vehicle mass
- θ is the inclination of the vehicle
- g is gravity
- F_{aero} is the force due to aerodynamic drag
- R_f is the front tire rolling resistance
- R_r is the rear tire rolling resistance
- F_{xf} is the force of the front tire in x direction
- F_{xr} is the force of the rear tire in x direction

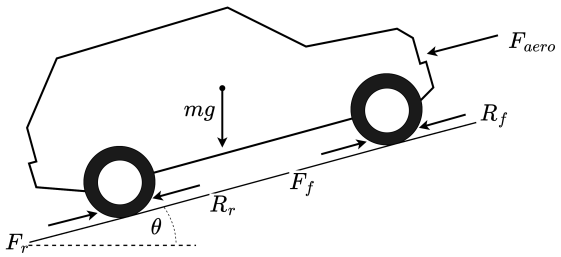

Figure 7.7: Force body diagram in the longitudinal direction of a vehicle.

The equation of motion in the longitudinal direction is:

$$m\ddot{x} = F_{x,r} + F_{x,f} - R_r - R_f - F_{aero} - mg\sin(\theta) \tag{7.1}$$

Aerodynamic drag is due to the effect of the wind and velocity acting on the vehicle. In general it is expressed as:

$$F_{aero} = \frac{1}{2}\rho C_d A_f (v_x + v_{wind})^2 \tag{7.2}$$

where ρ is the density of air, C_d is the drag coefficient, A_f is the cross-sectional area of the vehicle, $v_x + v_{wind}$ is the effective wind speed in the x direction. The drag coeficient can be modeled using computational fluid dynamics or measured empirically.

Rolling resistance describes the resistance to motion of the wheel. It opposes the direction of movement and is due to the deformation of the tire. The tire deforms due to the normal force caused by the weight of the vehicle and the torque at the wheel. As the tire rotates the vehicle compresses the tire. Some of this energy is returned as the wheel continues to rotate, however, some is lost depending on the inelasticity of the material in the form of heat. Rolling resistance is typically modeled as proportional to the normal force of the vehicle or,

$$R_r + R_f = f(F_{z,r} + F_{z,f}) \tag{7.3}$$

where f is the rolling resistance coefficient, $F_{z,r}$ and $F_{z,f}$ are the rear and front normal forces on the respective wheels. The normal forces are due to several factors including the mass of the vehicle, the center of gravity of the vehicle, the grade of the road, the moment of inertia due to wind resistance, and the moment of inertia due to acceleration/deceleration.

The $F_{x,r}$ and $F_{x,f}$ terms are friction forces of the tires acting on the ground. These are the forces that cause the vehicle to move. These forces are complex and in general are determined experimentally to depend on the slip ratio of the tire, the normal force of the tire, and the friction coefficient of the interface between the tire and the road. The slip ratio σ_x is defined to be the longitudinal velocity of the axle minus the calculated rotational velocity equivalent $r_{eff}\omega$, where r_{eff} is the effective radius of the tire and ω

is its angular velocity:

$$\sigma_x = \frac{r_{eff}\omega - v_x}{v_x} \quad \text{during braking} \tag{7.4}$$

$$\sigma_x = \frac{r_{eff}\omega - v_x}{r_{eff}\omega} \quad \text{during acceleration} \tag{7.5}$$

$$\tag{7.6}$$

The slip ratio is a nonlinear function of the tire and road surface and depends on the material used in the tire, the road surface itself, and the construction of the tire. For automobiles tires one empirical model used to describe the nonlinear behavior is the "Pacejka Magic" model [29]. If the slip ratio is small ($\sigma_x < 0.1$), and the normal forces are constant (i.e., during small accelerations), the friction forces can be modeled as a linear function of the slip ratio:

$$F_f = C_f \sigma_x \tag{7.7}$$

$$F_r = C_r \sigma_x \tag{7.8}$$

for the front and rear tires respectively. C_f and C_r are known as the stiffness coefficients of the tires.

For a complete treatment of the longitudinal dynamics refer to [35]. This treatment is, of course, oriented towards the dynamics of a full sized automobile that can travel at speeds in excess of 100 kph. The question of interest is whether these equations scale to the AGV. It is apparent that for the real automobile the aerodynamic drag—a function of the apparent wind speed squared—is the dominant term in the forces on the vehicle. The tire models are also empirically based on automobile tires and unlikely to translate with any fidelity to the wheels of the AGV. Finally, an automobile may use a combustion engine and gearbox so a drivetrain model is of little relevance to the AGV which uses BLDC motors and fixed ratio gearing.

These issues are addressed in the literature [20] for a much smaller scale vehicle, a 1:43 scale race car. They model the longitudinal dynamics forces using a friction model of a brushed DC motor and the rolling resistance and aerodynamic drag:

$$F_x = (C_{m1} + C_{m2} v_x) d - C_r - C_d v_x^2, \tag{7.9}$$

The authors don't define the meaning of C_{m1} or C_{m2} or d—these are evidently the parameters of the DC motor model—however, C_r is the rolling resistance coefficient and C_d is the coefficient of drag. This work also provides models for the lateral forces on the tires taken from an earlier tire modeling paper [2] that is substantially similar to [29] with similar authorship. Finding these coefficients requires an empirical study of the torque required to maintain a constant speed over a range of v_x achievable by the AGV and a reduction of the data using a least-squares regression. Note that similar to the AGV, the vehicles in [20] are rear wheel drive with no braking other than engine braking so the forces on the front wheel $F_{f,x}$ are ignored. Thus, Eq. 7.9 is equivalent to Eq. 7.1 when the vehicle is on flat ground ($\theta = 0$).

We present this material as important background to modeling the full vehicle dynamics. It is beyond the scope of this dissertation to find the parameters of such a model. In fact, it is not necessary to find a full dynamic model to achieve high performance with the AGV. In the development of the speed controller below we treat the longitudinal dynamics as a BLDC motor, however, a minimum to achieve high performance with the AGV is to develop a model of the tire-road friction coefficient in order to ensure the AGV remains in the small slip regime. This presents an interesting area for future research.

7.3.2 Lateral Model

The so-called bicycle model is commonly used to represent the lateral kinematics of an automobile, shown in Fig 7.8. It simplifies the geometry of a real car by considering only half of a vehicle with front wheel steering. Referring to the figure, the wheelbase is divided around the center of rotation, L_f and L_r. The steering angle is denoted δ_f. The side slip angles, α_f and α_r represent the direction of travel for the front and rear tires, respectively. Note that for slow speeds (≤ 5 m/sec) it is safe to assume that the side slip angles are small and the velocity is in the same direction as the wheels [34]. The vehicle rotates around its center of mass with an angular velocity, ω. The normal

forces on the tires are denoted as F_f and F_r. The position of the vehicle is referred to the center of the front tire in Cartesian coordinates of the inertial frame.

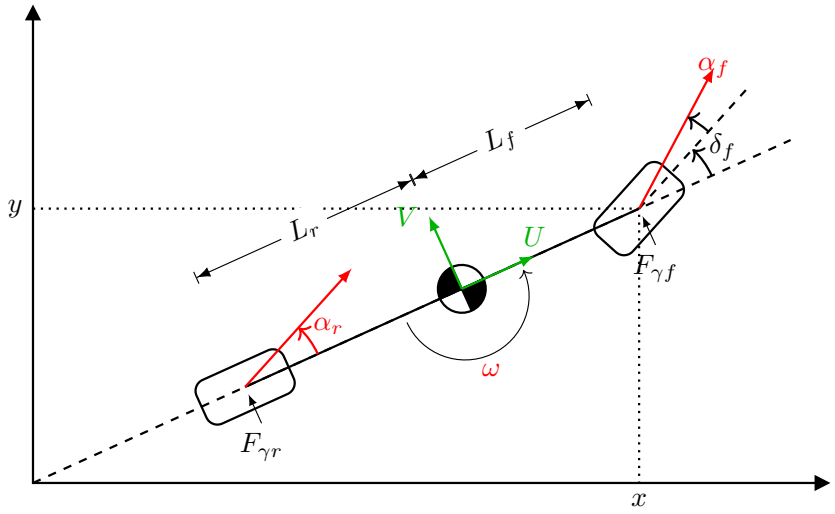

Figure 7.8: The bicycle model for defining the AGV kinematics.

For the AGV, therefore, the position of the car is referenced to the center of the front axle. The steering angle δ is the average of the left and right wheel angles. Note that for an Ackerman steering mechanism [34] these angles in general are not the same. The steering mechanism is designed such that both wheels rotate around a common center, thus enabling slip-free steering. Fig. 7.9 describes the geometry of this mechanism.

For the purposes of this dissertation, we ignore the full lateral dynamics of the vehicle. For basic autonomy we can operate the vehicle in the slow speed regime where the vehicle conforms to the kinematic constraints. For higher performance, at a minimum we would need to determine the tire model coefficients defined in [29] as done in [20], on their work on 1:43 scale model cars.

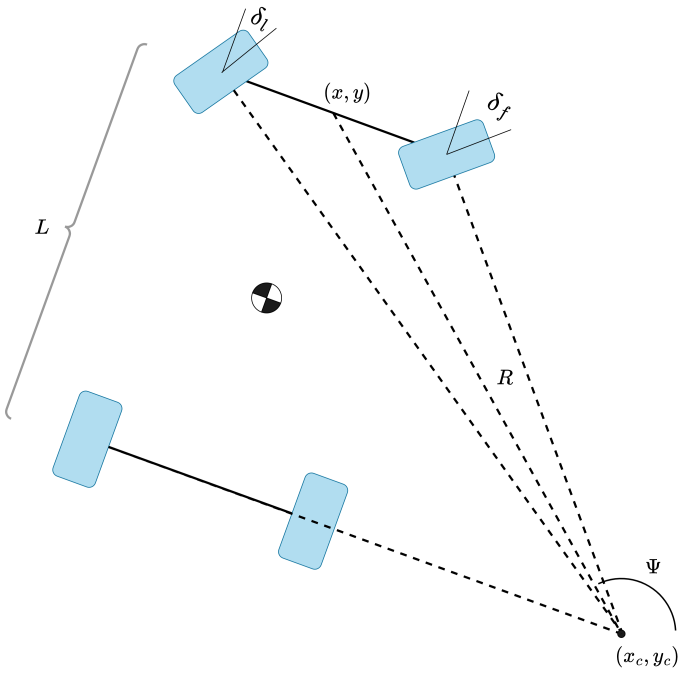

Figure 7.9: Ackerman steering geometry. The steering mechanism is designed such that all the wheels orbit around a common origin. The steering angles δ_l and δ_f are not equal. R is a function of $\delta = 1/2(\delta_l + \delta_r)$.

7.4 Odometry Model

The AGV uses the SQM variant of the complementary filter presented in Chapter 5 for attitude estimation. Recall that this is the single-precision, hand-coded, quaternion implementation. However, we also need an odometry model to estimate the position of the AGV over time. Odometry uses the kinematic model along with the measured speed of the rear wheels to compute the vehicle pose (i.e., the position and orientation[3]).

Referring to Fig. 7.8 and Fig. 7.9 we compute the radius of the turn, R, the angular velocity, ω, and the center of rotation (x_c, y_c) from the steering angle, $\delta = 1/2(\delta_l + \delta_r)$,

[3]Actually we already know the orientation from the AHRS filter—the z component of attitude—but we also can estimate it from the kinematic model.

the heading angle,Ψ, and the wheelbase, L:

$$R = \frac{L}{\sin\delta} \tag{7.10}$$

$$\omega = v/R \tag{7.11}$$

$$x_c = x - R\sin\Psi \tag{7.12}$$

$$y_c = y + R\cos\Psi. \tag{7.13}$$

$$\tag{7.14}$$

Given a small time differential, dt, we calculate the new vehicle state from the old state given that the vehicle orbits around the same center by ωdt radians:

$$x_c = x_{k+1} - R\sin(\Psi_k + \omega dt) \tag{7.15}$$

$$\implies x_{k+1} = x_c + R\sin(\Psi_k + \omega dt) \tag{7.16}$$

$$= x_k - R\sin\Psi_k + R\sin(\Psi_k + \omega dt) \tag{7.17}$$

$$y_{k+1} = y_k + R\cos\Psi - R\cos(\Psi_k - \omega dt) \tag{7.18}$$

$$\Psi_{k+1} = \Psi_k + \omega dt, \tag{7.19}$$

where $(x_{k+1}, y_{k+1}, \Psi_{k+1})$ is the new state, and (x_k, y_k, Ψ_k) is the old state.

Odometry is used to estimate the vehicle position in the absence of an absolute measurement, such as a GPS. On its own, however, it quickly accumulates errors to due to sensor noise and wheel slippage so its absolute positioning is poor. Combined with a GPS, however, it still serves a useful purpose—namely, it can interpolate between position updates. This enables less frequent GPS updates, reducing the bandwidth load on the processor and overall power use. An inexpensive GPS is typically limited to 10 Hz update rates and most come with a default rate of 1 Hz. For vehicles traveling at velocities significantly greater than 1 m/s, for example, this can lead to significant position uncertainty and poor control. For example, a vehicle moving at 10 m/s will travel one meter between update rates. A vehicle that needs to follow a strict trajectory—for example a race car—will likely require tighter control to meet its mission requirements.

Odometry combined with GPS allows for high bandwidth control and low uncertainty in absolute position[4].

Note that we have other means to estimate ω, namely from the bias-corrected gyro readings of the IMU returned by the AHRS algorithm. This redundant information leads to an intriguing possibility. We may be able to form a better estimate of ω by combining the two measurements in a complementary filter. This would enable higher precision odometry and potentially extend the time between GPS updates further. This is currently out of scope for this research but should be considered for the future.

7.5 Speed Control

In this section we present a simple speed controller for the AGV. This controller has two components, the first is the lower level controller that converts a pulse-width modulated signal into the angular velocity of the motor (ω [rad/sec]). This in turn is converted to vehicle motion by $v_x = r_{eff}\omega$, where r_{eff} is the effective radius of the tire. In the case of the AGV this controller is a physical component, the ESC discussed earlier. The second controller is a higher level controller that determines the acceleration necessary to control the vehicle to a desired velocity and produces the input to the ESC in the form of a PWM duty cycle. The block diagram of the system with controller is given in Fig. 7.10.

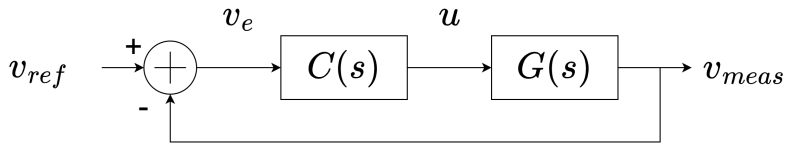

Figure 7.10: Block diagram of the velocity controller and plant system.

One approach to modeling the vehicle plant is to consider it as an extension of the motors, that is, a motor connected to the wheels of the vehicle which are subject to

[4]Also, it can be put into a complementary filter form.

rolling resistance and aerodynamic forces. We assume a transfer function for a DC motor in the Laplace domain that relates the angular velocity $\Omega_r(s)$ to its voltage input $V(s)$:

$$G(s) = \frac{\Omega_r(s)}{V(s)} = \frac{b}{s+a}. \tag{7.20}$$

The motors provide no means for measuring ω_r, but we can measure wheel velocity with the encoders mounted on the drive shafts. We can try to model the motor wheel system using the same transfer function. Using the autoregression with external input (ARX) procedure (detailed in Appendix B) we estimated the a and b parameters both motors of the AGV. For this experiment, the AGV wheels are allowed to rotate freely, that is, the vehicle is suspended above the ground surface. The results are shown in Fig. 7.11. We plot the one step lookahead error to quantify the error in one update interval, which in this experiment is 0.01 sec. The error plot has a zero mean suggesting a good fit to the data but there are noticeable spikes. These spikes correspond to the motor switching between acceleration and deceleration and may be due to the backlash in the gears between the motor shaft and wheel axle. Fig. 7.12 shows a step test performed on both motors and the modeled values from the ARX system identification. Although the motors behave similarly, the noise on the encoders is substantial. This is likely due to a small misalignment of magnet on the axle to the sensor, compounded by taking the differential between subsequent angle measurements to compute velocity. We observed that the magnitude of the error changed as the alignment of the magnets to the sensor changed, however, this design doesn't allow us to adjust both magnets independently. Therefore, we attempted to equalize the error between the two sensors. In practice, therefore, a low pass infinite impulse response (IIR) filter is needed to smooth the ripple, thus introducing more latency to the system. One area of future work is to design a sensor mount for the encoders that allows for adjustments to minimize the alignment error of each sensor independently.

The low pass IIR filter is a simple recursive filter which weights the difference between the current measurement and the previous value, $x_k - y_{k-1}$ by a factor, α, and adds it to the previous value as shown in Eq. 7.22. Smaller values of α shift the cutoff

frequency of the filter towards lower frequencies.

$$y_k = y_{k-1} + \alpha(x_k - y_{k-1}), \text{ or} \tag{7.21}$$

$$y_k = (1-\alpha)y_{k-1} + \alpha x_k \tag{7.22}$$

One important finding from this modeling is that the motor-wheel system has a rise time on the order of 10 msec. This is at least as fast as the loop time needed to command the AGV, so the ESC is more than adequate for the low level control. Not evident in the plots, however, is the lag time between when the command was issued and when motion is first detected at the wheel. This lag is significant, on the order of 25 msec, however, the 'zero-order hold,' (ZOH) or digital delay, is 20 msec for this experiment and comprises most of the delay. A simple modification to the OC module to reduce the timer period can reduce the ZOH as most ESCs can be driven at 200 Hz or faster, depending on the specfication of the ESC. These delays and the need for a low pass filter on the velocity signal introduces latency to the system, degrading its performance somewhat. The other (perhaps obvious) finding is that with one real pole at ($s = -a$) the system is stable for $a > 0$.

This ARX procedure may be repeated with the vehicle on flat ground in order to develop a simplified model for speed control. Note that we are essentially designing cruise control for the vehicle in order to achieve basic autonomy. In the low speed regime where v_x^2 is small we assume the aerodynamic forces are neglible and model the plant using Eq. 7.20.

A common approach for a speed controller is a proportional-integral, or PI, controller. The proportional term dominates the dynamic response of the controller while the integral term ensures zero steady state error. The continuous form of the PI controller is

$$v_e = v_x - v_{ref}, \tag{7.23}$$

$$u = \dot{v}_x = -k_p v_e - k_i \int_0^t v_e dt, \tag{7.24}$$

where u is the acceleration command sent to the velocity controller, v_x is the forward velocity, v_{ref} is the desired velocity, k_p is the proportional gain constant, and k_i is the

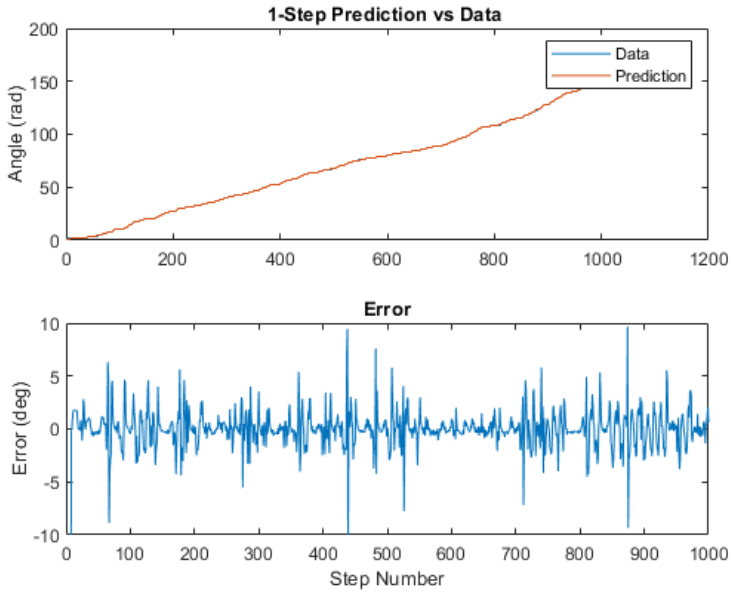

Figure 7.11: The ARX system identification procedure for the left motor of the AGV. The upper plot shows the one step look ahead angle measurement and the predicted value. The lower plot shows the error between the two. The spikes occur when the acceleration changes sign, probably due to backlash in the transfer gears.

integral gain constant. The transfer function of the controller is

$$C(s) = k_p + \frac{k_i}{s} \tag{7.25}$$

and the closed loop transfer function is given by

$$\frac{C(s)G(s)}{1 + C(s)G(s)} = \frac{bk_p s + k_i}{s^2 + (a + bk_p)s + k_i}. \tag{7.26}$$

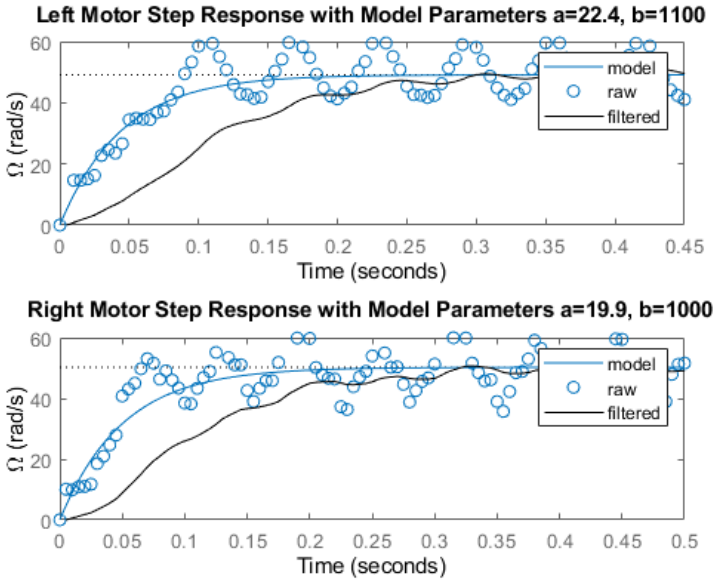

Figure 7.12: The modeled versus actual velocities of the AGV motors during a step test. Also plotted is the filtered velocity. Note the significant encoder noise on the velocity. This is likely due to misalignment of the magnets to the sensors.

7.6 Heading Control

In this section we discuss the heading control, that is, given a rear wheel speed and a reference orientation, the control of the steering angle to orient the car in the desired direction. The approach implemented below is adapted from the literature for attitude control of a quadcopter using the quaternion error product [13]. This approach is attractive because the AHRS filter provides the attitude of the AGV in quaternion form in the inertial reference frame q_v. Given a desired reference angle Ψ_{ref} we can compute the reference quaternion as:

$$q_{ref} = [\cos(\Psi_{ref}/2), 0, 0, \sin(\Psi_{ref}/2)]^T. \tag{7.27}$$

The error quaterion is defined as the quaternion product between q_{ref} and q_v^*, where q_v^* is the quaternion tranpose of q_v.

$$q_{err} = q_{ref} \otimes q_v^*. \tag{7.28}$$

If we assume flat ground ($q_{err}(2) = q_{err}(3) = 0$) then the heading control u_h is given by:

$$u_h = \text{sign}\ (q_{err}(1))k_p q_{err}(4) \tag{7.29}$$

where sign($q_{err}(1)$) indicates which direction to turn when crossing over $\Psi_{ref} = \pm 0$ or $\Psi_{ref} = \pm\pi$, and k_p is a proportional gain term.

Using Eq. 7.29 with $k_p = 1$ and applying the odometry model from the previous section, we can simulate the vehicle motion starting from Ψ_0 and driving towards a specific Ψ_{ref}. Fig. 7.13 shows the vehicle trajectories for Ψ_{ref} uniformly spaced around the compass. The resulting trajectories demonstrate that the algorithm works to orient the vehicle to a specified angle. Moreover, the algorithm is computationally inexpensive requiring two trigonmetric calculations (to compute q_{ref}, three float multiplies, and one float addition.

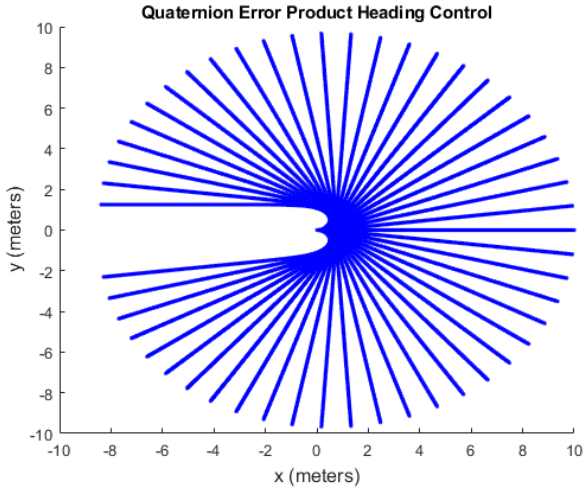

Figure 7.13: Simulation of AGV motion for angles uniformly spaced around the z axis. The vehicle is initially located at the origin, oriented in the $+x$ direction, then uses a proportional control command based on the quaternion error product and a constant velocity to navigate.

7.7 Waypoint Traversal

An autonomous mission ordinarily directs a vehicle to specific locations, rather than simply orient it in a fixed direction. This is known as pursuit guidance [19]. To do this, we must first compute the orientation of the waypoint with respect to the vehicle, then convert it into the reference direction. In other words, given a waypoint at coordinates (x_{wp}, y_{wp}) and the vehicle at (x_v, y_v), we compute the orientation (Ψ_{ref}) of the waypoint relative to the vehicle as

$$\Psi_{ref} = \text{atan2}((y_{wp} - y_v), (x_{wp} - x_v)) \tag{7.30}$$

and then construct the reference quaternion of Eq. 7.27. We then compute the error quaternion and compute the steering command from Eq. 7.29. Fig. 7.14 demonstrates this process using randomly generated waypoints. As in Fig. 7.13, the vehicle is initially oriented along the $+x$ axis and located at the origin. As the vehicle reaches a waypoint within a specified radius, a new waypoint is generated. This process continues for five waypoints, at which time the simulation ends.

There is an alternative approach to heading control also using quaternion algebra and fixed waypoints. In this approach, we compute a vector $\mathbf{V_i}$ pointing towards the waypoint from the vehicle, rotate the vector into the body frame using quaternion rotation, then compute the angle Ψ_{wp} to the waypoint in the body frame. The heading control is calculated with a proportional gain k_p multiplied with this angle with opposite sign. In summary,

$$\mathbf{V_i} = (x_{wp} - x_v, y_{wp} - y_v, 0)^T \tag{7.31}$$

$$q_{wp,b} = q_v \otimes \text{p}(\mathbf{V_i}) \otimes q_v^* \tag{7.32}$$

$$\mathbf{V_b} = \mathbf{vex}(q_{wp,b}) = (x_{wp,b}, y_{wp,b}, 0)^T \tag{7.33}$$

$$\Psi_{wp} = \text{atan2}(y_{wp,b}, x_{wp,b}) \tag{7.34}$$

$$u_{wp} = -k_p \Psi_{wp}. \tag{7.35}$$

Both algorithms work equally well in simulation, the only difference lies in the computation time, which may favor the first approach slightly due to computing the quaternion error product rather than a quaternion rotation.

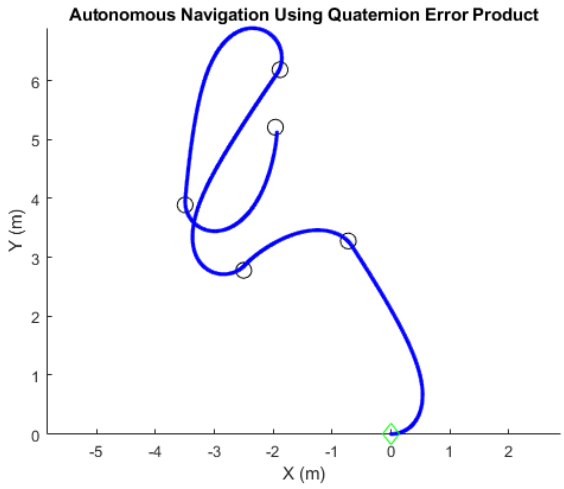

Figure 7.14: Simulation of AGV motion through a series of randomly generated waypoints. The vehicle is initially at the origin, oriented in the $+x$ direction. Each waypoint is represented by an open circle.

7.8 Experimental Validation

In this section we present experimental results to validate some of the theoretical treatments of the previous sections. We have already shown the system identification of the motor models, and discussed the need for filtering the wheel angular velocity measurements. For the following plots, we implemented a low pass IIR filter with an α parameter of .03, which smoothed the periodic ripple in the measuremets at the cost of about 200 msec delay. We then developed the inner controller for the vehicle velocity at the rear wheels. Fig. 7.15 shows results for the PI controller. The controller uses a proportional gain (k_p) of 80 for both plots, but an integral gain of $k_i = 20$ in the first plot, and a gain of $k_i = 80$ for the second plot. The k_p parameter was chosen to be responsive enough with minimal wheel slippage when performing a step test.

We increased integral gain iteratively to achieve zero steady state error quickly, in this case only 2-3 seconds versus about 8 for the initial controller. All of the experimental data was recorded using the SBC on the vehicle—demonstrating the usefulness of the distributed architecture.

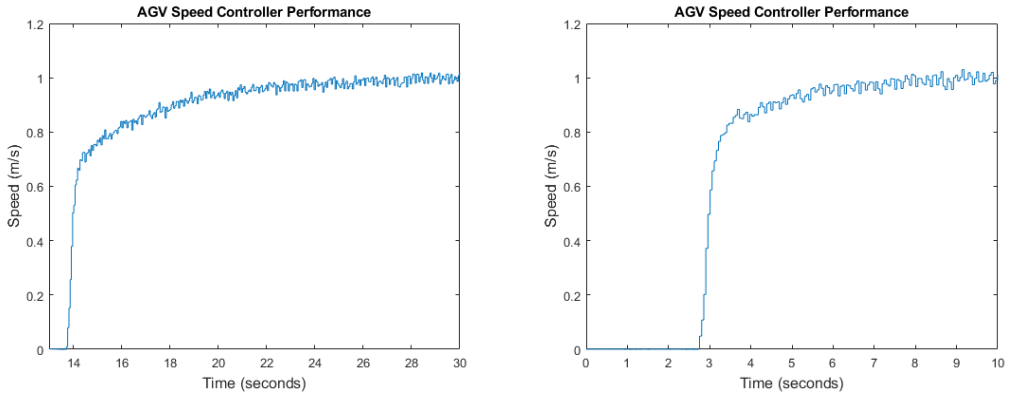

Figure 7.15: Velocity control of the AGV using the PI controller. The left hand image has $k_p = 80$ and $k_i = 20$. This tuning results in a relatively slow time to reach steady state, about 8 seconds. Therefore, we increased the integral gain iteratively to achieve the results in the right hand plot. The final parameters are $k_p = 80$ and $k_i = 80$.

Once the inner control loop was working, we implemented the heading controller and waypoint guidance algorithm. Rather than rely purely on the odometry model to determine the heading of the vehicle, the algorithm corrects the estimated heading using the attitude estimate from the AHRS filter after each odometry computation. Thus, the vehicle can orient itself even indoors enabling autonomous missions. The heading controller is a pure proportional controller, essentially always pointing the wheels in the direction of the next waypoint. This is an implementation of pursuit guidance [19], with multiple fixed waypoints. A simulated mission using five waypoints and a real mission with the same waypoints is shown in Fig. 7.16. The waypoints are the black diamonds in the plots. Initially, the vehicle is oriented south, that is, in the $-y$ direction. Both the simulation and the data demonstrate that the vehicle is able to orient itself properly and head for each waypoint in turn. When the vehicle is within a certain distance of

the current waypoint, in this case set to 1 meter, the algorithm loads the next waypoint. The mission ends when there are no more waypoints.

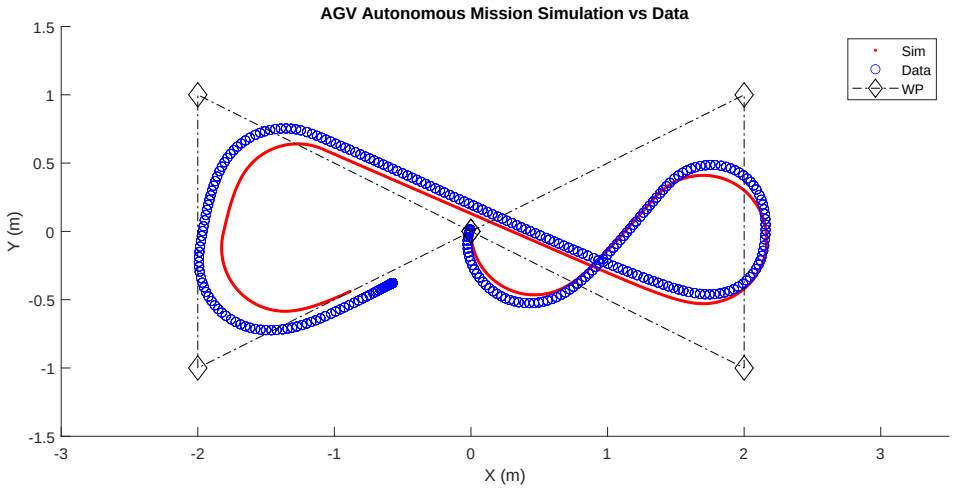

Figure 7.16: Simulation of an AGV mission and experimental validation. The initial orientation is in the $-y$ direction (south). The comparison suggests that the vehicle kinematic parameters may need refining but is nevertheless reasonable. The data plot was taken from an indoor mission using odometry and attitude estimation for navigation.

Fig. 7.17 is the same experimental mission, but in this case the vehicle is oriented in different directions at the start. This figure demonstrates the robust nature of the attitude estimation to orient the vehicle.

We performed one additional indoor experiment to evaluate the quality of the odometry measurements. For this experiment the vehicle performed a simple two waypoint mission, after which we manually drove it back to the start position. The total mission length was approximately 8 meters and repeated 10 times. we define the mission error to be the difference between the expected distance from the final waypoint (0.5 meter) and the actual distance at the end of the mission. The mean rms error and standard deviation after the missions was 0.48 ± 0.13 meters. If the odometry errors were normally distributed then we could expect the variance to be proportional to the length of the mission. Of course, the errors are not normally distributed due to the non-linearity

of the odometry model, however, the variance nevertheless increases with longer missions. This highlights the importance of having an absolute reference to position, e.g., the GPS sensor. For missions in the absence of GPS information, the vehicle must map its location relative to the local landmarks using a localization algorithm such as SLAM.

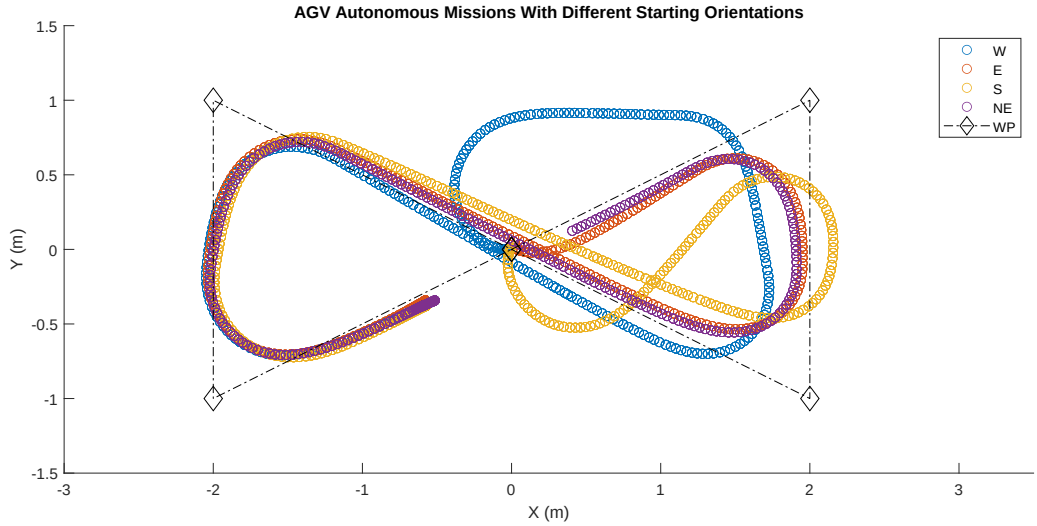

Figure 7.17: Autonomous missions with the AGV in different initial orientations. Each mission uses the same waypoints, shown as black diamonds. These missions used odometry and attitude estimation for navigation.

7.9 Conclusions

The AGV proved to be an ideal test bed to showcase the capability of the OSAVC and the overall control system architecture. It is also an excellent starting point for future research into many different areas, such as optimal time trajectory tracking, GPS-denied navigation, autonomous mapping and many other possibilities.

Chapter 8

Use Cases

In this chapter we detail three additional use cases that utilize the OSAVC architecture, an ASV, a quadcopter UAV, and a fixed-wing UAV.

8.1 Autonomous Surface Vessel

The ASV is shown in Fig. 8.1. It is designed to autonomously map a marine environment, shown here mapping the depth of a freshwater pond. The research goal of the ASV is to determine an optimal (in terms of energy expenditure) trajectory for mapping an unknown environment. The aim is to improve upon a traditional mapping technique known as ordinary kriging [44], which is an optimal method for estimating a field from a few discrete measurements using a form of Gaussian process regression. The main issue with this method is that its complexity grows as $O(n^3)$, where n is the number of observations and therefore is unsuitable for resource-constrained vehicles relying on an SBC for mapping and guidance computation. In addition to the field estimation problem, the other research goal is to determine an optimal path through the environment based upon the uncertainty of the field estimation.

Figure 8.1: The ASV mapping the depth of a freshwater pond in Santa Cruz, California.

8.1.1 Hardware

The ASV is a catamaran vessel with twin BLDC motors and rudders for propulsion and steering. Each motor is controlled independently with an ESC by an OC module. The rudders are connected to a single servo that is controlled by an additional OC module. The boat uses the Max32 development board and the OSAVC I/O rev 1.2 daughter board instead of the integrated OSAVC board.

The ASV employs three of the sensors for navigation: GPS, IMU, and a rotary encoder. These all use the sensor drivers from the OSAVC code repository. The encoder is used to measure the servo angle of the rudders, the GPS provides absolute position and velocity, and the IMU provides attitude. The boat uses an AHRS algorithm similar to the one used to benchmark the OSAVC but implemented by the developer of the boat and placed in the OSAVC repository as a contributor.

Like the AGV, the ASV uses the Raspberry Pi4b SBC for its guidance computer and communicates via USB using the MAVLink communication protocol.

In addition to the common hardware and firmware, the developer of the ASV (Pavlo Vlastos) also implemented some custom additions. The main one to mention here is a

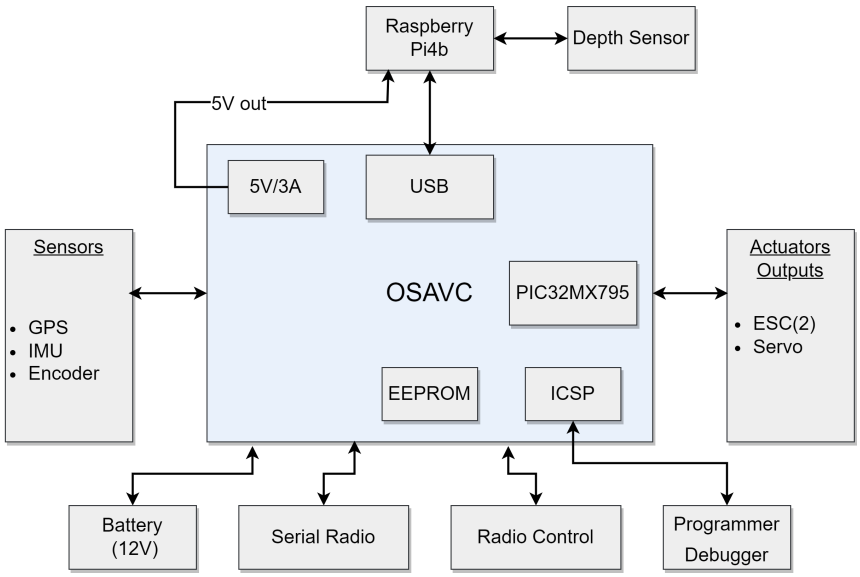

Figure 8.2: ASV hardware block diagram.

EMO hardware switch that disables the motors in case of emergency. Additionally, the he uses a sensor not in the repository—a sonic depth gauge used to map the ocean or lake floor. This sensor connects to the SBC directly. The block diagram of the ASV is in Fig. 8.2.

8.1.2 Results

To make the ordinary kriging method more computationally tractable, we introduced a method known as partitioned ordinary kriging [41]. Fig. 8.3 shows the theoretical results of this method against the true field and two other estimates. This method reduces the overall complexity of a field by subdividing it into smaller partitions and only updating the field estimate within the partition. We deployed it to the SBC to demonstrate feasible use in the field [43]. Vlastos introduced an optimal search method using the variance of the field estimate and implemented it on the ASV in his PhD disssertation [42]. More information regarding the ASV and these algorithms can be found there as well.

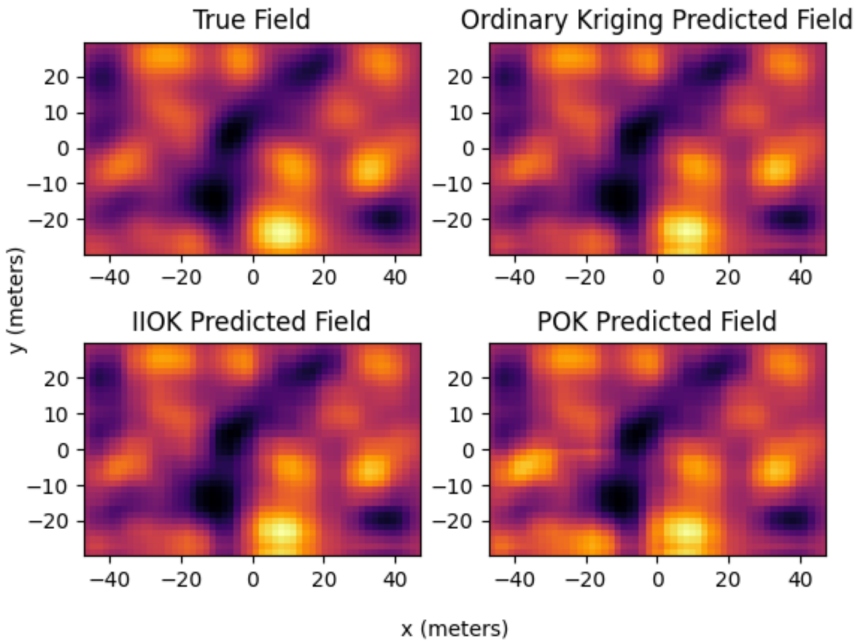

Figure 8.3: The method of partitioned ordinary kriging (POK) against the true field, the field estimate from the ordinary kriging (OK) method, and an iterative ordinary kriging method(IIOK). We demonstrated that POK had equivalent accuracy but improved the computational load significantly.

8.2 Quadcopter UAV

The next vehicle using the OSAVC architecture is a quadcopter. This vehicle is designed to localize itself in environments where GPS is either unavailable or intermittent. The research goal is to demonstrate a method to identify features in the landscape using the TPU and a monocular camera from a pretrained model. The vehicle has a map where these landmarks are geo-referenced to GPS. The source of the map can be taken from existing imagery or mapped and geo-referenced prior to the mission. The vehicle locates itself in the environment by comparing its current pose relative to two or more landmarks that it identfies in flight. Fig. 8.4 is an image of the vehicle during its hardware development.

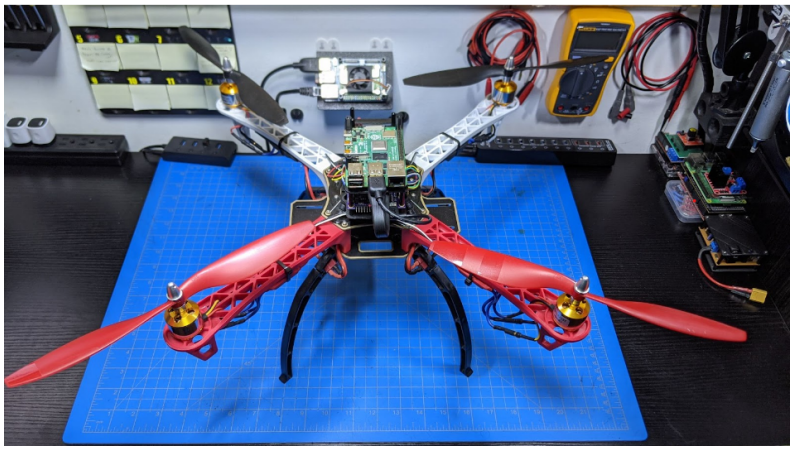

Figure 8.4: Quadcopter UAV hardware. The OSAVC is mounted underneath the SBC. Not visible is the GPS and IMU sensor.

8.2.1 Hardware

The quadcopter consists of a commercially available kit that includes the frame, power distribution board, ESCs, motors, and propellers. The motors are 930 K_v rated BLDCs and powered by a 12 V LiPo high discharge battery. Adapted to the frame is a custom module containing the OSAVC, the sensors, the radio control receiver, the serial radio, and the SBC.

The vehicle is equipped with IMU and GPS sensors. The next step in the development will incorporate a barometer for altitude measurements, a monocular camera for landmark detection, a LiDAR for range measurements to the landmark, and a TPU to run the landmark detection model.

With the exception of the barometer, all the sensors and devices listed above use the drivers from the OSAVC repository. The hardware block diagram is shown in Fig. 8.5.

8.2.2 Status

The quadcopter is under development. The current effort is to develop the flight control algorithms onto the OSAVC. The controller code and the AHRS algorithm have been adapted for use from the AGV examples. It currently has all the navigation sen-

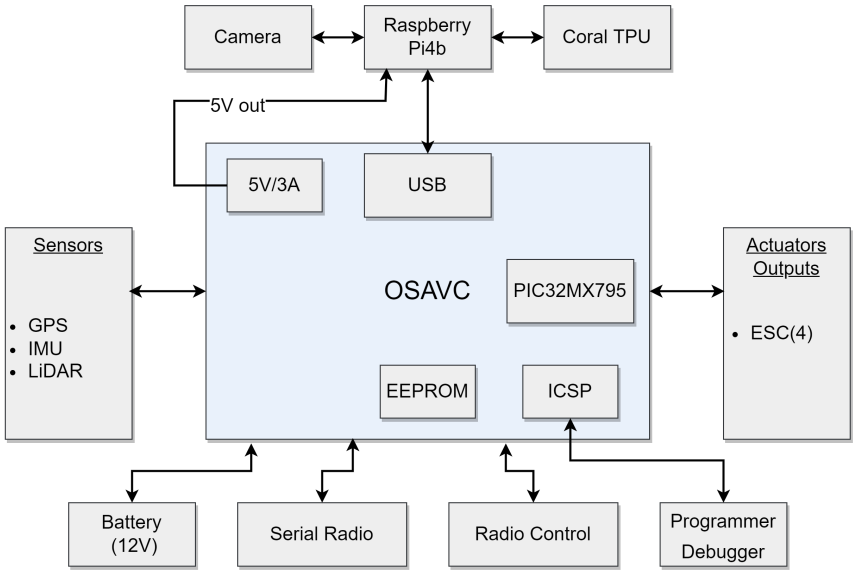

Figure 8.5: Quadcopter UAV hardware block diagram.

sors and the AHRS attitude estimation algortithm functioning. We plan to follow the approach in Fresk, et al. [13], using cascaded proportional controllers based on the quaternion error product, as it matches with the AHRS quaternion output. This approach looks promising in simulation but doesn't have experimental validation. One intermediate research goal is to provide this validation.

8.3 Fixed-Wing UAV

The last platform that we plan to equip with the OSAVC architecture is a fixed-wing UAV. This UAV is Young Wang's master's project from the Autonomous Systems Laboratory. The goal of the UAV is to provide an inexpensive, easy to fabricate UAV. A second team of engineering students are adapting the OSAVC to create a research platform for students to evaluate flight control algorithms.

8.3.1 Hardware

The UAV is designed entirely out of foamcore so that it can be fabricated on a laser cutter and glued together in a matter of hours. The airframe is equipped with a front-mounted BLDC motor and propeller powered by a three cell LiPo battery and controlled by a 25 A ESC. It has ailerons, a rudder, and elevators for control surfaces. They are all controlled by small servomotors. It is currrently equipped with only a GPS sensor, commercial flight controller (with a built-in IMU), and an RC receiver.

In this configuration it is not possible to develop new flight control algorithms for two reasons. The first is that without access to the IMU data, attitude estimation—a mandatory component for the experimental flight controller—is impossible. The second is that it is not possible (at least easily!) to import novel flight control algorithms into the commercial flight controller. Indeed, this is the main purpose of the OSAVC. These areas will be addressed in the next phase of development where the vehicle will be equipped with the OSAVC, IMU, and serial radio. It optionally may include an airspeed sensor. Unlike the other vehicles under development, this vehicle does not currently plan to use the full distributed control architecture. The hardware block diagram of the UAV in the final state is in Fig. 8.6.

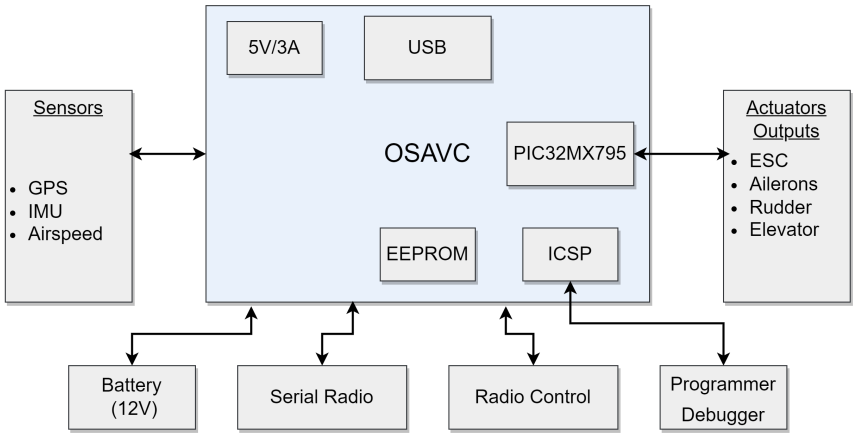

Figure 8.6: Fixed wing UAV hardware block diagram.

8.3.2 Status

The plane is fully designed and tested. In the academic 2022-2023 year a team of undergraduate researchers plan to implement the OSAVC architecture onto the plane for their senior engineering project. It will use an existing flight control model developed at the ASL, but the model gains will be developed in simulation by student researchers and flight tested on the aircraft. One stretch goal is to adapt the OSAVC firmware to an inexpensive commercial flight controller for this project as a way to reduce the cost of the aircraft. Fig. 8.7 shows the plane as well as a section of the model where the OSAVC is housed.

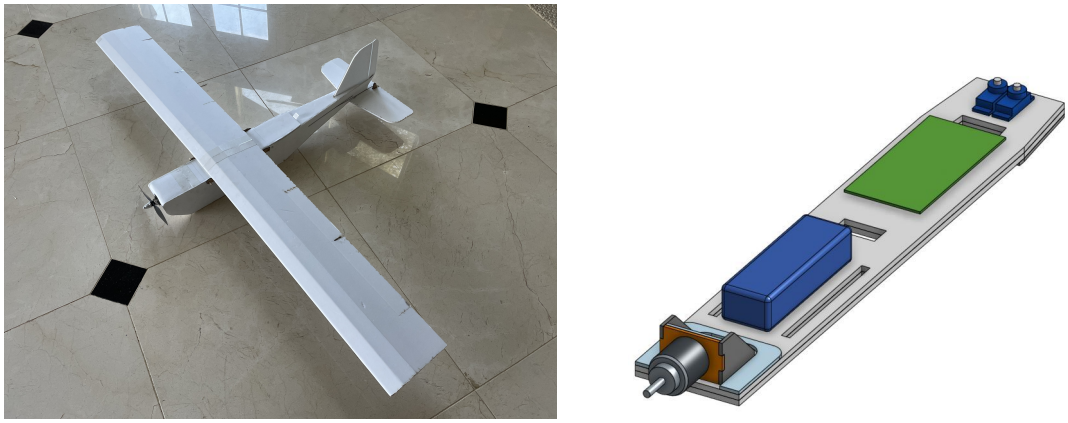

Figure 8.7: The left image shows the fixed-wing UAV. The right image is a model demonstrating where the OSAVC (green rectangle) mounts in the fuselage relative to the motor, battery (blue), and rudder and elevator servomotors.

As of this writing the team has already demonstrated the OSAVC in a hardware-in-the-loop configuration where a custom simulator is communicating to the OSAVC via MAVLink packets over serial, the autopilot control algorithm is running on the OSAVC and controlling the UAV actuators, and the OSAVC is sending actuator commands back to the simulator.

Chapter 9

Future Work

Perhaps the main value of the research done for this project is that it enables future research, because it is designed to be a starting point for a new autonomous vehicle. In this chapter we present several of the projects that are currently envisioned or already underway. Some of these have been presented in Chapter 8, others have been suggested along the way.

9.1 Quadcopter Flight Controller

One of the projects underway is the quadcopter flight controller detailed in Section 8.2. Rather than repeat that here, we suggest a development path for how to implement the controller and the distributed architecture discussed in Section 7.1. The status of the project at the time of writing is that the hardware is in place and includes: the quadcopter vehicle, the OSAVC, sensors, SBC, camera, and TPU. The manual control interface via the radio controller is complete including the mixing of the channels (quadcopters mix pitch, roll, and yaw controls to the motors depending on the orientation of the frame). The attitude estimation algorithm is in place. Attitude stabilization controller development is in progress.

The path towards vehicle autonomy is:

- Complete the stabilization algorithm (proportional inner loop control).

- Implement propotional motion control (outer loop control).
- Implement altitude control.
- Implement autonomous takeoff.
- Implement waypoint guidance as detailed in Section 7.7.

9.2 Fixed-wing Flight Controller

Another project currently underway is a flight controller for a fixed-wing plane, discussed in Section 8.3. This project is a test vehicle that engineering students will use to test controllers they have developed in a class on fixed-wing UAVs. The plane itself has been developed and includes a commercial flight controller. The steps to complete this project are:

- Acquire and implement any sensors that are needed (e.g., barometer for altitude measurement and optionally an airspeed sensor).
- Create the prototype control loop presented in Section 4.5.
- Implement the control algorithm used in the course with assistance of the course instructor.
- Optionally, recreate the firmware of the OSAVC to use an inexpensive, commercially available flight controller.

9.3 Hybrid Vision-LiDAR Mapping Sensor

Another project underway is to integrate the visual obstacle identification model and LiDAR data into a mapping sensor. Although we have developed the hardware and submodules for the sensor and integrated them onto the AGV, the final algorithm hasn't been developed and is presented here as future work. The specific implementation on

the AGV uses a pre-trained model on the Efficient-Det framework to identify three types of colored cones and provides the bounding box coordinates for each detection. These coordinates indicate a landmark within the field of view of the camera. The corresponding LiDAR data are used to confirm the existence of the obstacle and to provide a measurement of both the range and bearing of the landmark. These measurements can used to generate a map of the landmarks or put into a SLAM framework to localize the vehicle (e.g., in the case of a GPS-denied environment). This will appear in a future publication.

9.4 System Identification of a Ground Vehicle

Yet another project underway is to use GPS data to identify AGV parameters that inform its kinematic and dynamic model. The main goal of this project is to use the GPS data to refine and calibrate parameters of the vehicle that are difficult to measure directly, in particular, the effective tire radius, the relationship between the measured servo angle and the vehicle angular velocity, and the static coefficient of friction between the tires and a given road surface. The effective tire radius is used to determine the vehicle speed accurately in the odometry model. The GPS provides an independent estimate of the vehicle velocity. The ratio of GPS velocity and rear wheel angular velocity is the effective tire radius. By itself this parameter calibrates the odometry model for velocity. Once determined, it helps identify the transfer function between the steering servo angle and the vehicle angular velocity (as well as turning radius) using least-squares regression and GPS position data. We can use the calibrated odometry models to determine the lateral and longitudinal dynamics of the tires. Independent position and velocity measurements compared against the odometry estimates provide a convenient mechanism to determine wheel slip. Finally, slip detections keeps the AGV in the non-slip regime and is used to identify the road-tire interface parameters *in-situ*, that is, during mission operation. This work will be published the 2023 ION/PLANS conference proceedings.

9.5 Autonomous Race Car

A rich area of research using the AGV platform and combining Sections 9.3 and 9.4 is to develop an autonomous race car. In this project the mapping sensor is used to determine the inner and outer race track boundaries and the road-tire interface parameters determine the friction limits of the tires. The goal of the research would be to provide optimal guidance through a given course once its boundaries are autonomously mapped and the road-tire friction parameters determined.

9.6 Autonomous Vehicle Course

Yet another possible use of the OSAVC is as part of a high school or university course on autonomous vehicles. In this idea, a class is structured around developing an autonomous vehicle of some type using the OSAVC and distributed architecture presented earlier along with a suitable vehicle platform.

9.7 Final Thoughts

The ideas presented in this chapter are a small subset of possibilities enabled by this research. We hope that by detailing the ones in progress others can think of new areas to pursue. In fact, this project has already helped several student researchers through collaboration with CROSS and the Google Summer of Code. We believe it can continue to provide this opportunity going forward because of its open source bonafides and established relationships in the open source world.

Chapter 10

Conclusions

Unlike many academic research projects, this work has often been collaborative with students from ASL as well as interns from the Google Summer of Code program. It is our belief that individuals working together will always exceed the accomplishments of those same individuals working separately. This is one of the main reasons we chose to make this project open source–to promote collaborative development. It was exceedingly gratifying, therefore, to see so much interest in the project from all over the world as well as here at home. Perhaps the greatest potential contribution of this research is to provide a control platform to enable future autonomous vehicle research and a community to collaborate with.

Our hope is that by developing the OSAVC and integrating it into a distributed control framework, vehicle developers can take advantage of the power of embedded programming. We also hope that by following the example code they will be able modify it for their own purposes, tremendously shortening the learning curve.

Embedded programming in C can be a daunting prospect to the programmer unfamiliar with the process. Paradoxically, good embedded programming practices make understanding and troubleshooting real-time systems easier. This is because every aspect of the program is dictated by the programmer—there are no hidden mechanisms behind the scenes. Therefore, while hardware abstraction (RTOS or HAL) does allow for relatively easy coding of complex tasks, it hides important aspects of what is

happening at the hardware level. While this may not matter for many applications, for real-time control it is critical to understand the operations of the microcontroller and the hardware peripherals to ensure predictable latency and efficient code. Also, understanding the low-level hardware allows for easier troubleshooting of faulty code.

A summary of the contributions of this research are:

- An open source design of a real-time autopilot that is vehicle agnostic, that is, easily adaptable to many different types of craft. It fits within a modular system architecture of our design suitable for resource-constrained autonomous systems along with open source libraries for many common functions.

- An open source repository including hardware design files, microcontroller initialization code, sensor and actuator libraries for common sensors and outputs, attitude estimation and other navigation algorithms, and vehicle control algorithms.

- A benchmark algorithm to evaluate real-time hardware performance and an evaluation of the performance of four different processors (three microcontrollers and one SBC) using the benchmark.

- A ground vehicle platform suitable for demonstration of the controller capabilities presented in detail here. Three additional use cases using the real-time controller within the distributed architecture that are either completed or under development.

- A custom object detection algorithm trained to identify landmarks in the environment and deployed to a TPU.

- A sensor module consisting of a camera, a lightweight LiDAR, and a panning servo designed to be used in conjunction with the object detection algorithm for mapping landmarks.

Appendix A

Complementary Filter

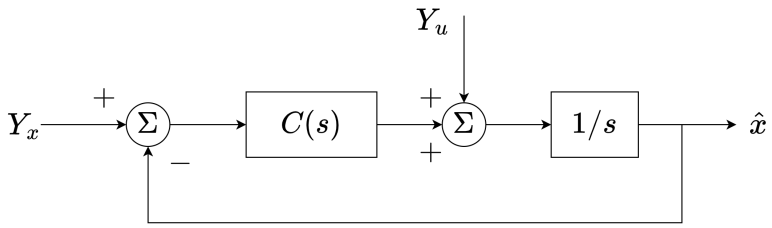

Figure A.1: Block diagram for the complementary filter.

The block diagram for a complementary filter is shown in Fig. A.1. In the figure Y_x represents one measurement of x, Y_u represents an independent measurement of $\dot{x}$ (the time derivative of x) and the filtered estimate is $\hat{x}$. We consider Y_u to be accurate in the high frequency domain, and Y_x to be accurate in the low frequency domain. $C(s)$ is the complementary filter itself. Using block diagram algebra, we see that:

$$\{(Y_x - \hat{x})C(s) + Y_u\}1/s = \hat{x} \tag{A.1}$$

solving for $\hat{x}$ yields:

$$\hat{x} = \frac{Y_x(s)C(s) + Y_u(s)}{s + C(s)} \tag{A.2}$$

$$= Y_x(s)\frac{C(s)}{s + C(s)} + Y_u(s)\frac{1}{s + C(s)} \tag{A.3}$$

$$= \frac{C(s)}{s + C(s)}Y_x(s) + \frac{s}{s + C(s)}\frac{Y_u(s)}{s} \tag{A.4}$$

We define

$$T(s) = \frac{C(s)}{s + C(s)} \quad \text{and} \tag{A.5}$$

$$S(s) = \frac{s}{s + C(s)} \tag{A.6}$$

where $S(s)$ is called the sensitivity and $T(s)$ is called the complementary sensitivity because $T(s) + S(s) = 1$ at all frequencies, s. Thus the complementary filter provides a weighted estimate as a function of frequency of x given two different measurement inputs. The dynamics of the filter $C(s)$ determine the crossover frequency between the two measurements. Typically, Y_x is accurate at low frequencies but noisy, whereas Y_u is accurate at higher frequencies but has some measurable inaccuracy at low frequencies, usually in the form of a slowly varying bias near DC.

Typically $C(s) = k_p$, a proportional gain, or if there is a constant bias, then $C(s) = k_p + k_i/s$ for integral control action to remove the bias. Using the Final Value Theorem we can show that integral action will remove a constant bias from the filter estimate completely.

Appendix B

Auto Regression with External Inputs

Autoregression with external inputs or ARX is a system identification model for a time-invariant system. The example below demonstrates how to find the motor parameters from a sequence of data taken with random white noise input. A common transfer function model for a DC motor relating the input voltage u to the motor velocity ω takes the following form in the Laplace domain:

$$G_{\omega u}(s) = \frac{b}{s+a} \tag{B.1}$$

The transfer function that relates the input voltage to the motor angle y, therefore takes the form

$$G_{yu}(s) = \frac{b}{s(s+a)} \tag{B.2}$$

The discrete form is found using the zero-order hold approximation by solving the following:

$$G(z)_{yu} = \frac{z-1}{z} Z\left\{\frac{b}{s^2(s+a)}\right\} \tag{B.3}$$

to find the Z transform of the expression in braces first calculate partial fraction expansion of the terms, then take the Z transform of each terms. Partial fraction expansion yields:

$$b\frac{1}{s^2(s+a)} = b\left[\frac{c}{s} + \frac{d}{s^2} + \frac{e}{(s+a)}\right] \tag{B.4}$$

With following values for c, d, e:

$$c = -\frac{1}{a^2}, d = \frac{1}{a}, e = \frac{1}{a^2} \tag{B.5}$$

Determining the Z transform of each term results in the following expression (where T is the sample time):

$$G(z)_{yu} = \frac{z-1}{z}\left[\frac{-b}{a^2}\frac{z}{z-1} + \frac{b}{a}\frac{Tz}{(z-1)^2} + \frac{b}{a^2}\frac{z}{(z-e^{-aT}}\right] \tag{B.6}$$

This simplifies to the following:

$$G(z)_{yu} = \frac{-b}{a^2} + \frac{b}{a}\frac{T}{(z-1)} + \frac{b}{a^2}\frac{z-1}{(z-e^{-aT})} \tag{B.7}$$

$$= \frac{b}{a^2}\frac{(aT + e^{-aT} - 1)z - aTe^{-aT} - e^{-aT} + 1}{(z-1)(z-e^{-aT})} \tag{B.8}$$

To determine the motor parameters we need to develop the difference equation of an autoregressive model with external inputs (ARX). The general form is:

$$Y(z) = \frac{B(z)}{A(z)}u(z) + \frac{1}{A(z)}e(z) \tag{B.9}$$

The first step is to determine the difference equation of the discrete transfer function developed in B.8.

$$\frac{Y(z)}{U(z)} = G(z)_{yu} = \frac{b}{a^2}\frac{(aT + e^{-aT} - 1)z - aTe^{-aT} - e^{-aT} + 1}{(z-1)(z-e^{-aT})} \tag{B.10}$$

$$\implies Y(z)(z^2 - (1 + e^{-aT})z + e^{-aT}) \tag{B.11}$$

$$= U(z)\frac{b}{a^2}\left[(aT + e^{-aT} - 1)z - aTe^{-aT} - e^{-aT} + 1\right] \tag{B.12}$$

$$\implies y(k) = (1 + e^{-aT})y(k-1) - e^{-aT}y(k-2) + \tag{B.13}$$

$$\frac{b}{a^2}\left[(aT + e^{-aT} - 1)u(k-1) + (1 - e^{-aT} - aTe^{-aT})u(k-2)\right] \tag{B.14}$$

This has the general form for a second order system of:

$$y(k) = -a_1y(k-1) - a_2y(k-2) + b_1u(k-1) + b_2u(k-2) \tag{B.15}$$

Now the data is placed into matrix form where each row represents the difference equation for a given time k.

$$Y = S\bar{\Theta} \text{ where:} \tag{B.16}$$

$$\bar{\Theta} = \begin{bmatrix} a_1 \\ a_2 \\ b_1 \\ b_2 \end{bmatrix} \tag{B.17}$$

where one row of the matrix equation takes the form:

$$y(k) = \begin{bmatrix} -y(k-1) & -y(k-2) & u(k-1) & u(k-2) \end{bmatrix} \bar{\Theta} \tag{B.18}$$

Equating the parameters from the system transfer function we find:

$$a_1 = (1 + e^{-aT}) \tag{B.19}$$

$$a_2 = e^{-aT} \tag{B.20}$$

$$b_1 = \frac{b}{a^2}(aT + e^{-aT} - 1) \tag{B.21}$$

$$b_2 = \frac{b}{a^2}(1 - e^{-aT} - aTe^{-aT}) \tag{B.22}$$

To find the best estimate of the polynomial coefficients a least squares approximation to the data is found:

$$\hat{\Theta} = (S^T S)^{-1} S^T Y \tag{B.23}$$

The last step is to extract the a and b parameters from the polynomial coefficients. The parameter a can be calculated from a_2 and b can be determined from b_1

$$a = -\log(a_2)/T \tag{B.24}$$

$$b = b_1 a^2/(aT + e^{-aT} - 1) \tag{B.25}$$

These values of a and b are the best estimate of the continuous parameters of the motor given the model in B.1.

Bibliography

[1] Space technology comes down to Earth in new agricultural device. https://www.sciencedaily.com/releases/2016/05/160510085141.htm. Accessed 2023-02-19.

[2] Egbert Bakker, Lars Nyborg, and Hans B. Pacejka. Tyre Modelling for Use in Vehicle Dynamics Studies. *SAE Transactions*, 96:190–204, 1987. Publisher: SAE International.

[3] Christian Berger and Michael Dukaczewski. Comparison of architectural design decisions for resource-constrained self-driving cars - a multiple case-study. In E. Plödereder, L. Grunske, E. Schneider, and D. Ull, editors, *Informatik 2014*, pages 2157–2168, Bonn, 2014. Gesellschaft für Informatik e.V.

[4] Andrea Bonarini, Matteo Matteucci, Martino Migliavacca, and Davide Rizzi. R2P: An open source hardware and software modular approach to robot prototyping. *Robotics and Autonomous Systems*, 62(7):1073–1084, July 2014.

[5] Bosch's Giant Robot Can Punch Weeds to Death - IEEE Spectrum. https://spectrum.ieee.org/bosch-deepfield-robotics-weed-control. Accessed 2023-02-19.

[6] Marco Caruso, Angelo Maria Sabatini, Daniel Laidig, Thomas Seel, Marco Knaflitz, Ugo Della Croce, and Andrea Cereatti. Analysis of the Accuracy of Ten Algorithms for Orientation Estimation Using Inertial and Magnetic Sensing

under Optimal Conditions: One Size Does Not Fit All. *Sensors*, 21(7):2543, January 2021. Number: 7 Publisher: Multidisciplinary Digital Publishing Institute.

[7] ChibiOS free embedded RTOS - ChibiOS/RT.

[8] USB Accelerator datasheet.

[9] César Debeunne and Damien Vivet. A Review of Visual-LiDAR Fusion based Simultaneous Localization and Mapping. *Sensors*, 20(7):2068, January 2020. Number: 7 Publisher: Multidisciplinary Digital Publishing Institute.

[10] E. Dorveaux, D. Vissière, A. Martin, and N. Petit. Iterative calibration method for inertial and magnetic sensors. In *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pages 8296–8303, December 2009. ISSN: 0191-2216.

[11] Emad Ebeid, Martin Skriver, Kristian Husum Terkildsen, Kjeld Jensen, and Ulrik Pagh Schultz. A survey of Open-Source UAV flight controllers and flight simulators. *Microprocessors and Microsystems*, 61:11–20, September 2018.

[12] Gene F Franklin, J David Powell, Michael L Workman, et al. *Digital control of dynamic systems*, volume 3. Addison-wesley Reading, MA, 1998.

[13] Emil Fresk and George Nikolakopoulos. Full quaternion based attitude control for a quadrotor. In *2013 European Control Conference (ECC)*, pages 3864–3869, 2013.

[14] Arthur Gelb et al. *Applied optimal estimation*. MIT press, 1974.

[15] Aaron Hunter, Pavlo Vlastos, Carlos Isaac Espinosa-Ramirez, Rishikesh Vanarse, Rupal Sharma, and Gabriel Elkaim. A Distributed Control Architecture for Resource-constrained Autonomous Systems. In *2022 IEEE International Systems Conference (SysCon)*, pages 1–6, April 2022. ISSN: 2472-9647.

[16] Colleen Josephson. *Exploring Low-Power Ubiquitous Sensing Using Rf Backscatter - ProQuest*. PhD thesis, Stanford University, 2020.

[17] Juraj Kabzan, Miguel I Valls, Victor JF Reijgwart, Hubertus FC Hendrikx, Claas Ehmke, Manish Prajapat, Andreas Bühler, Nikhil Gosala, Mehak Gupta, Ramya Sivanesan, et al. Amz driverless: The full autonomous racing system. *Journal of Field Robotics*, 37(7):1267–1294, 2020.

[18] M. Kesäniemi and K. Virtanen. Direct Least Square Fitting of Hyperellipsoids. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(1):63–76, January 2018. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

[19] Ching-Fang Lin. *MODERN NAVIGATION, GUIDANCE, AND CONTROL PROCESSING*. 1991.

[20] Alexander Liniger, Alexander Domahidi, and Manfred Morari. Optimization-based autonomous racing of 1:43 scale RC cars. *Optimal Control Applications and Methods*, 36(5):628–647, 2015. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/oca.2123.

[21] Mariano Lizarraga, Gabriel Hugh Elkaim, and Renwick Curry. Slugs uav: A flexible and versatile hardware/software platform for guidance navigation and control research. In *2013 American Control Conference*, pages 674–679. IEEE, 2013.

[22] Robert Mahony, Tarek Hamel, and Jean-Michel Pflimlin. Nonlinear Complementary Filters on the Special Orthogonal Group. *IEEE Transactions on Automatic Control*, 53(5):1203–1218, June 2008. Conference Name: IEEE Transactions on Automatic Control.

[23] Curtis Manore, Pratheek Manjunath, and Dominic Larkin. Performance of Single Board Computers for Vision Processing. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0883–0889, January 2021.

[24] Mars helicopter tech demo. https://mars.nasa.gov/technology/helicopter/. Accessed: 2021-10-5.

[25] Intel® Neural Compute Stick 2.

[26] Unmanned systems (UxS): Revolutionizing NOAA missions. https://sab.noaa.gov/wp-content/uploads/2021/08/. Accessed: 2021-10-5.

[27] Matthew O'Kelly, Varundev Sukhil, Houssam Abbas, Jack Harkins, Chris Kao, Yash Vardhan Pant, Rahul Mangharam, Dipshil Agarwal, Madhur Behl, Paolo Burgio, and Marko Bertogna. F1/10: An Open-Source Autonomous Cyber-Physical Platform. *arXiv:1901.08567 [cs]*, January 2019. arXiv: 1901.08567.

[28] Open Source Autonomous Vehicle Controller. https://github.com/uccross/open-source-autonomous-vehicle-controller/, August 2022. original-date: 2021-06-07T20:50:17Z.

[29] Hans B. Pacejka and Egbert Bakker. THE MAGIC FORMULA TYRE MODEL. *Vehicle System Dynamics*, 21(sup001):1–18, January 1992.

[30] Parallel flight website. https://www.parallelflight.com/. Accessed: 2021-10-5.

[31] Jonghoon Park and Wan-Kyun Chung. Geometric integration on Euclidean group with application to articulated multibody systems. *IEEE Transactions on Robotics*, 21(5):850–863, October 2005. Conference Name: IEEE Transactions on Robotics.

[32] Pic32mx5xx6xx7xx Family Datasheet. https://ww1.microchip.com/downloads/, November 2022. Accessed: 2022-11-19.

[33] Sharon Rabinovich, Renwick E Curry, and Gabriel H Elkaim. Toward dynamic monitoring and suppressing uncertainty in wildfire by multiple unmanned air vehicle system. *Journal of Robotics*, 2018, 2018.

[34] Rajesh Rajamani. Lateral and Longitudinal Tire Forces. In Rajesh Rajamani, editor, *Vehicle Dynamics and Control*, Mechanical Engineering Series, pages 355–396. Springer US, Boston, MA, 2012.

[35] Rajesh Rajamani. Longitudinal Vehicle Dynamics. In Rajesh Rajamani, editor, *Vehicle Dynamics and Control*, Mechanical Engineering Series, pages 87–111. Springer US, Boston, MA, 2012.

[36] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement, April 2018. arXiv:1804.02767 [cs].

[37] ROB0170 NXP Cup Race Car Chassis - DFRobot | Mouser.

[38] 'Saildrone' footage offers rare peek inside a category 4 hurricane. https://www.nytimes.com/2021/10/03/us/saildrone-boat-hurricane-video.html. Accessed: 2021-10-5.

[39] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[40] Mingxing Tan, Ruoming Pang, and Quoc V. Le. EfficientDet: Scalable and Efficient Object Detection. *arXiv:1911.09070 [cs, eess]*, July 2020. arXiv: 1911.09070.

[41] P. Vlastos, A. Hunter, and R. Curry. Partitioned gaussian process regression for online trajectory planning for autonomous vehicles. In *The 21st International Conference on Control, Automation, and Systems (ICCAS2021)*, October 2021.

[42] Pavlo Vlastos. *Guidance, Navigation, and Control of Autonomous Surface Vehicles for Optimal Exploration and Low-cost Oceanography*. PhD thesis, University of California, Santa Cruz, 2022.

[43] Pavlo Vlastos, Aaron Hunter, Renwick Curry, Carlos Isaac Espinosa Ramirez, and Gabriel Elkaim. Applied Partitioned Ordinary Kriging for Online Updates for Autonomous Vehicles. In *2022 IEEE International Systems Conference (SysCon)*, pages 1–5, April 2022. ISSN: 2472-9647.

[44] Hans Wackernagel. Ordinary Kriging. In Hans Wackernagel, editor, *Multivariate Geostatistics: An Introduction with Applications*, pages 79–88. Springer, Berlin, Heidelberg, 2003.

[45] A robot to help improve agriculture and wine production. https://www.sciencedaily.com/releases/2015/01/150128113713.htm. Accessed 2023-02-19.