REAL-TIME CONTROL OF BALANCING ROBOT USING ROS

by

Mike Moore

THESIS

Presented to the Faculty of

The University of Houston-Clear Lake

In Partial Fulfillment

Of the Requirements

For the Degree

MASTER OF SCIENCE

in Computer Engineering

THE UNIVERSITY OF HOUSTON-CLEAR LAKE

DECEMBER, 2018

REAL-TIME CONTROL OF BALANCING ROBOT USING ROS

by

Mike Moore

APPROVED BY

Thomas Harman, PhD, Committee Chair

James Dabney, PhD, Committee Member

Luong Nguyen, PhD, Committee Member

Carol Fairchild, M.S., Committee Member

APPROVED/RECEIVED BY THE COLLEGE OF SCIENCE AND ENGINEERING:

Said Bettayeb, PhD, Associate Dean

Ju H. Kim, PhD, Dean

Dedication

This work is dedicated to three of my most important supporters and unconditional advocates. Dad, thank you for teaching me the importance and value of education. Though you didn't know it at the time, you were the first engineering teacher I ever had. I may have been a reluctant pupil in those days, but it is wonderful to reflect on how far I have come. None of this would have happened without you. Mom, you spent many patient hours with an impatient kid. Thank you for teaching me to write. It has given me an outlet for my creativity while encouraging clarity of mind. I will carry those lessons with me for the rest of my life. Kristen, my future wife, it has been an incredible journey so far, and it is only the beginning. Thank you for the love and encouragement throughout all the recent late nights and grumpy next mornings. As you do with your own students, you bring to light the best in me. I cannot wait to begin the next phase of our lives together. We still have so much to offer to the ones we love.

Acknowledgments

We don't accomplish anything in this world alone... and whatever happens is the result of the whole tapestry of one's life and all the weavings of individual threads from one to another that creates something.

– Sandra Day O'Connor

The "tapestry" that is Bobble-Bot is more of the two-wheeled, mini-robotic, Frankenstein variety. He is held together by 3D printer filament, caffeine induced control software, and of course, a little bit of duct tape. Nevertheless, the quote above applies all the same. I must take a moment to recognize and give thanks to those around me. They too, for better or worse, share in his creation.

First, thank you to my thesis committee for help in promoting this work, reviewing it, and inspiring me to see it through to the end. I am grateful for the faculty and staff at UHCL, and especially grateful to my advisors Dr. Koc and Dr. Harman.

Thank you to all my friends and colleagues at NASA Johnson Space Center and CACI. Bob Zehenter, Dan Erdberg, Louis Nguyen, Rachel Borland, Jason Harvey, and Robert Mcphail have been especially great supporters and mentors to me over the years. I am thankful to all of my friends, but most especially James Holley, Josh Sooknanan, Daniel Ponce, and Andy Welton. Bobble-Bot would not exist without the electrical and mechanical designs contributed by James and Josh. Daniel and Andy, thanks for being supportive of all of our crazed efforts over the years. Thanks to you guys, this all started in a garage off Ramada Drive over five years ago.

Lastly, I want to thank my brother Dennis, my sister Leah, and my friends Ryan Brewer and Zach Barrera. You all have been with me through the good times and the bad. Your friendship, love, support, and inspiration have made this work possible.

ABSTRACT

REAL-TIME CONTROL OF BALANCING ROBOT USING ROS

Mike Moore

University of Houston-Clear Lake, 2018

Thesis Chair: Thomas Harman, PhD

Real-time computing is an important feature in many robotic systems, particularly safety and mission-critical applications such as autonomous vehicles, spacecraft, and industrial manufacturing. To help meet the growing needs of the robotics community, the Robot Operating System (ROS) is currently undergoing a major redesign in which one of its primary design goals is to prioritize support for real-time computing (ROS2). This paper documents the design and development of a custom built, two wheeled, self-balancing robot that successfully demonstrates the use of ROS for high bandwidth, real-time control of an unstable system. An overview of the hardware and software design is provided before detailing the approach taken to implement and test the real-time system. A high fidelity simulator is developed to test the controller initially in software. An analysis is then carried out in order to predict desirable control gains and their expected performance. The work concludes with results from hardware system tests that show good comparison with the predictions made by the simulator.

TABLE OF CONTENTS

Chapter Page

1. Introduction 1
1.1 Inverted Pendulum Systems 1
1.2 Real-Time Systems 8
1.3 ROS 11

2. Theory 15
2.1 Mathematical Formulation 15
2.1.1 Pendulum Dynamics 17
2.1.2 Wheel Dynamics 18
2.2 Linearization 19
2.3 State Space Form 20

3. Design 23
3.1 Hardware Components 24
3.2 Assembly 28
3.3 User Interface 31
3.4 Software Components 33
3.5 Controller 39
3.5.1 Control Algorithm 41
3.6 Simulation 45
3.6.1 Python 46
3.6.2 Matlab Simulink 46
3.6.3 Unreal Engine 4 47
3.6.4 Gazebo 48
3.7 Summary 52

4. Implementation 55
4.1 Achieving Real-Time 56
4.1.1 Building the Real-Time Kernel 57
4.1.2 Benchmarking the System 59
4.1.3 Best Practices for Real-Time Programming 61
4.1.4 A Real-Time ROS Node 65
4.2 Device Drivers 68
4.2.1 CAN Communications 69
4.2.2 Motor Driver 72
4.2.3 IMU and State Estimation 75
4.3 Control Algorithm 76
4.4 Control Tuning 83
4.4.1 Simulated Tilt Control 84

4.4.2 Simulated Velocity Control 86
4.4.3 Simulated Turning Control 89
4.4.4 Validation with Hardware Testing 91

5. Conclusion 98
5.1 Hardware vs Simulation 99
5.2 Future Work 101

Glossary 110

6. Appendix 113
6.1 Bobble-Bot Chassis Mass Properties 113
6.2 Bobble-Bot Wheels Mass Properties 114
6.3 Python Double Pendulum 115
6.4 RasPi CAN Driver Configuration Files 118

LIST OF TABLES

Table Page

3.1 Hardware Modules with Data Sheet Ref. 26

3.2 Bobble-Bot User Specific Features 31

3.3 List of Software Modules 39

4.1 Bobble-Bot CAN Devices 69

4.2 BLDC Motor Configuration 73

4.3 IMU Sensor Summary 75

4.4 Controller States 81

4.5 Select controller configuration parameters 83

4.6 Control Gains for HW Test 92

LIST OF FIGURES

Figure		Page
1.1	A Modern Inverted Pendulum System	2
1.2	PENS-Wheel Inverted Pendulum Model Riattama et al. (2016)	4
1.3	Inverted Pendulum on a Mobile Base	5
2.1	Pendulum Free Body Diagram	17
2.2	Pendulum Free Body Diagram	18
2.3	State Space Fricke (2012)	20
3.1	Bobble-Bot Physical Design	23
3.2	Bobble-Bot System Diagram	25
3.3	Primary Hardware Components (1/2)	27
3.4	Primary Hardware Components (2/2)	28
3.5	Bobble-Bot Assembly	29
3.6	Actuator Assembly CAD	30
3.7	Left Actuator Parts	30
3.8	User vs Developer Controls	32
3.9	Real-Time System Monitoring	34
3.10	Bobble-Bot Software Architecture	36
3.11	Controller Block Diagram	42
3.12	Same Controller in HW and Sim	44
3.13	Bobble-Bot Simulator	45
3.14	Simulink Based TWIP Simulation Yamamoto (2018)	47
3.15	Bobble-Bot in Simulated Garage Environment	48
3.16	Bobble-Bot Gazebo Simulation	49
3.17	Controller Architecture in Simulation Environment Chitta et al. (2017)	50
3.18	Simulated Impulse Force Testing	52

3.19 Tilt Control Stability With Varying CG 53
3.20 Velocity Control Stability With Varying CG 53
4.1 Bobble-Bot on Test Stand 55
4.2 Comparison of Cyclic Test Benchmark 61
4.3 Bobble-Bot Real-Time Verification 68
4.4 Bobble-Bot CAN Communications Architecture 70
4.5 SocketCAN Architecture Linux-Foundation (2012) 71
4.6 Bobble-Bot CAN Messages 71
4.7 Chupacabra BLDC Motor Controller 72
4.8 Motor Test Stand 74
4.9 Motor Driver Verification 74
4.10 General 1D Digital Filter 79
4.11 Controller States 80
4.12 Tilt Control Tuning 84
4.13 Tilt Control Without Velocity Control 86
4.14 Velocity Control Tuning 87
4.15 Velocity Control 88
4.16 Turning Controller Performance 89
4.17 Tilt/Velocity While Turning 90
4.18 Tilt/Velocity During HW Impulse Test 93
4.19 Turn Rate/Position During HW Impulse Test 95
4.20 Turn Rate/Heading Control During HW Turning Test 96
4.21 Tilt and Position During HW Turning Test 97
5.1 Simulation vs Hardware Impulse Response 99
5.2 Simulation vs Hardware Square Drive Comparison 100

1

INTRODUCTION

Bobble-Bot is a modern take on a classical problem in control theory. The robot represents a unique solution to the well understood problem of control of a two wheeled inverted pendulum (TWIP). This subject has a rich history in both academia and industry. Inverted pendulum control systems see application within robots for domestic and industrial use, control of rockets and missiles (see figure 1.1), and object transport utilizing drones Lundberg and Barton (2010). Boubaker (2017) describes how inverted pendulum systems have been well documented and understood within the research community for over one hundred years. Though the theory is well understood, a real implementation of such a robot remains anything but a trivial endeavor. The Bobble-Bot control system requires coordination between several important systems – power, sensors, control, communications, and a variety of cooperating software algorithms all have to run in harmony in order for the robot to perform its qualitatively simple, but technically complex task of balancing itself. This report will focus on a modern design including the hardware component selection, software implementation, and validation of the robotic system. The design of a real-time controller optimized for low latency will be the focus of this work. Simulation will be used to determine control gains for the system before testing and demonstrating the final working implementation. This first chapter will provide some background on inverted pendulum systems, real-time control, and ROS.

1.1 Inverted Pendulum Systems

The inverted pendulum and TWIP systems are a classical problem in dynamics and control theory. They are often used as a demonstration system for testing different

Figure 1.1: *A Modern Inverted Pendulum System*

control design and implementation approaches Boubaker (2017). In this respect, Bobble-Bot is no different. Historically, studies of inverted pendulum systems have contributed to advancements in control of under-actuated robotic systems, design of mobile inverted pendulums (the Segway being the most recognizable), and even gait pattern generation for humanoid robots Boubaker (2012). The list of contributions made by researchers in this area is extensive. This section will cover a few selected works that were useful to the understanding of the theory or influenced the design and development of Bobble-Bot in some way.

A particularly interesting and inspiring application of inverted pendulum control is the attitude control of rockets during ascent and landing. One of the most striking and modern applications of the theory can be witnessed when watching SpaceX's recent landing of the Falcon Heavy boosters (see SpaceX 2018 video). This company is currently developing re-usable rocket technology whose design leverages the theory of inverted pendulum control to accomplish the feat depicted in figure 1.1. Each

Falcon Heavy booster accomplishes its balancing technique in part by using its nine engines, each of which can be throttled and gimbaled. The engines are commanded by a feedback controller that is continually monitoring the rocket's dynamical state as it comes in for its landing. Just like Bobble-Bot's controller, the mathematics of inverted pendulum control theory can be applied to help design the Falcon Heavy's booster landing controller. Of course, this is a simplification of the real detail involved in the full entry, descent, and landing software for the boosters. For the interested reader, Tan and Wheeler (2014) and Carson et al. (2011) provide a good starting point towards understanding some of the finer details. Landing rocket boosters is a critical first step in a re-usable rocket design that has the potential to save U.S. tax payers millions of dollars Mosher (2017). These recent SpaceX Falcon booster landing demonstrations show that the study of inverted pendulum systems continues to produce important technological advancements.

Sometimes inverted pendulum systems are implemented for less serious reasons than the previous example. Many have heard of and used the Segway. This is one of the most recognizable inverted pendulum systems today. Riattama et al. (2016) provides a lesser known implementation of an inverted pendulum system designed to transport a person. As seen in figure 1.2, the PENS-Wheel also uses an inverted pendulum model for its balance controller. The motivation for building the PENS-Wheel appears to be similar in spirit to the motivations for building Bobble-Bot. The main conceptual difference between the two comes in the number of actuators available to the controller. Bobble-Bot is a TWIP, whereas the PENS-Wheel is more closely related to the cart-pendulum system.

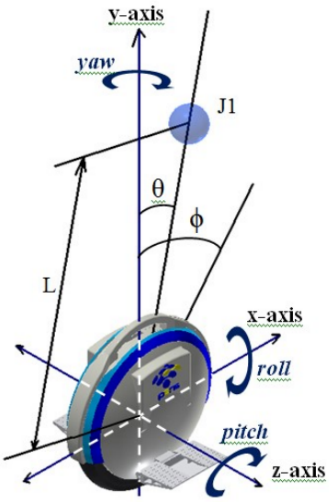


Figure 1.2: *PENS-Wheel Inverted Pendulum Model Riattama et al. (2016)*

The cart-pendulum system is a form of an inverted pendulum that consists of a mass at the end of a massless rod that is attached by a rotational pin joint to a movable cart. This system is shown in figure 1.3. The inverted pendulum on top of the cart is in an unstable equilibrium when it is standing upright. Theoretically, this equilibrium can be maintained without active control as long as there are no disturbance forces acting on the system. Clearly such conditions do not exist in reality. The real system can be kept in balance by using a force based feedback controller. This controller applies a force to move the pendulum's center of gravity (CG) in such a way that it dampens the resulting motion. The force is actively controlled based on sensor feedback in order to bring the system back to its natural unstable equilibrium. The system is analogous to a person balancing a broom in their hand.

The cart-pendulum system shown in figure 1.3 is a simplified version of the TWIP as it has only two degrees of freedom as opposed to the TWIP's three degrees of freedom. In 1.3, we see that the pivot point of the pendulum constrains its motion to a single rotation axis coming out of the page. Furthermore, the force, F, is controlled

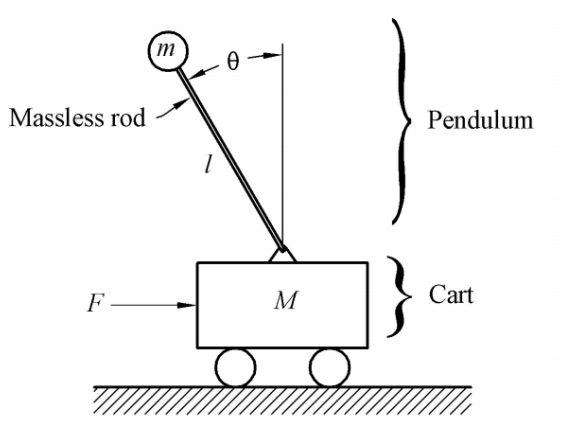


Figure 1.3: *Inverted Pendulum on a Mobile Base*

to move the cart mass in a single horizontal direction. Taken together, these motions describe the cart-pendulum system's two degrees of freedom. A simple math model for the system is provided using the variables depicted in 1.3. The following equations of motion were verified using the Newtonian and Lagrangian approach and come from Castro (2012).

$$(M+m)\ddot{x} - ml\ddot{\theta}cos\theta + ml\dot{\theta}^2 sin\theta = F$$

$$l\ddot{\theta} - gsin\theta = \ddot{x}cos\theta$$

The robot presented in this paper is an inverted pendulum anchored to a base platform with a wheel on each side. Bobble-Bot's CG is about six inches above the center of its wheels. In this case, motors drive each wheel independently. The torque from the motors spins the wheels and moves the base of the robot in order to keep the tilt angle of the pendulum in balance. In this sense, the torque imparted to the wheels accomplishes the same thing as the force controlled in the cart-pendulum system example. Unlike the cart-pendulum system though, the TWIP can act like a differential drive robot. Commanding a torque differential between the two wheels allows the robot to turn. A TWIP with two independently driven motors, like Bobble-

Bot, can move along curved paths. The TWIP is a more maneuverable system than the cart-pendulum, but it is a more complicated dynamical system to model. Chapter two presents a simplified set of the Bobble-Bot equations of motion that were used early on to gain insight during the design of the balance controller. Katariya (2010) is a great starting point reference that provides a more rigorous derivation of the non-linear equations of motion of the TWIP system with independently driven wheels. Katariya also presents a controllability and observability analysis of the system using Matlab. This is one of many references available in academic literature that focuses on mathematical modeling, simulation, and controllability of TWIP systems. An and Li (2013), Oktay Erkol (2018), and Perez-Polo et al. (2014) provide even more background material on the relevant dynamics and control theory.

This work extends the theoretical approaches by providing a working design and implementation of a high performance balance controller running as a real-time process within an embedded Linux system. The final implementation produces a self-balancing robot that is capable of a great deal of maneuverability. Bobble-Bot can drive quickly, make sharp turns, and go up and down ramps. The focus of this work is kept on the design and implementation details that make this possible. The software design calls for the development of interfaces that enable real-time monitoring and logging of many of the critical system parameters as the robot is driving. This allows the robot to be used as a platform for testing the accuracy of the math models and control theory found in the literature. As an added benefit, many elements of the adopted software architecture for Bobble-Bot are flexible enough to be reused for the control of other robotic systems that may be defined by a different set of differential equations.

Linear control theory is one of the most commonly adopted approaches when designing controllers for arbitrary robotic systems. Azar et al. (2019) and Pratama et al. (2015) show that these kinds of classical controller designs can produce practical working solutions for TWIP balance control. For this reason, Bobble-Bot adopts a cascaded Proportional-Integral-Derivative (PID) controller design for its balance and drive controllers. This design adds yet another entry to the list of non-linear dynamical systems that can be effectively controlled using the techniques of linear control theory. As suggested by the numerous publications on the various modeling and simulation approaches to TWIP control design, the performance of a TWIP balance controller can be greatly enhanced by analyzing and tuning the closed loop system using techniques from the field of modeling and simulation. This work confirms the effectiveness of that approach in section 4.4.

The implementation of a real TWIP balance controller that is capable of robust differential drive control heavily depends on minimizing the latency seen by the balance controller during the execution of its feedback control loop. This is a topic that is often overlooked in the literature. Furthermore, providing a data logging and real-time monitoring capability is heavily dependent upon the selected software architecture. In Bobble-Bot's case, the latency problem is addressed by incorporating a real-time operating system into the software design. The logging and real-time monitoring capability is accomplished by leveraging the open-source tools and software packages found within the ROS community. The final two sections of this chapter will provide a bit of background and terminology relevant to these two topics.

1.2 Real-Time Systems

When reviewing the literature cited in the previous section we see that real-time control issues are not generally discussed in the presentation of TWIP control designs. Many of the implementations rely on Matlab's Real-Time Workshop. While this is a useful software tool for rapid prototyping, it has the disadvantage of hiding the real-time aspects from the control engineer. This creates a separation from the real world problems that often emerge during time-critical applications like inverted pendulum systems. It is useful to study best practices for implementing real-time controllers especially before attempting to build a system like Bobble-Bot. Gambier (2004) provides a nice introduction to the subject.

We begin with the definition of a real-time system from the IEEE Portable Operating System Interface Standard (POSIX) repeated in Natale (2014).

> *A real-time system is one in which the correctness of a result not only depends on the logical correctness of the calculation but also upon the time at which the result is made available.*

This definition makes it clear that real-time systems are inherently concerned with how their own internal timing synchronizes with the outside world. These types of systems are composed by various tasks that must control or react to events within the environment that are happening in "real time". Real-time computing is concerned with guaranteeing a result is available at a prescribed point of time within a defined time tolerance. This requirement is referred to as a deadline. Real-time systems are further classified into hard and soft real-time systems depending on the criticality of meeting these deadlines. Hard real-time systems are systems which will fail when timing deadlines are missed. These failures may cause serious damage or loss of life.

See Koopman (2014) and Faccini (2013) for two recent examples of critical failures in hard real-time systems which resulted in disaster and tragedy. It is clear that hard real-time systems must take a considerable amount of extra precaution in order to keep them as safe as possible. Fault tolerance is an important concern in these safety-critical systems because without it, a single component failure could lead to a missed deadline and a catastrophic system failure.

Bobble-Bot falls under the latter category of real-time systems known as soft real-time systems. In these systems, the meeting of the timing deadline is desirable but not mandatory. Delay and jitter within the balance control loop causes degraded performance and can result in the robot toppling over. For that reason, great care is taken to ensure that the control loop operates at as close to a fixed 250 Hz loop rate as possible. Banking systems, video game consoles, air conditioning, temperature control, and streaming audio and video are some other examples of soft real-time systems.

A real-time system can be implemented by carefully programming an embedded microcontroller like the ATmega328P found on the Arduino Uno. The problem with this approach is that it leaves the developer to manage the system's timing on their own. It is very easy to make mistakes and inadvertently adopt programming paradigms which violate best practices for real-time programming. Furthermore, many of the commonly used Arduino libraries are not written with real-time systems in mind. They often make use of hardware interrupts that make real-time determinism impossible. The serial communication libraries that are pervasive in the example sketches are a good example of this. Many self-balancing robots found online are able to achieve limited success with their balance controllers using an Arduino based system.

Likely the jitter within their control loop is hampering the system from performing as well as it could.

In practice, problems like this can be avoided by adopting a Real-time Operating System (RTOS) like VxWorks, QNX, eCos, and RTLinux. These operating systems are especially designed with real-time programming in mind. They provide software libraries and drivers that are carefully designed for real-time applications. Within an RTOS, real-time tasks are scheduled by a special scheduling algorithm. These scheduling algorithms enable the developer to have more control of the overall computer-system orchestration by the use of process priorities. An RTOS generally contains a real-time kernel and other higher-level services such as file management, protocol stacks, a Graphical User Interface (GUI), and other components. Most of the additional services revolve around real-time Input/Output I/O device drivers Micrium (2015).

To achieve its own soft real-time requirements, Bobble-Bot makes use of the Preemptive Real-Time patch set to the Linux kernel (PREEMPT_RT) to convert vanilla Linux into an RTOS. Section 4.1 covers the subject in much greater detail. The PREEMPT_RT patch was completed in 2009 by a small team of kernel developers. Lots of companies use PREEMPT_RT as their RTOS of choice in order to build industrial systems with somewhat relaxed timing requirements of around one millisecond precision. BMW is one such company that uses the PREEMPT_RT patch set to build real-time capable autonomous driving prototypes within their Car IT division. The development of these self-driving car prototypes and their reliance on a patched version of the Linux kernel caused quality concerns within the organization. Hence, BMW Car IT began making significant contributions towards making main-line Linux

real-time capable in 2014 Bulwahn (2018). At the time of this writing, the community is still waiting on real-time capability within main-line Linux.

This section serves as a mere introduction to an entire sub-discipline within controls engineering. Consult the provided references for a jumping off point. The final section of this introductory chapter will provide background on ROS and provide some justification in its selection as Bobble-Bot's core software infrastructure.

1.3 ROS

Most of the self-balancing TWIP robots that were researched for this work were either based around an Arduino or other microcontroller and SDK platforms that support Matlab's Real-Time Workshop. Hau-Shiue and Lum (2013) and Gu et al. (2013) are some examples of implemented TWIP robots using an Arduino or embedded Real-Time Workshop. Arduino based TWIP platforms come with their own challenges often related to tuning the controller for the system. This is a challenge on the Arduino because it is difficult to get insight into the controller's performance as it is running. The Matlab Real-Time Workshop approach relies heavily on Matlab Simulink for simulation to do the controller design and tuning. Once the desired performance is achieved in simulation, the controls developer auto-generates real-time C code that implements the logic captured in the Simulink diagram of the controller. This approach works well when the simulation adequately captures the real system dynamics. Matlab has a variety of tools to assist in this whole process which they refer to as Model Based Design (MBD). The trade-off is that these tools all come at a hefty price tag for anyone that is not a student. Furthermore, the controller's intrinsic reliance on auto-generated C code which is targeted at a particular embed-

ded platform restricts the generality of the solution approach. Future code re-use for similar systems becomes unlikely. It also becomes more difficult to change and adapt the controller to other platforms and systems later on.

Significantly less common in the available literature are self-balancing robots that run ROS. RoboSavvy (2017) and Radovnikovich (2017) are examples of simulated TWIP robots within the ROS environment. These works do not include published information on any real robot that implements the controller they have running in their simulators. One potential reason for this is the lack of direct support or readily available documentation on how to design and build ROS enabled real-time systems. It is possible to do, Bobble-Bot serves as proof, but it is not an easy endeavour. One goal for this work is to lay out an architecture for others to follow, but before that is done, it is helpful to provide some explanation of what ROS is and why one would want to build a robot that uses it.

The Robot Operating System (ROS) is an open-source collection of software frameworks useful for the development of robotics software. Despite what the name might imply, ROS is not actually an operating system. Instead, ROS provides a common messaging system, device drivers, controllers, hardware abstraction layers, simulation frameworks, and automation tools. While responsiveness and low latency are desirable qualities in many robotic systems, ROS is not an RTOS and it provides very limited support for real-time systems. However, ROS2 is being developed to address these limitations Kay (2015). When you also include hobbyist robots, most of the robots in the world run Linux. For this reason, ROS and ROS2 are built on top of Linux. A major component at the core of ROS is a distributed messaging system. This enables system designers to de-couple the functionality of their robots into sep-

arate nodes connected together by the messaging system. This has the advantage of increasing reliabilty as it helps to prevent single points of failure. Furthermore, modularity is generally a preferable approach when architecting complex systems. Encouraging robot software components to follow a modular design also enables code re-use. Many common robotic capabilities are available already as open-source ROS packages. This allows developers of ROS enabled robots to focus on the unique challenges of their own system rather than on reinventing the wheel. Perhaps the most attractive quality of ROS is that it is free and open-source.

Despite the real-time limitations, ROS can still be used in a system with real-time requirements. However, the software in such a system must be carefully designed to ensure real-time responsiveness. The timing critical components of the software stack must be isolated and contain high speed and direct access to sensor and effector hardware. Furthermore, the control loop must be capable of sampling all sensor data, potentially filtering it, computing state error, applying a control law, and writing out effector commands in a fixed amount of time. Even with all of that accomplished, real-time performance is still not possible without an operating system which is real-time capable. For Linux, this can be obtained for free by modifying the kernel of the operating system with the PREEMPT_RT kernel patch. Once this is achieved, the ROS package ros_control can be used alongside a preemptive strict priority based scheduler policy to implement the control law. This system will be real-time capable so long as the controller task itself does not overrun its allotted time. More detail on the approach is provided in section 4.1 of this work.

This chapter has provided background and explanatory information needed to understand the terminology and implementation approach taken with Bobble-Bot. In

summary, inverted pendulum systems are inherently unstable but can be controlled reliably by standard linear controllers operating within a real-time system. Implementing real-time systems comes with its own challenges and concerns that are often overlooked in the academic literature. These problems must be addressed when implementing real systems. This can be done by carefully reviewing and applying standard best practices in real-time programming when designing the software architecture for the real-time system. ROS is an attractive software achitecture for robots as it enables re-use of many pre-existing open-source robotics software packages that solve a wide variety of common problems in robotics for free. The downside with ROS is that it provides extremely limited support for architecting real-time systems. The rest of this work shows the design and implementation of a hardware and software architecture for the real-time control of an unstable system that attempts to bridge this gap.

2

THEORY

This chapter formulates the Bobble-Bot velocity and tilt control problem mathematically utilizing a state space representation by linearizing the equations of motion about the robot's balance point. This is done not only as an excercise in understanding the unstable and non-linear dynamics of a Bobble-Bot like system, but also to serve as a basis for evaluating prototype simulations in an effort to select viable control system gains using control theory. Once these gains have been selected, a higher fidelity simulation that takes into account the fully non-linear dynamics is used to test a prototype of the control algorithm introduced in chapter three. The following two references are a more complete derivation of the dynamics of mobile inverted pendulum systems: Katariya (2010) and Saam Ostovari (2013). Please consult those references for a more rigorous treatment of the theory.

2.1 Mathematical Formulation

In this section we derive equations of motion for a Bobble-Bot analogue. The derivation can be conveniently split into two parts by the introduction of internal forces. P_{x} and P_{y} capture the forces imparted on the pendulum body at the pendulum/wheel joint. As will be seen as the derivation progresses, these forces are solved for and used to combine the equations derived from the two separate pendulum and wheel free-body diagrams. We end up with equations of motion that are in terms of the torque applied by the motor, pendulum length, mass, inertia, and wheel diameter.

LIST OF SYMBOLS

$\hat{i}$	Unit vector x direction
$\hat{j}$	Unit vector y direction
ϕ	Wheel rotation angle
τ	Torque imparted by motor on wheel
θ	Pendulum tilt angle
F	Horizontal wheel force
g	Gravitational constant
I_r	Inertia of robot about axis of tilt
I_w	Inertia of wheel about axis of rotation
k_t	Motor torque constant
k_v	Motor back EMF constant
L	Distance from wheel to mass center
m_r	Mass of robot
m_w	Mass of wheel
N	Normal force acting on wheel
P_x	Wheel joint reaction force acting on body in x direction
P_y	Wheel joint reaction force acting on body in y direction
r	Position of mass center
r_w	Radius of wheel
x	Horizontal direction
y	Vertical direction

2.1.1 Pendulum Dynamics

Starting with a free body diagram of the pendulum, the position of the center of mass can be expressed as

$$r = x\hat{i} - Lsin(\theta)\hat{i} + Lcos(\theta)\hat{j}$$

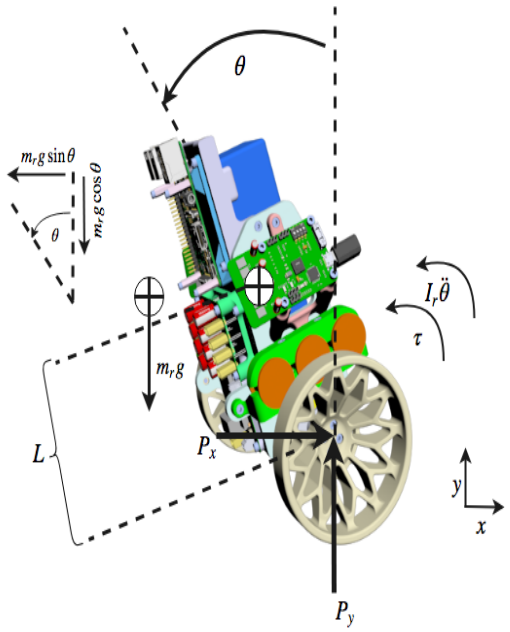


Figure 2.1: *Pendulum Free Body Diagram*

Differentiating the above twice gives the acceleration of the center of mass

$$\ddot{r} = (\ddot{x}cos\theta - \ddot{\theta}L)(cos\theta\hat{i} + sin\theta\hat{j}) - (\ddot{x}sin\theta + \dot{\theta}^2L)(cos\theta\hat{j} - sin\theta\hat{i})$$

Summing forces in the x and y direction and combining the expression results in the following second-order differential equation describing the translational motion of Bobble-Bot's mass center.

$$m_r(\ddot{x}cos\theta - \ddot{\theta}L) = -m_r gsin\theta + P_y sin\theta + P_x cos\theta$$

Likewise, we sum the torques about the center of mass in order to get another second order differential equation this time describing the rotational motion about

the Bobble-Bot mass center.

$$I_r\ddot{\theta} = -\tau + P_y L sin\theta + P_x L cos\theta$$

Combining the linear and rotational equations of motion we get the following for the pendulum dynamics

$$-(m_r L cos\theta)\ddot{x} + (I_r + m_r L^2)\ddot{\theta} = m_r g L sin\theta - \tau$$

2.1.2 Wheel Dynamics

Start with a free body diagram of one of the wheels.

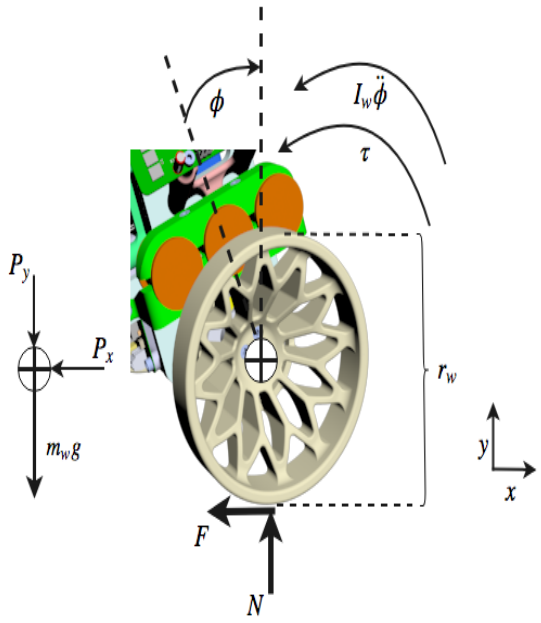


Figure 2.2: *Pendulum Free Body Diagram*

Summing the forces in the x-direction gives

$$m_w\ddot{x} = -P_x - F$$

Likewise, summing the torques about the wheel center gives

$$I_w\ddot{\phi} = \tau - Fr_w$$

Combining the above two equations and solving for P_{x} we get

$$P_x = -m_w\ddot{x} + \frac{I_w\ddot{\phi} - \tau}{r_w}$$

The force in the x direction, F, is known from the previous derivation of the pendulum dynamics. Substituting that force in and eliminating P_{x} by combining the linear and rotational wheel dynamics we get the following

$$m_r(\ddot{x} - \ddot{\theta}Lcos\theta + \dot{\theta}^2Lsin\theta) = -m_w\ddot{x} + \frac{I_w\ddot{\phi} - \tau}{r_w}$$

Rearranging the above gives the following second order non-linear differential equation describing the wheel dynamics

$$I_w\ddot{\phi} - (m_r r_w + m_w r_w)\ddot{x} + (m_r r_w Lcos\theta)\ddot{\theta} = m_r r_w\dot{\theta}^2 Lsin\theta + \tau$$

2.2 Linearization

In order to apply the theory of linear systems and controls, we would like to linearize the equations derived in the previous section. This allows us to develop a useful linear model which approximates the real Bobble-Bot system dynamics in the vicinity of its balance point. The following assumptions and simplifications allow us to linearize the equations of motion from the previous section.

Small angle approximation.

$$\theta^2 \approx 0, sin\theta \approx \theta, cos\theta \approx 1$$

No slip

$$\ddot{x} = -r_w\ddot{\phi}$$

Ideal motor

$$\tau = \frac{2k_tV}{r_m} - \frac{2k_tk_v\dot{\phi}}{r_m}$$

2.3 State Space Form

Representing our system as a set of differential equations is a bit cumbersome, and it is not immediately useful for most simulation tools. The state space representation of a system replaces an nth order differential equation with a single first order matrix differential equation. The state space representation of a system is depicted below.

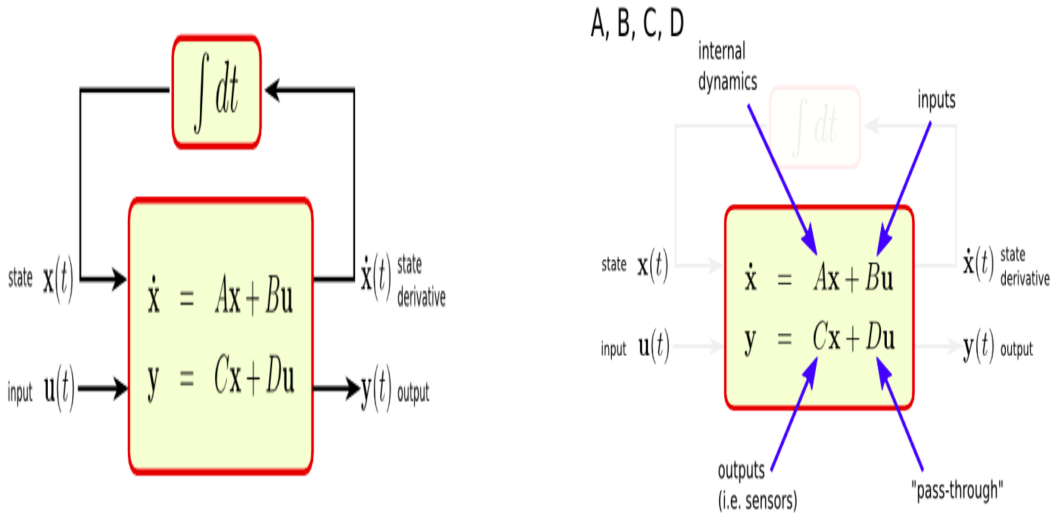


Figure 2.3: *State Space Fricke (2012)*

The first equation for $\dot{x}$ is called the state equation. The second equation for y is called the output equation. For an nth order system with r inputs and m outputs, the size of each of the matrices is as follows:

- x is $nx1$ (n rows by 1 column). This is the state vector, a function of time.
- A is nxn and is called the state matrix. It is constant.
- B is nxr and is called the input matrix. It is constant
- u is $rx1$ and is called the control input. It is a function of time.
- C is mxn and is called the output matrix. It is constant
- D is mxr and is called the feedthrough matrix. It is constant.

- y is $mx1$ and is called the ouput. It is a function of time.

To use this form for Bobble-Bot, we first define the following system constants

$$\alpha = r_m(m_r L^2 + I_r)$$

$$\beta = r_m(I_r + m_r r_w^2 + m_w r_w^2)$$

$$\gamma = m_r L r_m$$

Next, we apply the linearizing assumptions of section 4.3 to the equations of motion derived in sections 4.1 and 4.2. This results in a system of second order linear differential equations that can be encoded in state space form as follows:

$$\begin{bmatrix} \dot{\theta} \\ \ddot{\theta} \\ \dot{\phi} \\ \ddot{\phi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{\beta\gamma g - r_w^2\gamma^2}{\beta\alpha - \gamma^2 r_w^2} & 0 & 0 & \frac{2k_t k_v(\gamma r_w + \beta)}{\beta\alpha - \gamma^2 r_w^2} \\ 0 & 0 & 0 & 1 \\ \frac{\gamma r_w \alpha - \gamma^2 r_w g}{\beta\alpha - \gamma^2 r_w^2} & 0 & 0 & \frac{2k_t k_v(-\gamma r_w - \alpha)}{\beta\alpha - \gamma^2 r_w^2} \end{bmatrix} \begin{bmatrix} \theta \\ \dot{\theta} \\ \phi \\ \dot{\phi} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{2k_t k_v(-\gamma r_w - \beta)}{\beta\alpha - \gamma^2 r_w^2} \\ 0 \\ \frac{2k_t k_v(\gamma r_w + \alpha)}{\beta\alpha - \gamma^2 r_w^2} \end{bmatrix} V$$

Now that we have the state-space form of our system, we are ready to use a variety of simulation tools to perform a linear analysis. These tools all use numerical integration in order to propagate the system state over time from a known set of initial conditions. The linearized state-space model can be used to simulate a mobile inverted pendulum system only when it is near its balance point. The non-linear dynamics must be used to simulate the full mobile inverted pendulum motion. These simulators can be used as a first-cut approximation of Bobble-Bot's tilt and translational velocity dynamics. The equations do not consider turning motion. The linearized system and the state-space form is needed for controllability and observability analysis, and also for the design of pole placement controllers. This is not the focus of this work, however, so we will stop here with the theory. The full linear analysis has already been covered in many other works. See Katariya (2010) and Castro (2012) and Kim and Kwon

(2015) for a more rigorous treatment of the theory. The remainder of this work will focus on the design and implementation of a balance controller for the real system.

3

DESIGN

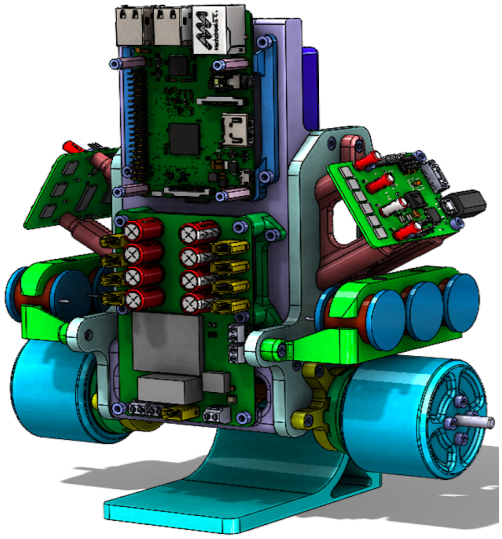


Figure 3.1: *Bobble-Bot Physical Design*

The equations of motion given in the previous chapter help to make our understanding of Bobble-Bot's dynamics more concrete. They also serve as a simplified test case that is useful during the development of higher fidelity simulations that take into account the full non-linear dynamics. These higher fidelity simulations are ultimately what is used for testing a prototype balance control algorithm. At this point, we are ready to start considering the design of Bobble-Bot. This chapter will discuss requirements and selection of the hardware and software components used to implement Bobble-Bot and the Bobble-Bot simulator.

The complexity of the Bobble-Bot system grows quickly due to the number of hardware and software modules needed for its implementation. In order to avoid developing a monolithic, complicated, and difficult to maintain system, Bobble-Bot's design makes use of a modular approach on both the hardware and software side.

When necessary, custom hardware was designed when a commercial solution was not easily available that could meet the Bobble-Bot requirements. The detailed design of the custom hardware components is beyond the scope of this work. The two custom components are the Power Distribution Unit (PDU) and the Brushless DC Motor Controller (BLDC). On the software side, ROS was selected in order to provide a unified software architecture for each of the necessary modules. This selection drives the design of the software modules used in order to implement the control law. Following ROS conventions, the Bobble-Bot software is written in a combination of C++ and Python. C++ is used for sections of the software that are optimized for speed and real-time execution. Python is used when ease of development and flexibility are prefered at the expense of performance. The sections that follow will elaborate further on the hardware components and software modules within Bobble-Bot.

3.1 Hardware Components

Figure 3.2 provides a summary of the components that make up Bobble-Bot. This diagram captures all of the hardware components, the communication channels, effector commands, state feedback, and a simplification of the operating system and controller modules. It is a helpful reference for understanding Bobble-Bot at the system level. In this diagram, the RTOS and software modules are all consolidated and grouped in order to be contained within the dashed blue line boundary. This is done to simplify the diagram and keep the focus on the major system interfaces. The RTOS and design of the embedded controller software will be elaborated upon in the sections to come. For now, we keep the focus on the hardware, communications channels, and data flow between the system's major components.

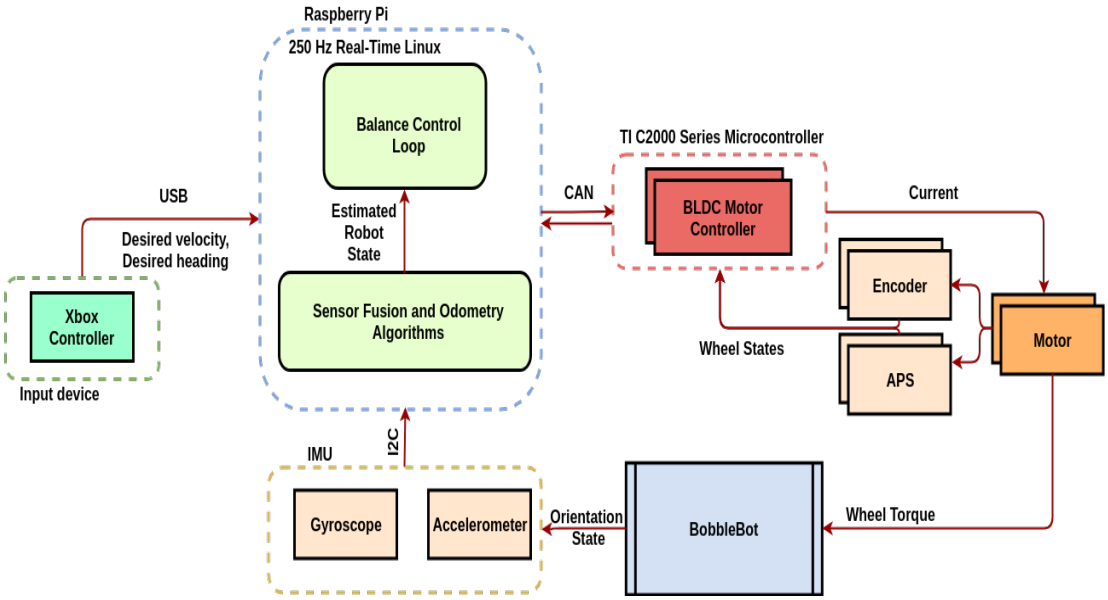


Figure 3.2: *Bobble-Bot System Diagram*

As seen in figure 3.2 the RTOS and controller software sits in the center of the system's feedback loop. A combination of three separate communications channels are employed to close this loop. The Universal Serial Bus (USB) or Bluetooth channels are used to deliver the desired Bobble-Bot states to the balance controller. These channels are reserved for sending in commands related to the desired forward velocity, turn rate, and control mode selection. They are a low bandwidth communication bus that sends the desired state command data into the balance controller at 25 Hz. In contrast, the Controller Area Network bus (CAN) is a high bandwidth channel used by the motor drive loop. This bus facilitates real-time, two-way, communication between the Raspberry Pi RPi and the BLDC motor drivers. The bus carries two motor effort command packets and two state feedback status packets to and from the RPi and BLDC motor drivers at 250 Hz. The position and velocity of the corresponding left and right motors are contained within the status packet. A normalized voltage command is contained within the left and right motor command packets. The third

and final communication channel used by Bobble-Bot is an Inter-Integrated Circuit channel (I2C) for orientation state feedback. The ADXL-345 accelerometer and ITG-3200 gyroscope use I2C to report their sensed values. These six independent readings (3 axes per sensor) are fed into a state estimation routine called by the balance controller at 250 Hz. The state estimation routine is a real-time software module that receives the sensor readings, filters them, and fuses them together to produce an estimate of the absolute orientation and orientation rate of change.

Table 3.1: *Hardware Modules with Data Sheet Ref.*

Component	**Description**	**Ref**
RPi	Embedded controller	Foundation (2016)
ADXL-345	Accelerometer	ADXL-345 (2015)
ITG-3200	Gyroscope	ITG-3200 (2011)
Li-Po	Lithium polymer battery	Turnigy (2014)
PDU	Power distribution unit	
BLDC	Motor controller	
TACON	BLDC motor	Tacon (2012)
MikroBus	RPi CAN shield	MikroBus (2014)
MCP-2515	CAN transceiver	Microchip (2012)
AS5047D	Absolute position sensor	AS-5047D (2016)

Table 3.1 lists the hardware components used by Bobble-Bot and provides their corresponding data sheet for reference. The RPi was selected primarily because it was a cheap open-hardware Linux System on a Chip (SoC) that could easily run ROS. Brushless DC motors were used because of their advantages in torque control, low power consumption, and durability. CAN hardware was selected in order to facilitate

sending command and status packets to and from the motor controllers at a high rate (1 Mbps) with limited need for additional hardware and wiring. A Lithium Polymer (LiPo) battery and PDU was selected in order to supply enough power to the 12V and 5V channels off a single supply. The battery was sized to provide Bobble-Bot with three hours of continuous operation. The ADXL-345 accelerometer and ITG-3200 gyroscope were selected to be used in combination as an orientation sensor solution. When used in combination with a sensor fusion algorithm, the sensors are capable of providing a sufficiently accurate orientation and orientation rate of change estimate. These particular sensors require only a minimal level of driver software development in order to integrate into the selected control system software architecture. The AS5047D absolute position sensor was selected in order to provide an accurate motor position and velocity reading. Figures 3.3 and 3.4 provide pictures of each of the major components and a short description.

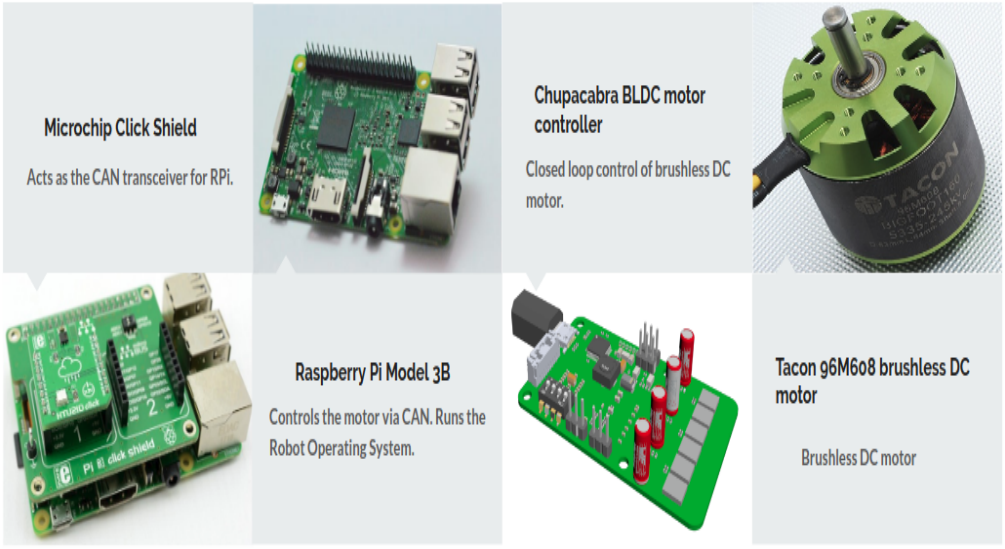


Figure 3.3: *Primary Hardware Components (1/2)*

Figure 3.4: *Primary Hardware Components (2/2)*

3.2 Assembly

Once all the hardware components were settled upon, the next major challenge was producing a mechanical design that stayed true to the TWIP reference design. The main goals for the design were as follows:

- Affordable construction
- Careful placement of the CG
- Mass less than 5 kg
- Approximate dimensions 30 cm x 25 cm x 15 cm
- Thoughtful component placement with respect to electrical wiring
- Facilitate repairs and component swapping
- Sufficiently rigid to reduce vibrations that may effect balance
- Inertial Measurement Unit IMU placement to reduce noise and coupling due to motion

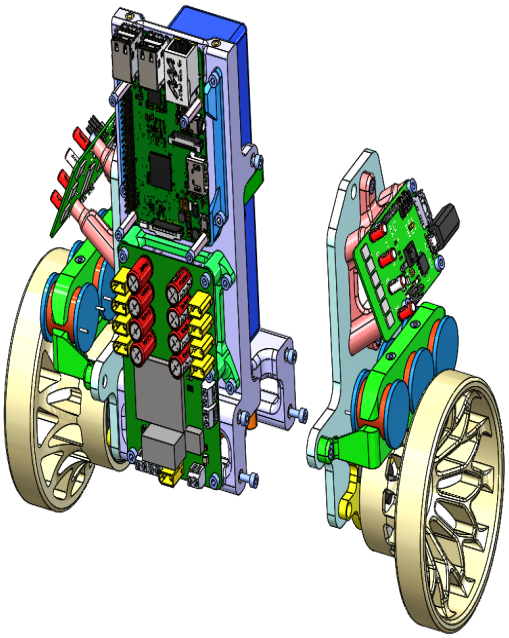


Figure 3.5: *Bobble-Bot Assembly*

Given the above goals, the use of 3D printing for manufacturing parts was a natural first choice. 3D printing the components keeps the system modular and makes it easy to change the design as new things are learned during development and testing. SolidWorks was used to design each component of the mechanical assembly. Figure 3.6 shows the actuator assembly design created in SolidWorks. Capturing the mechanical design in SolidWorks also helped track the overall system's mass properties. This ensured that the CG was kept in a desirable location. SolidWorks also has the ability to export the 3D model into a format that is easily read into 3D simulation environments. This allows the simulator to keep pace with changes made to the mechanical design. Section 3.6 provides more details on the simulation.

One potential downside to 3D printing is that parts are prone to defects and the materials used are not as rigid as other more expensive alternatives. This proved to not be an issue for this particular system. Bobble-Bot is intended for tinkerers and

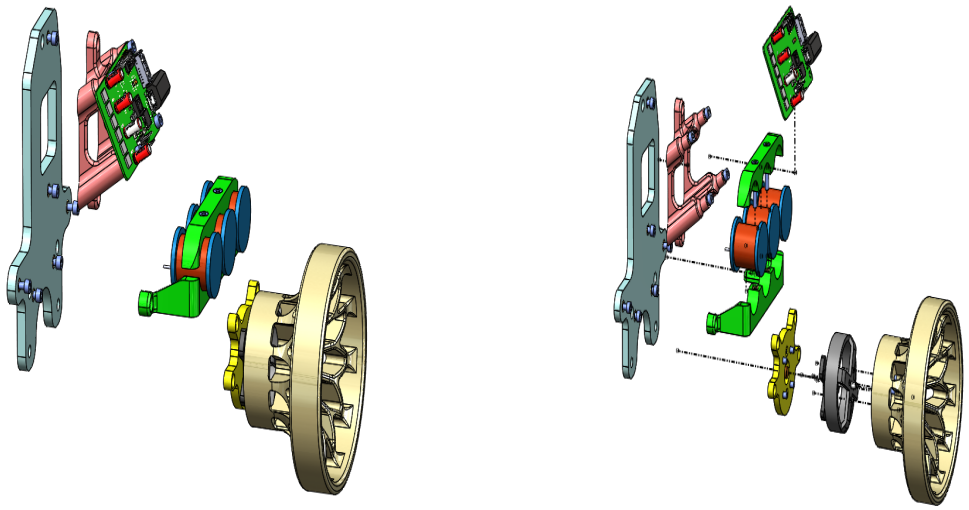


Figure 3.6: *Actuator Assembly CAD*

hobbyists interested in robotics development. As such, it does not have particularly demanding structural and thermal requirements. This makes it a great candidate for 3D printing. The selection of SolidWorks for Computer Aided Design (CAD) software and 3D printing for part manufacturing was instrumental to keeping costs down and the project on schedule. Furthermore, this approach encourages others in the community to download the design and print their own Bobble-Bot. Figure 3.7 shows the components of the left actuator assembly and how one would put it together.

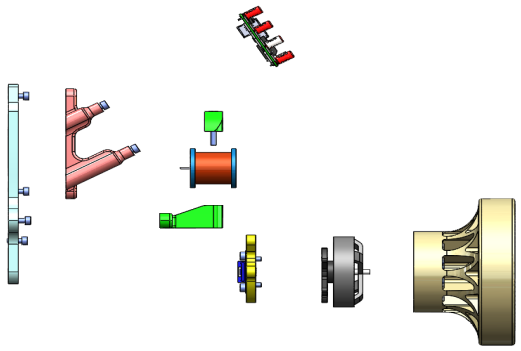


Figure 3.7: *Left Actuator Parts*

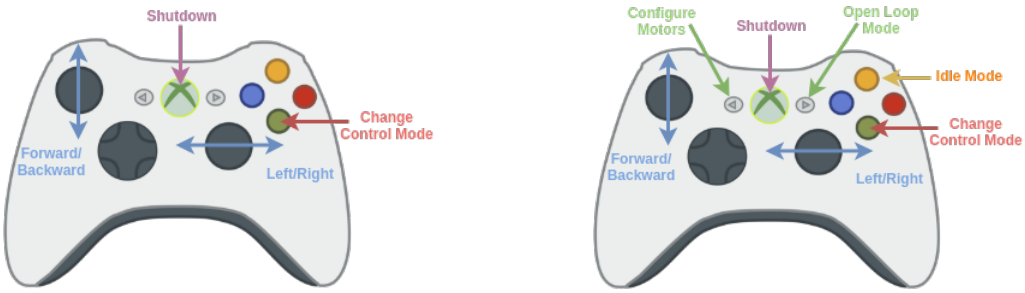
3.3 User Interface

This section focuses on how different users will interact with the robot. This is an important consideration before proceeding much further as it has an impact upon the design of the software modules and their implementation. Bobble-Bot is intended to be both a demonstration robot and an educational tool. To meet both use-cases, it is important to consider two distinct types of users: normal users and developers. Table 3.2 summarizes the important distinctions between the two classes of user.

Table 3.2: *Bobble-Bot User Specific Features*

Feature	**Normal**	**Developer**
Automatic balance mode	X	X
Manual drive mode	X	X
Emegency stop	X	X
Open loop mode		X
Motor tuning mode		X
Data logging	X	X
Real-time plots	X	X
State machine configuration		X
State machine log		X
Motor configuration		X
Programmable control gains		X
Extendable control logic		X

As can be seen from table 3.2, developers are given a special set of extra features over normal users. This effectively allows developers priviledged access to the core software modules and the configuration files that define the Bobble-Bot balance con-



(a) *User mode controls* ***(b)*** *Developer mode controls*

Figure 3.8: *User vs Developer Controls*

troller state machine, drivers, control modes, and controller gains. Normal users are restricted to a subset of the robot's state machine and configuration files. This ensures that the robot's core balance control module can operate safely with limited risk to damaging the hardware. Generally speaking, user mode is for demonstrations while developer mode opens the robot up to tinkering and simultaneously provides the greatest risk and educational value. Developer mode is enabled by issuing a special command during the robot's boot sequence which instructs the robot to launch the primary state machine using a developer mode ROS launch file.

Whether a developer or a normal user, Bobble-Bot requires a peripheral device to capture user inputs and map them to particular commands that are sent into the robot's primary state machine. The software that peforms this mapping is designed to be largely agnostic to the particular peripheral device used. This functionality is implemented in a module known as the Bobble-Bot Input Device Manager (IDM). The device manager provides an abstraction layer that allows developers to extend the framework to whatever peripheral device they would like to use to control the robot. At the time of this writing, USB and Bluetooth Xbox controllers are supported along with standard keyboard input. Figure 3.8 shows the default mapping for an Xbox controller for both user and developer mode.

Data logging and system status monitoring are an important feature available to both normal users and developers. The capability allows users to log and plot the robot's internal state variables as the system is running. Needless to say, this feature is of huge value to developers and users alike. It provides both a command line and graphical user interface that can be used to generate data for analysis, debug issues during run-time, tune control gains, among many other uses. Fortunately, the software that implements these features is largely provided out of the box when building a robot that utilizes the ROS software stack. In fact, the availabilty of this infrastructure was a primary motivator in the selection of ROS for Bobble-Bot. To enable this feature, the balance controller simply needs to publish a ROS status message using a real-time publisher. Once that is done, the ROS messaging infrastructure allows these messages to be logged to a file during run-time or plotted using a graphical user interface. Figure 3.9 shows a screenshot of these features in action during a run of the Bobble-Bot simulator. It is important to note that the logging and real-time plotting features are not restricted to simulation. The same can be accomplished when running the real system as well. Thanks to the Open-Source Robotics Foundation (OSRF) and PlotJuggler author, Davide Faconti, for building this great set of tools for robot development and providing them to the community as open-source software. The Bobble-Bot software simply uses these open-source packages in order to meet two of its most important user interface requirements. See the GitHub pages for more infomation Faconti (2018) OSRF (2018).

3.4 Software Components

The task of balancing Bobble-Bot is achieved by an algorithm that implements a linear control law in the C++ programming language executed at 250 Hz. So long as

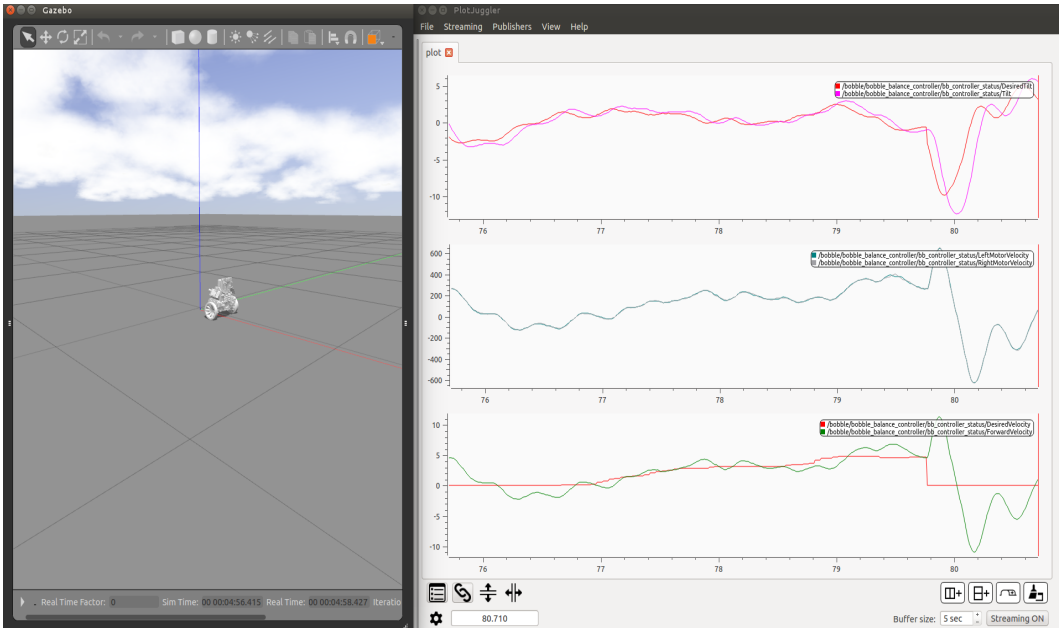


Figure 3.9: *Real-Time System Monitoring*

all the required input and output signals are supplied to and sent from the algorithm at precise timing intervals, the implementation of the control logic is straight forward. However, Bobble-Bot does not simply devote its entire focus to balance control alone. In fact, the robot is executing many processes simultaneously. For example, at any given period of time, Bobble-Bot is maintaining its balance, reading data from input devices and sensors, writing to log files, publishing messages to real-time monitoring applications, sending and receiving packets over WiFi, and running other miscellaneous developer and operating system processes. Facilitating the execution of all these processes while simultaneously guaranteeing real-time performance to the balance controller is the single greatest challenge faced by the Bobble-Bot software. This section is devoted to laying out a software architecture capable of rising to that challenge.

Before getting into the design of the Bobble-Bot software we list some of the most important motivating goals below:

- Real-time process for balance control at 250 Hz
- Provide open-source framework enabling extension of balance control algorithm
- Rely solely on open-source software libraries
- Network based robot Application Programming Interface API for command and status messages
- Support real-time data monitoring
- Provide extendable framework for input device peripherals
- Support custom developer defined non-real time tasks scheduled to be executed within the robot's primary state machine.

Real-time is not possible using the default RPi Linux kernel. In order to satisfy our most important requirement for balance control the robot's Linux kernel must be patched. Section 4.1 describes the details of how this is done. The proceeding design assumes the kernel has been re-compiled with the PREEMPT_RT patch applied. Futhermore, the RPi should then be benchmarked and shown to be capable of meeting Bobble-Bot's latency requirements.

To meet the goal of leveraging open-source software libraries, Bobble-Bot relies on the ROS ecosystem. ROS is primarily used to orchestrate the various robot processes (both non real-time and real-time). To accomplish inter-process communication, ROS provides a distributed publish/subscribe messaging system. This system helps to satisfy the goal of providing a network based API and real-time data monitoring. It is important to note that, in general, ROS's messaging framework is not real-time

capable. Care must be used when designing interfaces that need to interact directly with the hardware. To date, Bobble-Bot's non real-time processes are exclusively written in the Python programming language. ROS's messaging framework supports communication between C++ and Python processes. For this reason, developer's can take their pick on which language to use to extend Bobble-Bot's base capabilities. In general, the flexibility of Python is preferred for processes without stringent timing requirements. C++ is used for processes that need performance, and C++ is exclusively used for the balance control process.

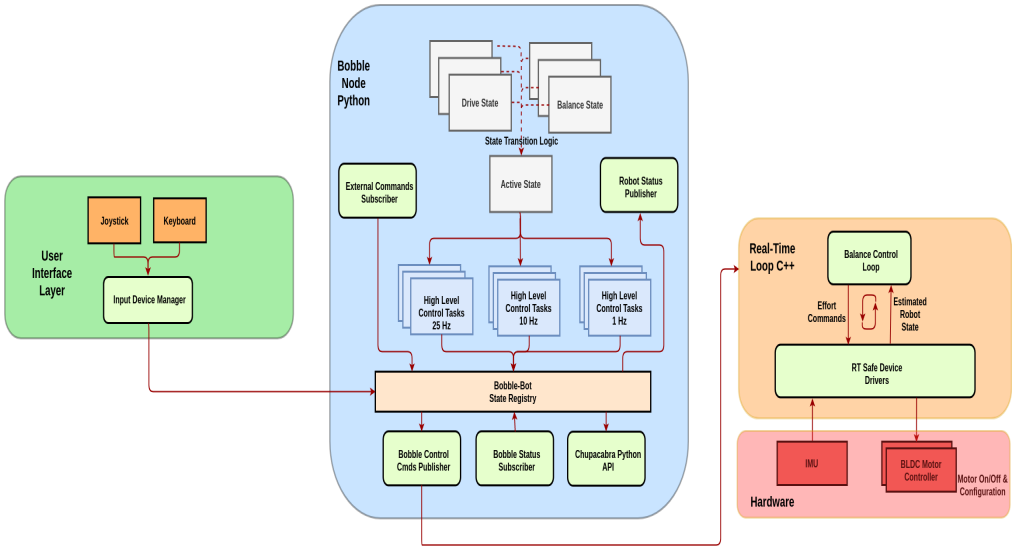


Figure 3.10: *Bobble-Bot Software Architecture*

Figure 3.10 shows the Bobble-Bot software architecture. This diagram captures some of the most important software modules and the data flow between them. At the user interface layer, input device peripherals route serial data into an input device manager module. The input device manager is an extendable framework that maps peripheral input signals to commands that are published into the robot's state registry. The Bobble-Bot state registry is a region of shared memory that can be read from and written to by non real-time processes using the ROS publish/subscribe messaging

system. The most important consumer and producer of state registry data is the robot's main execution process for non real-time tasks. This process is known as the Bobble node. This particular ROS Python node executes an extendable finite state machine at a base frequency of 100 Hz. The implementation of this state machine relies upon a state machine design pattern implemented in object-oriented Python. This pattern was selected in order to provide a framework capable of being extended to support arbitrary non real-time control tasks scheduled at a developer defined frequency. This framework is the preferred method for implementing autonomous navigation routines and other robot autonomy algorithms that may change based on the robot's state. Section 4.1 covers the implementation details of Bobble-Bot's primary state machine.

The Bobble node also contains both a ROS publisher and subscriber. The publisher is used to report robot status variables to any other applications that want to see the data. This is the mechanism that enables real-time monitoring with PlotJuggler. The subscriber listens for requests from external applications which desire to write to the Bobble-Bot state registry. The input device manager is one such example, but others are possible. The Bobble node has two mechanisms for communicating with the BLDC motor driver. The first is used to route the desired state commands into the controller through the real-time loop. The second route is through a Python API. This is a non real-time API that is only intended to be used for powering on the motors and loading the BLDC driver firmware with motor specific configuration parameters.

The balance controller is the most critical piece of software running on the robot. Details on the design of this controller will be left to a section of its own (see 3.5).

The controller software design leverages object oriented C++ in order to implement an interface that is both real-time capable and hardware/simulation agnostic. The controller also leverages a generic digital filter and PID controller module. These generic modules are configurable by setting values within a YAML Ain't Markup Language (YAML) file. YAML is a standard configuration file format commonly used by ROS packages for setting configuration values intended to be loaded in at run-time. Once the developer sets these values, a Python task running in the Bobble-Bot state machine can be invoked to reload the file at run-time. This is a very handy feature that enables tuning the digital filters and control gains as the robot is running. The typical workflow is to start the robot in the idle state and then command it to the balance state. The developer then observes the control performance while the robot is in the balance state utilitizing the real-time monitoring tools mentioned in the previous section. After making these observations, the developer commands the robot back to the idle state (motors off). Based on the observations made, the developer can then adjust filter and controller gains within a YAML file stored on the robot. This file is then reloaded by the Python state machine upon transitioning from the idle state back into the balance state. This workflow enables developers to quickly tune the balance control algorithm and easily monitor the effect of different PID and filter gains on the system.

The last important consideration for the robot's software design includes how to manage the software configuration in such a way that facilitates ease of maintenance through automated testing and adequate version control. To meet these design goals, the Bobble-Bot software adopts build, version control, and automated testing tools that are common across the ROS ecosystem. This is an often over-looked but important aspect of any set of software destined for the open-source community. The

tools selected to achieve these goals include: Catkin, Git, Google Test, ROSTest, and Python's unit-testing framework. The architecture depicted in figure 3.10 is managed as ten different ROS packages each tracked as their own Git repository. All of these packages include their own set of automated unit tests using the testing tool appropriate for their implementation. Packages that heavily rely on the ROS infrastructure are tested using ROSTest. Table 3.3 lists these packages and describes their primary function.

Table 3.3: *List of Software Modules*

Module Name	**Description**	**Language**
bldc_motor_control	Motor controller firmware	C++
bno055_ros	IMU device driver	C++
bobble_controllers	Balance controller	C++
bobble_description	Simulation source and URDF	XML
chupacabra_comm	Motor communication driver	C++
chupacabra_ros	Motor driver ROS interface	Python
configuration	Hardware and software config files	YAML
executive	Robot primary state machine executive	Python
generic_filter	Generic digital filter implementation	C++
launch	Robot launch files	XML

3.5 Controller

With the mechanical assembly, wiring, on-board computer, sensors, and effectors all in place, the remainder of the Bobble-Bot capability comes from a carefully designed control algorithm implemented in software and running on the RPi. The control algorithm implements a mathematical equation that relates system state errors to

motor torque commands. The goal of the controller is to drive the state error to zero by modulating the effort commands sent to the BLDC motor drivers. These effort commands are approximately proportional to torque for Bobble-Bot's nominal operating conditions. The algorithm is executed at 250 Hz within a single real-time process running on the RPi. This section will cover the key elements of the Bobble-Bot controller design. To start with, we list the following assumptions that were made:

- Motor controller firmware will accept torque commands at 250 Hz.
- Motor controller firmware will report motor position and velocity at 250 Hz.
- Control loop jitter will not exceed 1.5 milliseconds.
- Control loop will run no slower than 100 Hz.
- Control loop will accept translational and rotational velocity commands no slower than 25 Hz.
- Controller commanded effort is assumed to be linear with the delivered motor torque.
- Control software will support three modes: Idle, Balance, and Drive.
- Wheel radius assumed to be 5 cm.
- Robot velocity will be within the range of +- 1.5 m/s.
- Pendulum length assumed 30 cm.
- Mass properties assumptions on inertia and c.g. are given in 6.1

In addition to the assumptions above, the following design constraints arise from the physical system and its assumed operating environment.

- Motor controller firmware imposes safety limits on torque command.
- CAN bus bandwidth for motor control is limited to 850 Kbps.
- Real-time loop is limited to 250 Hz due to motor controller communication bus bandwidth.
- IMU tilt angle has a bias of approximately 6 degrees.
- Angular position and velocity estimates provided to the controller at 250 Hz.
- Angular velocity measurements noise remains within 0.4 deg/s rms.

One constraint worth elaborating on is related to the achievable motor communication bit rate. CAN 2.0 was the communication protocol selected for this application. The CAN 2.0 specification states that 1 Mbps speeds are possible at network lengths of less than 40m (which is certainly the case here). In reality, the Bobble-Bot controller's motor communication bus is limited to approximately 825 Kbps. This limitation is due to the CAN transceiver selected for RPi compatability. The crystal osciallator used on that particular circuit board limits the bus clock to 10 Mhz. This results in limiting the controller to motor communications to 250 Hz. The control performance analysis provided in section 4.4 shows why this particular limitation was deemed acceptable for this particular application. This constraint is worth noting, however, if this particular hardware architecture is to be considered for control systems requiring higher bandwidth.

3.5.1 Control Algorithm

The mathematics provided in the preceeding chapter captures the challenges imposed by the non-linear and unstable system dynamics. Despite the challenges imposed by the system dynamics, the system is controllable. The control algorithm

design proposed in this section outlines an approach to controlling the torque commanded from each motor in a manner that allows the system to adequately and quickly reject disturbance forces and torques that would otherwise work to cause the robot to topple over. In addition to controlling the robot tilt angle, it is also desirable to control both the translational velocity and the turning rate. Figure 3.11 shows a block diagram of a controller design that hopes to achieve these goals.

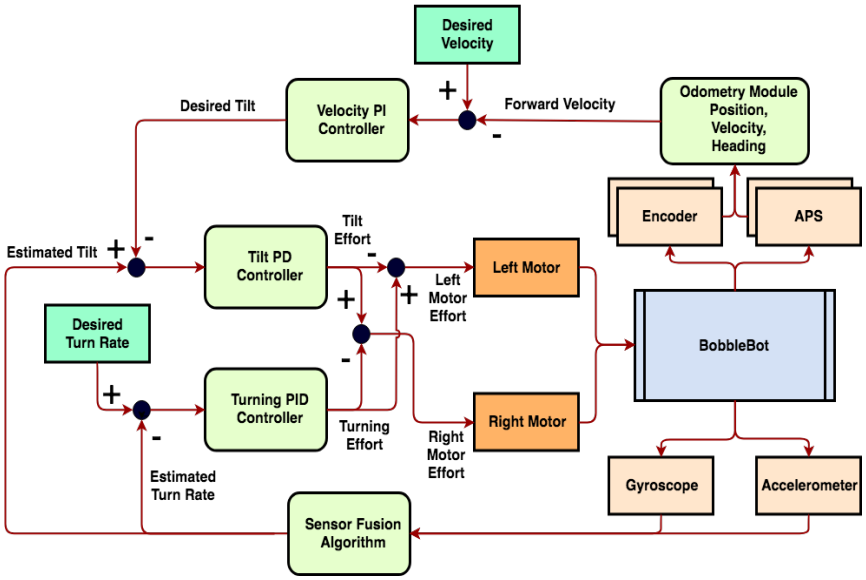


Figure 3.11: *Controller Block Diagram*

The above diagram shows that the proposed controller relies on a combination of cascaded PID controllers working together to produce two torque commands that are sent simultaneously to each motor. The control law is straight forward to implement in software. The challenges of utilizing this control design on this robot are found in the details of the implementation. Due to the unstable dynamics of this particular system, the control law needs to execute at a high rate with a fixed time step between each iteration. This particular design assumes that the maximum latency can be kept below 15 milliseconds. This turns out to be a challenging problem in and of itself. It

requires the implementation of high performance and real-time software drivers for all sensors and actuators used by the controller. It also requires modification of the RPi stock Linux kernel, and the creation of a high priority real-time thread that the control law is scheduled to execute within. The sensor and actuator drivers must also be able to move their data into and out of this real-time thread in a way that does not impose excessive latency and does not lead to undesirable race conditions. The section on real-time Linux in the implementation chapter covers the details of these particular challenges.

Another significant implementation challenge that was considered during the early phases of the controller design was related to how to test and verify the controller software implementation. It is desirable to achieve this verification in such a way that minimizes the risk of damaging the hardware. Discovering the sources of unacceptable hardware system latencies while simultaneously developing, testing, and tuning the control law is a recipe for frustration, delay, and a costly set of integrated hardware and software tests. It is much more desirable to keep the hardware and driver related development and testing entirely separate from the software development and testing of the control algorithm. This reduces the risk as it encourages the systems to be independently verified before attempting closed loop control in an integrated hardware and software testing environment. Figure 3.12 depicts a software architecture that attempts to mitigate these risks.

The architecture shown in figure 3.12 assumes the availability of a high fidelity Gazebo based simulation with the Gazebo ROS plugin loaded into the simulation environment. The development of this simulation was considered a critical milestone in the design and verification of the control algorithm. The simulator enabled controller

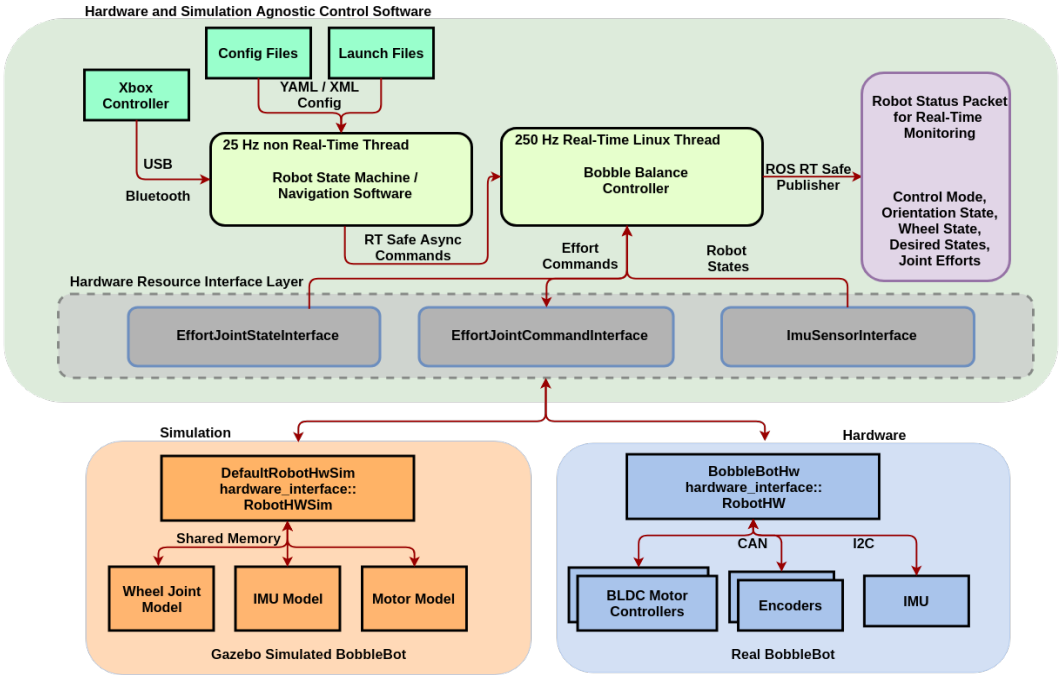


Figure 3.12: *Same Controller in HW and Sim*

testing with a purely software based approach. This allowed for a test-driven approach to control algorithm development that resulted in a more robust set of control software with a clearly defined interface. Furthermore, the simulator immediately eliminates the risk of damaging hardware due to the inevitable bug or two introduced during the early stages of writing the control algorithm. Additionally, it allows separate developers to simultaneously implement and test their control algorithms using their own instance of the Bobble-Bot simulator. This facilitates a great deal of experimentation and flexibilty when designing and testing different control schemes. This sort of flexibilty is not feasible with an approach that is reliant upon hardware testing and verification alone. For all of the reasons above, the simulator was utilized early in the design phase of the controller and it significantly reduced the development cost and schedule for this particular project.

3.6 Simulation

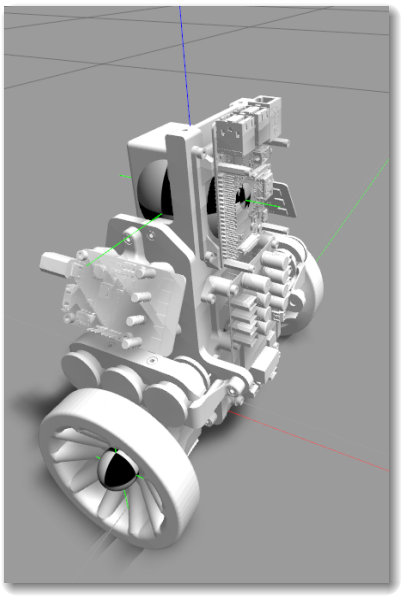


Figure 3.13: *Bobble-Bot Simulator*

There are many simulation tools available to robotics and controls developers. This particular project considered simulation tools found within Python, Matlab, and the Unreal Engine before finally settling upon Gazebo and the ROS plugin. All of these tools were used at different points during the Bobble-Bot controller and simulation development cycle. Gazebo was settled upon as the simulation framework of choice due to its support for ROS control (which was selected for the hardware). The Bobble-Bot Gazebo simulation is capable of running alongside the identical real-time control process used for the controller on the actual hardware. This makes it an invaluable tool for testing out the controller in software and immediately deploying it to hardware. It should be noted that Matlab Simulink is also capable of this, but it is closed source and was deemed cost prohibitive for this project. Furthermore, the reliance on embedded code generation was undesirable for this application. The

following sections will briefly discuss each of these tools and how they were employed during the design of Bobble-Bot.

3.6.1 Python

Python is a general purpose, interpreted programming language with a wide variety of applications. It is a great language for quick prototyping and experimentation. NumPy and SciPy are widely used free Python libraries for engineering and scientific computation. These libraries were useful for verifying the equations of motion derived in the previous chapter. A NumPy and SciPy based simulation of a double pendulum is provided in 6.3. This simulation serves as a good example of how to implement a simple 2D simulation from state equations represented in state space form. Matplotlib is used to animate the results. The system parameters are defined at the top of the script and can be easily changed prior to running the simulation and inspecting the results. While Python and its associated open-source libraries were not selected for Bobble-Bot's primary simulator, the tools are still heavily used to perform analysis on results generated by the Gazebo simulation.

3.6.2 Matlab Simulink

Matlab and Simulink work together to combine textual and graphical programming into an integrated simulation environment. Developers can immediately gain access to thousands of libraries containing algorithms authored for Matlab. This allows the developer to focus on their domain specific application. Users can combine Simulink library blocks with their own custom created blocks that wrap their application specific Matlab code. This allows users to rapidly build up system simulators by dragging

and dropping blocks from reusable component libraries. The full power of Matlab is also available to Simulink users to analyze the output of their simulations.

As an example of using Matlab Simulink to design a Bobble-Bot like system, refer to the controller design in Yamamoto (2018). This simulation utilizes Simulink to design and auto generate the controller code. The image below shows that it also comes with a 2D visualization tool.

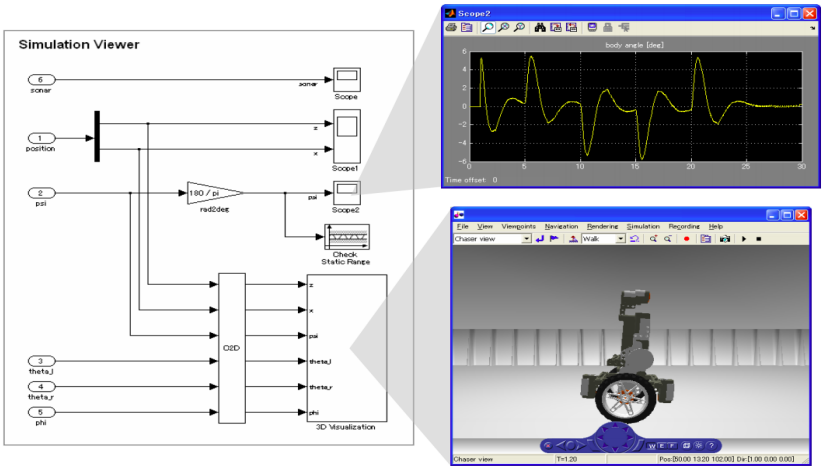


Figure 3.14: *Simulink Based TWIP Simulation Yamamoto (2018)*

3.6.3 Unreal Engine 4

Unreal Engine 4 is a suite of integrated tools for developers to design and build games, simulations, and visualizations. The engine is open-source and free to download, but developers should check the licensing agreement if they are using it to develop a commercial product. Unreal Engine is still relatively new on the scene for 3D simulation development. The engine relies on NVIDIA's PhysX physics engine. Microsoft's research laboratory has recently used Unreal Engine to build a high fidelity drone and self-driving car simulator Shah et al. (2018). AirSim is an open-

source, high-fidelity physics, and photo-realistic robot simulator. Microsoft built the simulation framework on top of Unreal Engine in order to help verify control and perception software for robot designers and developers. This effort shows that the use of Unreal Engine 4 will likely increase in popularity over the years to come. An Unreal Engine simulation of Bobble-Bot has been started but has yet to be completed at the time of this writing. A screen shot of this simulation is shown in figure 3.15.

Figure 3.15: *Bobble-Bot in Simulated Garage Environment*

3.6.4 Gazebo

Gazebo is an open-source 3D dynamic simulator that is commonly used to simulate robotic systems. While Gazebo's development environment has the look and feel of a 3D game engine, Gazebo offers physics simulation at a much higher fidelity than is typical of most video games. In addition, Gazebo also contains a large library of open-source robot, sensor, and effector models. Gazebo is the recommended simulation framework for developing and testing control algorithms developed for ROS. The following list highlights some key features.

- Library of robot models and environments
- 3D development environment
- Multiple physics engines
- Support for ROS controllers
- Understands URDF (ROS robot model description format)
- Library of sensor models

For the reasons above, Gazebo was selected as the simulation framework of choice for developing a high-fidelity Bobble-Bot simulator. This simulation is used to tune the Bobble-Bot controller gains and test out higher level control logic. Future applications include enabling the development and test of Simultaneous Localization and Mapping SLAM related algorithms and sensor modeling. The Bobble-Bot Gazebo simulator is also capable of running with hardware in the loop. The image below is a screenshot of the Bobble-Bot Gazebo simulation environment.

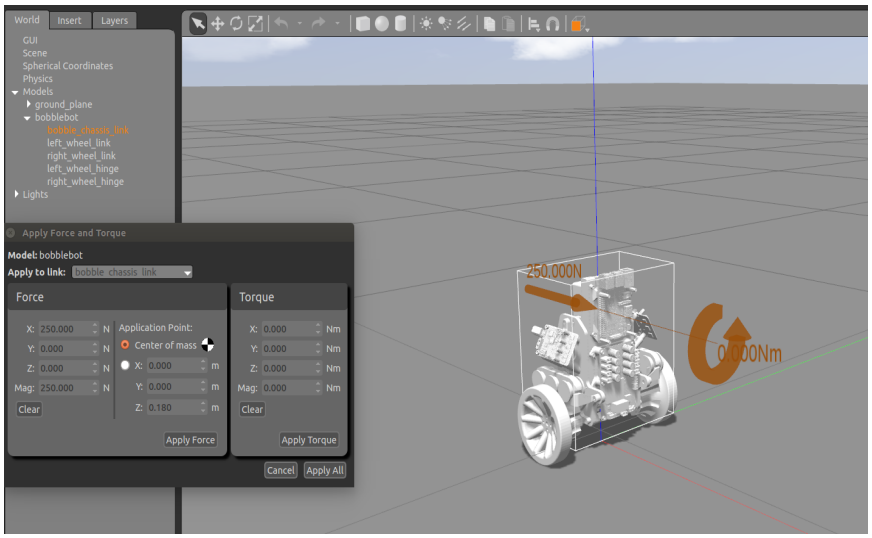


Figure 3.16: *Bobble-Bot Gazebo Simulation*

ROS control is the controller architecture that was adopted for Bobble-Bot. One of the key advantages of this architecture is that it provides carefully designed software abstraction layers that enable running real-time controllers with either simulation or hardware in the loop. The controller is agnostic about whether it is running with the Gazebo simulator in the loop or the real system. This approach allows developers to iron out any lingering bugs and tune their control algorithms using simulation. In practice, simulation based testing reduces the risk of damaging property or injuring people during controller development. The architecture depicted in figure 3.17 shows how to design a ROS controller capable of being tested in simulation and then deployed to hardware.

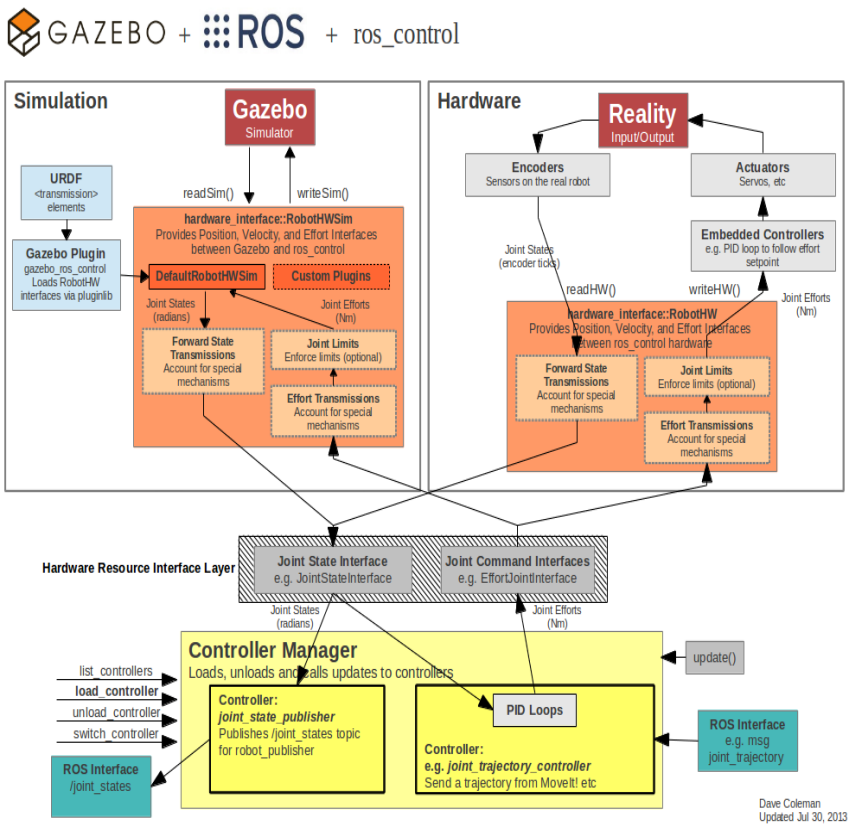
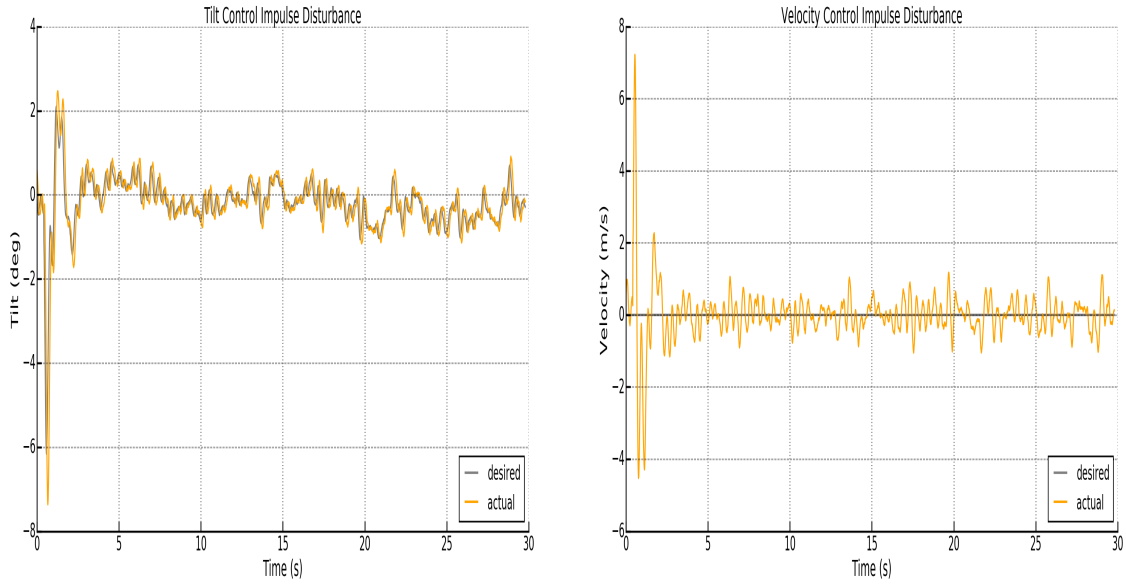


Figure 3.17: *Controller Architecture in Simulation Environment Chitta et al. (2017)*

The Bobble-Bot simulator includes a model of the mass properties, wheel friction, motor joints, gravity, and the IMU. When the simulator runs without the controller in the loop, the Bobble-Bot model immediately topples over as expected. As in real life, the Bobble-Bot model requires active control. It is only when the balance controller is run alongside the simulator that the simulated Bobble-Bot can maintain its balance. One of the nice advantages of the Gazebo simulator is that it can be executed within the ROSTest automated testing framework. This feature was used heavily in order to tune the controller by repeating many separate impulse force tests within the simulated environment. These tests involve modeling a disturbance impulse force applied to the Bobble-Bot model. This distubance force gives the simulated Bobble-Bot a jolt that the controller must adequately recover from and dampen out. The automated test is ran repeatedly as control gains are adjusted between each run. This testing process is repeated in simulation until a desirable response is settled upon. Controller data is logged during each of these runs. A sampling of the results from this type of test are shown in figure 3.18. A much more detailed tuning analysis is provided in section 4.4.

The simulator can also be used to test the controller performance under changes to the Bobble-Bot mechanical design. During early testing and tuning of the controller, it was useful to try out larger wheels. The simulator can easily accommodate test runs with varying wheel sizes. Likewise, the simulation can be used to understand the control stability under different mass properties. Figure 3.19 shows the tilt control response as the CG is shifted to different locations along the X and Z axes. These shifts represent moving the CG forward and backward (X axis), and up and down (Z axis). This simulates the effect of adding mass to different parts of the robot's structure. For example, a set of vision sensors are being considered as additions to the



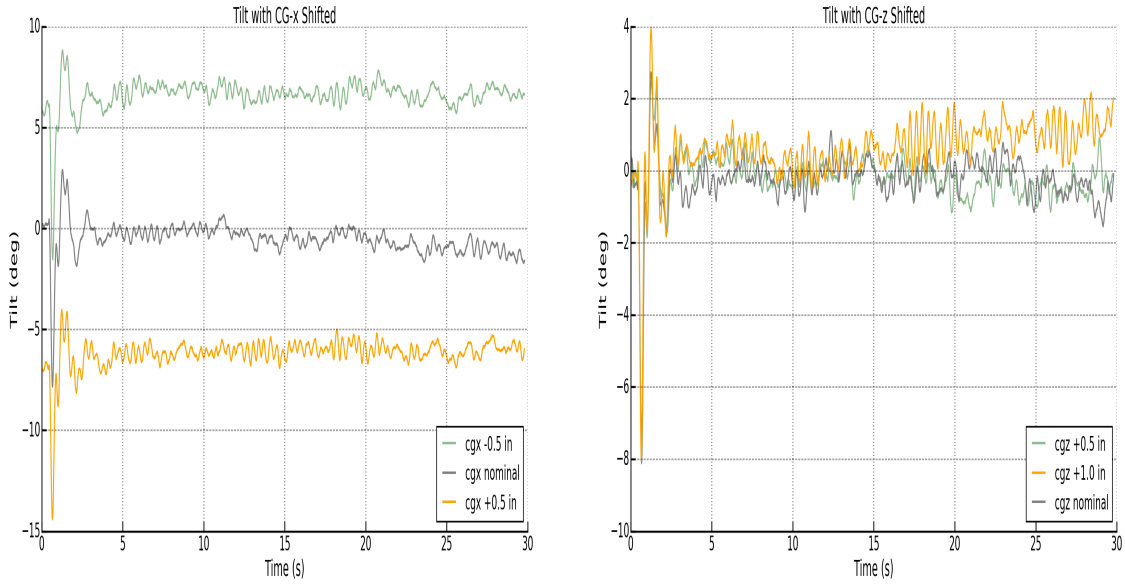
(a) *Tilt Response* ***(b)*** *Velocity Response*

Figure 3.18: *Simulated Impulse Force Testing*

top of Bobble-Bot. This will likely move the CG up along the Z axis slightly. It may also result in small shifts along the X and Y axes as well. The analysis depicted in figure 3.19 and figure 3.20 shows that the tilt and velocity controller should be capable of accommodating these changes. As one would expect, moderate shifts along the Z axis have a minimal impact to the tilt and velocity response. Shifts along the X axis cause the steady state tilt angle to change, but otherwise the system can remain balanced as long as the CG shift is not too severe.

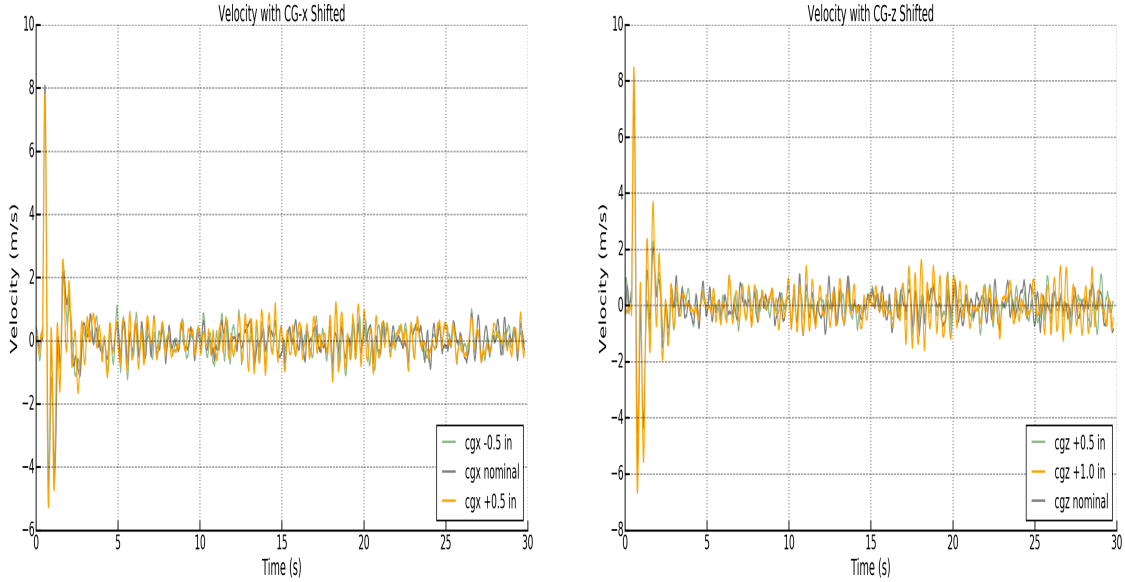
3.7 Summary

This chapter was dedicated to detailing a hardware and software design that should be capable of meeting Bobble-Bot's requirements. The results obtained by the simulator in the previous section help to establish early confidence in the controller design outlined in section 3.5. The design of the controller assumes a particular set of state



(a) CGx Shift *(b) CGz Shift*

Figure 3.19: *Tilt Control Stability With Varying CG*



(a) CGx Shift *(b) CGz Shift*

Figure 3.20: *Velocity Control Stability With Varying CG*

feedback variables can be supplied to the balance algorithm at a rate of 250 Hz. Furthermore, the controller design assumes it can transmit motor effort commands at

the same rate. The simulation results show that so long as the hardware can meet these timing constraints, the Bobble-Bot reference design should be controllable using a set of cascaded PID controllers. A preliminary analysis of the control stability under varying impulse disturbance forces and CG locations was carried out. The results from this analysis show that the proposed controller design is robust to force disturbances and uncertainties in the final CG location. In addition, the simulation has also been used to adequately test the high level state machine and input device manager. The input device manager, state machine, and controller software have been architected in such a way that the implementation is hardware independent. At this point, only the hardware device drivers for the orientation sensor and BLDC motors remain to be implemented and benchmarked. Additionally, the real-time Kernel patch must be applied to the RPi and a timing analysis must be carried out. The next chapter will cover these implementation details and will also revisit the control software implementation in more detail as it is tested once again in simulation and then finally on the hardware. To verify the implementation, hardware based tests will be carried out and the results will be compared with predictions made by the simulator.

4

IMPLEMENTATION

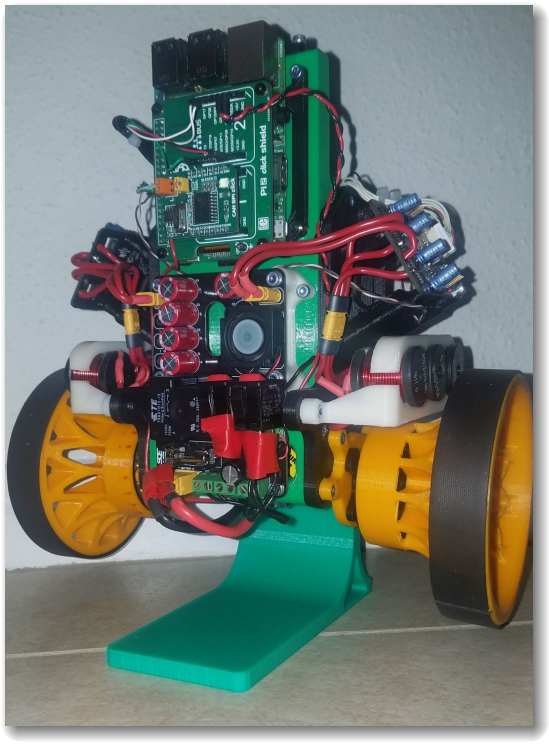


Figure 4.1: *Bobble-Bot on Test Stand*

The previous chapter provided a look at a potential controller design and software architecture that could be employed to achieve our requirements for Bobble-Bot. This chapter will focus on how this architecture can be implemented using hardware and software. As has been stated before, a software loop exhibiting deterministic timing with low latency is one of the most important pre-requisites for implementing the robot's balance controller. As such, it makes sense to begin with a description of how this real-time software loop can be achieved on the RPi.

4.1 Achieving Real-Time

Out of the box the RPi runs the Raspbian Operating System (OS). Raspbian is Debian based, and it is the officially supported OS for the RPi single board computer. Raspbian comes pre-installed with useful and commonly used software packages for education, programming, and general use. For example, Raspbian comes with Python, Scratch, Sonic Pi, Java and more. Additionally, there is a lot of documentation within the ROS community to install ROS for Raspbian. This allows developers to quickly build the OS layer for their their own ROS powered robot. Detailed instructions for how to setup ROS for Raspbian can be found in reference Gutierrez (2018).

Installing Raspbian and ROS on the RPi is the very first step in the process of building up the Bobble-Bot software stack. The stock RPi kernel used by Bobble-Bot's version of Raspbian is rpi-4.14.y. This particular kernel is a fork of the official Linux 4.14 kernel that adds RPi specific device driver code. As stated in section 1.2, the stock Linux kernel is not real-time capable. Section 3.5 of the design dictates that Bobble-Bot's control loop has a strict maximum latency requirement of 1.5 ms. Benchmarks provided in section 4.1.1 show that Raspbian's stock Linux kernel cannot meet these requirements. There is no point in proceeding with the Bobble-Bot software implementation on the RPi until the system can be proven to meet the maximum latency requirement. The steps outlined below will be followed throughout the remainder of this section. These steps show how the RPi stock OS kernel can be modified and a ROS real-time task created to help meet Bobble-Bot's performance needs.

- Build the PREEMPT_RT based Raspbian real-time kernel
- Install the kernel and configure the system for real-time

- Perform a preliminary benchmark test to assess performance margins
- Observe important considerations for programming real-time tasks
- Create the ROS real-time node that will execute the sensor/actuator drivers and balance control algorithm

4.1.1 Building the Real-Time Kernel

To transform Bobble-Bot to a real-time capable system we start with building the appropriate Linux kernel from source. The source code for the Raspbian real-time Linux kernel that Bobble-Bot uses can be found in reference RPi-Foundation (2018). Before proceeding, we note that modifying the RPi's kernel is a risky endeavor. It is highly advisable to proceed with caution and create appropriate data and software back-ups. Bobble-Bot's kernel swap took a handful of attempts before it was done correctly. With that warning, we proceed by outlining the general process. The given steps are specific to Bobble-Bot's particular kernel version, but the general process is adaptable for other real-time RPi based systems. Building the kernel on the RPi is very slow and unreliable. For that reason, it is advisable to cross compile the kernel on a more powerful host machine. The RPi uses the Advanced RISC (Reduced Instruction Set) Machine architecture (ARM). We use the tool-chain appropriate for that architecture to do the cross-compilation. The commands below capture the process for cross-compiling the Raspbian real-time kernel on an Ubuntu host machine.

Listing 4.1: *RT Kernel Build Steps*

```
1 git clone https://github.com/raspberrypi/linux.git
2 cd linux; git checkout rpi-4.14.y-rt
3 git clone https://github.com/raspberrypi/tools
4 export PATH=$PATH:./tools/arm-bcm2708/gcc-linaro\
```

```
5           -arm-linux-gnueabihf-raspbian-x64/bin
6 export KERNEL=kernel7
7 export ARCH=arm
8 export CROSS_COMPILE=arm-linux-gnueabihf-
9 export INSTALL_MOD_PATH=../rpi-kernel/rt-kernel/boot
10 export INSTALL_DTBS_PATH=../rpi-kernel/rt-kernel/boot
11 make -j4 bcm2709_defconfig; make -j4 zImage
12 make -j4 modules; make -j4 dtbs
13 make -j4 modules_install; make -j4 dtbs_install
14 mkdir -p $INSTALL_MOD_PATH/boot
15 ./scripts/mkknlimg ./arch/arm/boot/zImage \
16           $INSTALL_MOD_PATH/boot/$KERNEL.img
```

The above steps will produce the kernel image, kernel modules, and device tree binary needed for the install. All that remains is to zip up the build artifacts and move them over to the RPi. Once the build artifacts are on the RPi the commands below can be used to install the real-time kernel. Make backups before executing the removal commands.

Listing 4.2: *RT Kernel Install*

```
1 sudo rm -r /lib/firmware/
2 sudo rm -r /boot/overlays/
3 cp arch/arm/boot/zImage /boot/kernel7.img
4 cp arch/arm/boot/dts/* /boot/
5 cp arch/arm/boot/dts/overlays/* /boot/overlays/
```

After successfully completing the above steps, the RPi will now contain a kernel which is fully pre-emptible and capable of executing real-time processes. The final step is to set security limits that will enable setting an appropriate real-time priority limit of 99 and an unlimited maximum on locked memory size. The unlimited memlock setting is needed to help avoid page-faults when loading new memory into Random Access Memory (RAM) for a real-time process. This will help ensure our real-time process does not encounter a page-fault during its execution. A page-fault would cause unacceptable latency when the needed memory has to be fetched from disk. The listing below contains the necessary additions to /etc/security/limits.conf

Listing 4.3: *Real-Time Additions to /etc/security/limits.conf*

```
1 @realtime     soft     cpu          unlimited
2 @realtime     -        rtprio       99
3 @realtime     -        nice         40
4 @realtime     -        memlock      unlimited
5 *         soft     core         unlimited
6 *         hard     core         unlimited
```

With those security configurations, the system is now ready for benchmark testing. The following section covers one common procedure for confirming that the real-time kernel and configuration are working correctly.

4.1.2 Benchmarking the System

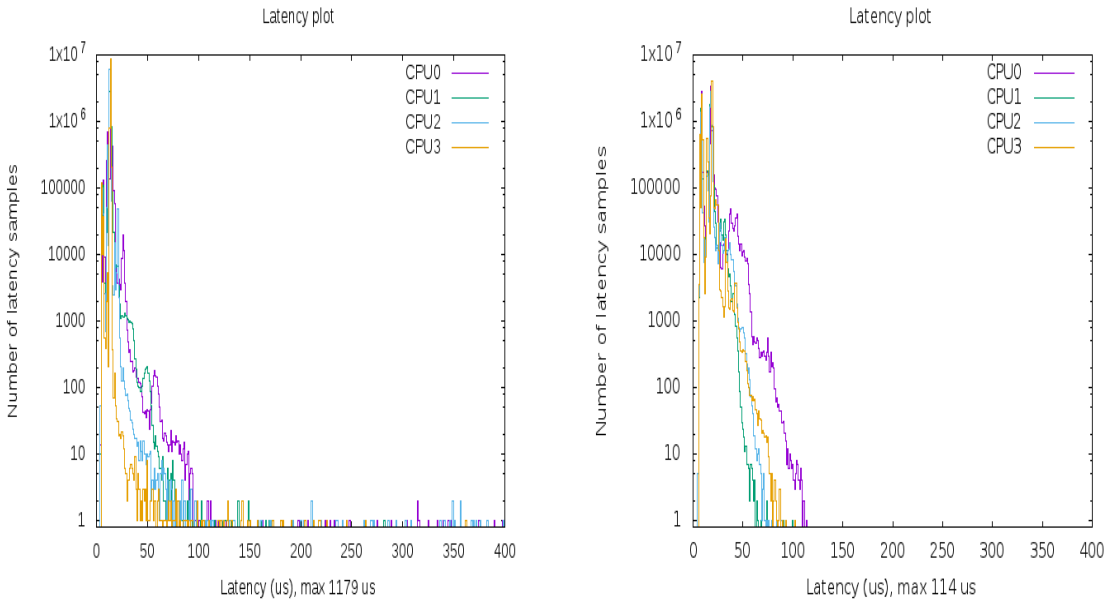
Cyclictest is an open-source real-time Linux benchmarking tool. A download link is provided in Balci (2017). Results from cyclictest are the most frequently cited real-time Linux metric. As such, it serves as a great first check on Bobble-Bot's

real-time performance. The fundamental concept of cyclictest is very simple. The program attempts to measure the latency of response to a stimulus (external interrupt trigger). However, the implementation details of the actual source code for cyclictest and the interpretation of its results can get complicated. The pseudo code provided below conveys the basic idea of cyclictest.

Listing 4.4: *Cyclictest Pseudo-code Rowand (2013)*

```
clock_gettime((&now))
next = now + par->interval
while (!shutdown) {
    clock_nanosleep((&next))
    clock_gettime((&now))
    diff = calcdiff(now, next)
    // update stat-> min, max, total latency, cycles
    //update the histogram data
    next += interval
}
```

The implementation details of cyclictest are beyond the scope of this work. The pseudo code above conveys the main idea and is enough to get a basic understanding of the benchmark results that were used to establish confidence in proceeding with Bobble-Bot's balance controller. The following references are provided to give the reader a starting point for using cyclictest on their own systems and then interpreting the results Balci (2017) Rowand (2013). Figure 4.2 shows the results of running cyclictest on Bobble-Bot's stock kernel and then the real-time kernel each for six hours under a nominal OS load.



(a) Cyclic Test non-RT *(b) Cyclic Test RT*

Figure 4.2: *Comparison of Cyclic Test Benchmark*

As expected, the real-time kernel decreases the maximum latency of the system. Generally, benchmarking a real-time system is not a straightforward task. The results from purely software based tests like cyclictest are certainly open to interpretation. This first result is mostly used to confirm that the real-time kernel is in fact improving the system's latency. The real confirmation comes when integrating the real-time balance controller with the device drivers and hardware. The final confirmation on performance is done by using an oscilliscope to verify proper timing on Bobble-Bot's sensor and actuator I/O.

4.1.3 Best Practices for Real-Time Programming

Now that we have a real-time capable system, it is wise to do some research on best practices for real-time programming before attempting to write the balance controller. In general, real-time programming is all about writing software that can guarantee a

response within a specified time constraint. In practice, this amounts to a real-time capable scheduler that can wake up a task at a precise time. From there, it is up to the task programmer to avoid a situation where the task can overrun its allotted time. To do that, the task programmer must carefully manage the task's memory and use of peripheral I/O. Linux-Foundation (2017) and Kay (2018) provides some helpful starting material to real-time programming in general. Kay and Tsouroukdissian (2015) adds detail on the specifics for accomplishing this within a ROS based system.

The basic unit of execution in a real-time application running under PREEMPT_RT is a POSIX thread (pthread). In general, a real-time application may make use of many threads, some of which may be running concurrently. In order for these threads to meet the application's timing requirements, the programmer must consider the fundamental concepts of real-time programming including: scheduling, priority, memory locking, and stack prefaulting.

Scheduling and priority go hand in hand. The programmer is responsbile for configuring this for all the real-time threads in their particular application. The scheduling policy and priority is all about allowing the kernel to know which real-time threads can and cannot be interrupted by the others. Selecting scheduling policies and defining the application's thread priorities should be one of the first steps taken by the real-time application programmer. Bobble-Bot uses just two real-time threads. One for running the sensor/actuator I/O, and one for running the balance control algorithm. The thread responsible for reading and writing to hardware is the highest priority thread. These thread priorities were intentionally set in such a way so that the sensor/actuator I/O is not interruptable by the balance controller. The balance

control law is still a high priority real-time thread though, and as such, it is not interruptable by other OS tasks and non real-time programs.

The other important consideration for a programmer writing a real-time application is proper memory management. This is a tricky and critical aspect for ensuring real-time performance. The main goal here is to eliminate the possibiliy of a page fault occuring within the real-time code path. Page faults cause the CPU to hault execution on the task so that the missing memory page can be loaded from disk into RAM. Loading data from disk is non-deterministic and slow. In normal computing, these page faults are necessary because the RAM is a finite size and could become exhausted during general use of the system. In these situations, page faults are necessary in order to relieve memory pressure on the system.

The real-time application programmer should understand the memory size required by their real-time threads preferably at compile time. In this way, the programmer can set the appropriate stack size by using POSIX's pthread_attr_setstacksize function call. In addition to setstacksize, the programmer should use the Linux system call, mlockall, in order to lock the process's virtual address space into RAM. This prevents the memory that will be used by the process from getting paged into swap space. It is possible to do dynamic memory allocations in a real-time way but it requires even more care. All of the required memory for Bobble-Bot's real-time loop is statically allocated, so this was not a concern for this application. Kay (2018) provides more information on how to do dynamic memory allocations in a way that is safe for real-time applications.

Another important consideration for real-time applications comes with the design of real-time capable interface mechanisms with non real-time threads. Bobble-Bot's

balance controller is designed to receive desired state commands from other non real-time processes. Refer to the software architecture design in section 3.4 for details on how the software design carefully separates the non real-time processes from the real-time one. ROS's standard publish/subscribe mechanism for process to process communication is not real-time safe and cannot be used in the usual way within the real-time code path. To solve this problem, the balance controller checks a statically allocated section of memory containing an external commands structure that is protected via a mutex. The standard ROS subscriber callback mechanism is then used to receive commands from non real-time processes asynchronously within a separate non real-time thread. The callback function receives the external commands and then waits to acquire the mutex before populating the section of memory that the real-time process is using. In this way, the balance controller can safely and regularly check for any new commands. In the worst case, it reads stale desired state data for a few frames. This is acceptable because it does not cause the execution of the control law to wait, and therefore it poses no risk to the robot becoming unstable.

In summary, real-time programming is possible only when the programmer takes extra care over managing the priority, scheduling, memory access, and I/O for the real-time process. As long as this is done, a real-time ROS node is possible. The next section provides a bit more detail on the application of the principles covered in this section in order to create a real-time ROS node dedicated to the execution of the balance controller and sensor/actuator I/O. The list below contains a final summary on best practices for real-time programming. A full list is found in Linux-Foundation (2017) and Kay (2018).

- Avoid sources of non-determinism in real-time code (networking, file I/O, etc)
- Keep disk reads/writes outside of the real-time code path

- If logging is necessary, spin up non real-time threads to log the data of interest
- Pre-allocate resources for the real-time thread in the non real-time path
- Lock the process address space into RAM to avoid page faults
- Create real-time threads at the start of the real-time program

4.1.4 A Real-Time ROS Node

At the most basic level, a node in ROS is simply an executable process. Typically ROS nodes also make use of some of the various ROS client libraries like the parameter server and the publisher/subscriber infrastructure. Armed with the real-time kernel built in section 4.1.1, this process can be made to run real-time just like any other process that is programmed carefully. All that is left to do to start writing a real-time ROS node is to observe the best practices outlined in the previous section. The code snippet below shows the initialization steps of Bobble-Bot's real-time node. Notice the use of the POSIX scheduler API and Linux system call mlockall. This bit of code comes straight from the references given in section 4.1.3. The other important bits to note are the declaration of the BobbleBotHw and ControllerManager objects. BobbleBotHw contains the hardware interface performing sensor/actuator I/O. The controller manager is responsible for running the balance control law. More detail on these two components will be provided in sections 4.2 and section 4.3.

Listing 4.5: *Setup of Real-Time ROS Node*

```
1  // Placed inside very start of main.cpp
2  struct sched_param param;
3  param.sched_priority=sched_get_priority_max(SCHED_FIFO);
4  if(sched_setscheduler(0,SCHED_FIFO,&param) == -1)
5    ROS_WARN("Failed to set real-time scheduler."); return -1;
```

```
6  if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1)
7    ROS_WARN("Failed to lock memory."); return -1;
8  ros::init(argc, argv, "bobble_bot_control_node");
9  BobbleBotHw bobble_bot;
10 controller_manager::ControllerManager cm(&bobble_bot);
11 ros::AsyncSpinner spinner(1); spinner.start();
12 bobble_bot.init();
13 // ... on to the rest of the RT-loop
```

The code above handles setting the priority, locking the processes memory into RAM, and statically allocating the needed components of the real-time process on the stack. These are the first lines of Bobble-Bot's real-time main function. After initialization, all that remains is the code for the real-time loop. The code snippet below shows that this loop is implemented as an infinite while loop. It is a simple read, update, write loop.

Listing 4.6: *ROS Real-Time Loop*

```
1  ros::NodeHandle pnh("~");
2  double loop_rate; pnh.param("LoopRate", loop_rate, 200.0);
3  ros::Rate rate(loop_rate);
4  ros::Time prev_time = ros::Time::now();
5  while (ros::ok()){
6    const ros::Time time_now = ros::Time::now();
7    const ros::Duration period = time_now - prev_time;
8    prev_time = time_now;
9    bobble_bot.read();
10   cm.update(time_now, period);
11   bobble_bot.write();
```

```
12      float period_sec = period.toSec();
13      if(period_sec > (0.03/loop_rate))
14          ROS_ERROR("SIGNIFICANT DELAY: %f", period_sec);
15      rate.sleep();
16  }
```

In the code above, the lines prior to the while loop are actually still part of the initialization. A ROS node handle is constructed to allow for setting of the loop rate via the ROS parameter server. The loop rate is defaulted to 200 Hz, but other loop rates 250 Hz and below are possible. The read and write calls are methods defined in the BobbleBotHw interface class. They are the hooks to call the sensor/actuator drivers to read and write data to and from the hardware. The update call is a method defined in the ros_control ControllerManager class. Section 4.3 will provide more detail on the ControllerManager. Finally, note that the elapsed time between iterations of the loop is monitored and stored within a delta-t variable. An error message is written out if the delta-t becomes larger than three percent of the loop rate. This is the source code implementation of the 0.15 ms latency requirement dictated in section 3.5. It should not occur during proper real-time execution. Figure 4.3 shows a plot of this delta-t variable during the execution of a Bobble-Bot real-time loop verification test. This plot shows the major result of this section. The Bobble-Bot real-time loop has been implemented correctly on the RPi, and the maximum latency has been verified to meet the 0.15 ms maximum latency requirement. The loop serves as the real-time process that orchestrates the execution of the hardware control aspects of the robot. The next sections will cover what exactly is happening during the execution of this loop, and how it enables the robot to keep its balance while also responding to desired state commands issued by the human driver.

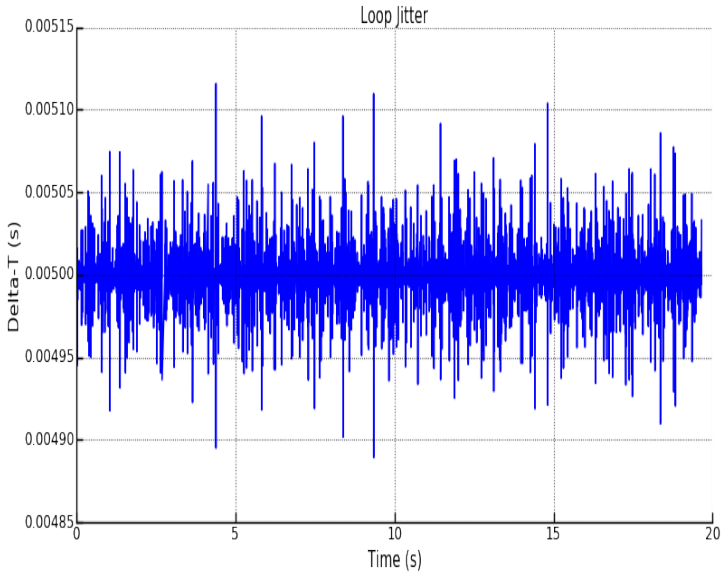


Figure 4.3: *Bobble-Bot Real-Time Verification*

4.2 Device Drivers

Now that the timing of the real-time loop has been verified, the next step is to start incorporating hardware components and device drivers. Proper implementation of these drivers is critical for achieving robust balance control. In fact, the latency shown in figure 4.3 is in part due to the time required to send and receive data over the sensor and actuator communication channels. Keeping this time below 0.15 ms helps to ensure that the balance controller has sufficient margin on control bandwidth. The sensor and actuator I/O all happens within device driver functions called during the read and the write part of the real-time loop. There are three drivers executed within this loop. They are listed below.

- CAN bus communication driver.
- BLDC motor device driver.
- IMU driver using UART.

4.2.1 CAN Communications

Bobble-Bot uses a CAN bus to execute the motor drive loop. The communication channel is two-way between the RPi and the BLDC motor drivers. In order to meet the control loop requirements, the bus must carry two motor effort command packets and two state feedback status packets to and from the RPi and BLDC motor drivers at 250 Hz. In addition to these packets, the CAN bus is also used to transmit motor configuration parameters. This motor configuration packet is only sent during the tuning modes of Bobble-Bot's operation. It is not sent during the execution of the real-time loop. To facilitate ease of use, this configuration packet can be sent from non real-time tasks. This is done using a Python interface to the CAN hardware. Though the device driver supports both non real-time and real-time access to the CAN bus, the motor torque commands should only be sent from within the real-time loop. This is the only way to guarantee proper use of the motors for balance control. It is, however, appropriate to use the non real-time interface via the Python driver module for motor diagnostics.

Table 4.1: *Bobble-Bot CAN Devices*

Device Name	**Description**	**ID**
Left Motor Tx	Left motor commands channel	0x101 (257)
Left Motor Rx	Left motor status channel	0x201 (513)
Right Motor Tx	Right motor commands channel	0x102 (258)
Right Motor Rx	Right motor status channel	0x202 (514)

The MCP-2515 CAN transceiver was selected because of its compatibility with the RPi and availability within an inexpensive, off the shelf, RPi extension module. The MikroBUS click shield is a general purpose I/O extension board that can fit

directly on top of the RPi. It enables I/O peripheral expansion and has support for the MCP-2515 CAN transceiver. A pair of wires, the MikroBUS click shield, an MCP-2515, and two BLDC motor controllers make up the Bobble-Bot CAN bus physical network implementation. Table 4.1 summarizes the CAN network including device addresses. The software implementation is composed of the MCP-251x Linux driver, an RPi specific MCP-2515 device tree overlay, SocketCAN, and a Bobble-Bot specific real-time C++ and non real-time Python driver. Figure 4.4 depicts the CAN driver implementation including hardware components. The relevant RPi CAN configuration files are included in 6.4.

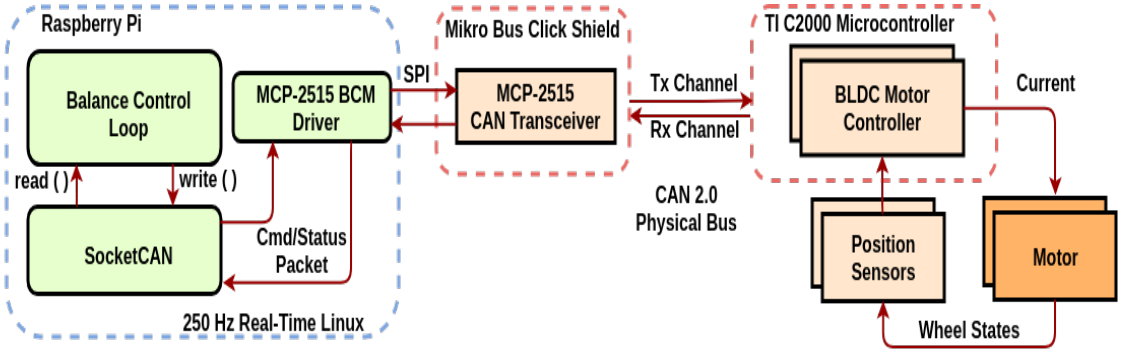


Figure 4.4: *Bobble-Bot CAN Communications Architecture*

As can be seen in figure 4.4, Bobble-Bot's real-time loop makes function calls into SocketCAN during the execution of its read and write functions. SocketCAN is a collection of open-source packages used by the Bobble-Bot C++ and Python CAN device drivers. The open-source libraries can be found in Hartkopp (2015) and Thorne (2018). This open-source software is an implementation of CAN protocols for Linux. SocketCAN uses the Berkeley socket API and the Linux network stack to implement CAN device drivers as network interfaces. This is a useful architecture as it allows the higher level, Bobble-Bot specific, CAN drivers to interact with its devices in a

way that is reminiscent of socket programming. Figure 4.5 depicts the different layers that make up a system utilizing SocketCAN.

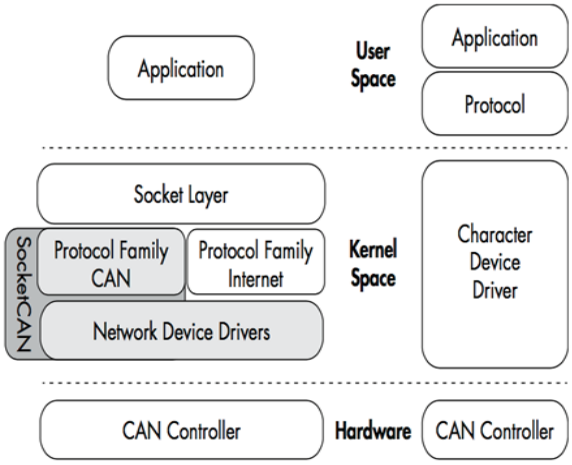


Figure 4.5: *SocketCAN Architecture Linux-Foundation (2012)*

All of these hardware and software components work together to stream motor command and status packets at 250 Hz over the two Bobble-Bot CAN wires. Figure 4.6 shows a snapshot of these packets from the SocketCAN utility candump. This utility allows users to sniff the packets being sent over the network. In the next section, we verify that these CAN packets are able to perform their basic function of commanding torque and reporting wheel status data from the BLDC motor driver.

```
pi@bb01:~ $ candump can0
  can0  102   [8]  07 A5 00 00 00 00 00 00
  can0  201   [8]  79 B9 2C 40 00 00 00 00
  can0  201   [4]  00 00 00 00
  can0  202   [8]  FD 97 78 40 00 00 00 00
  can0  202   [4]  00 00 00 00
  can0  101   [8]  07 A5 00 00 00 00 00 00
  can0  201   [8]  C2 BF 2C 40 00 00 00 00
  can0  102   [8]  07 A5 00 00 00 00 00 00
  can0  201   [4]  00 00 00 00
  can0  202   [8]  FD 97 78 40 00 00 00 00
  can0  202   [4]  00 00 00 00
```

Figure 4.6: *Bobble-Bot CAN Messages*

4.2.2 Motor Driver

Most of the motor control functionality is implemented as firmware on the Chupacabra BLDC motor controller. This is a custom designed, general purpose, BLDC and DC motor controller. Figure 4.7 shows the design of the Chupacabra circuit board. The details of the design of this board and its firmware are beyond the scope of this work. The Bobble-Bot balance controller treats its motor controllers as two separate devices on its CAN bus. It sends each an effort command, and receives back motor position and velocity. How exactly the effort command is translated into current and torque is not of any concern to the balance controller as long as the torque is approximately linear with effort and the motor feedback signals are reliable. As such, the motor driver software implemented on the RPi turns out to be a light layer built on top of SocketCAN. In the read portion of the real-time loop, this particular software layer unpacks the received motor status into a structure that is made available to the balance controller. In the write portion it packs two effort commands into the appropriate CAN message structure and sends it out over the network.

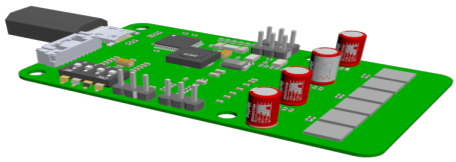


Figure 4.7: *Chupacabra BLDC Motor Controller*

The other function of the Bobble-Bot side of the motor driver interface is related to motor configuration. This is a necessary and practical feature that limits the number of hardcoded values placed inside an otherwise generic set of motor controller firmware

running on the Chupacabra. The Chupacabra CAN message exposes an interface for setting all of its configuration parameters. The Bobble-Bot side of the interface just needs to provide a convenient mechanism for setting these parameters. This functionality is implemented by defining a motor configuration task that is scheduled to be executed within a motor tuning state defined in Bobble-Bot's non real-time state machine executive. When a developer wants to change motor control constants they set values defined within a YAML configuration file and press a button to command the robot into the motor tuning state. Once in this state, Bobble-Bot's state machine executes the motor configuration task and the Python-CAN driver is then used to send down the values to each Chupacabra. Table 4.2 provides a brief listing of some of these configuration constants and their units.

Table 4.2: *BLDC Motor Configuration*

Parameter Name	**Description**	**Units**
Electrical Offset	Angle between A phase and encoder north pole	r
Electrical Direction	Motor spin in relation to + counts	–
EMF Constant	Motor speed in relation to voltage	r/Vs
Torque Constant	Torque in relation to current	Nm/A
Phase Resistance	Captures motor resistance	Ohms
Phase Inductance	Captures motor inductance	Henrys
Number of Poles	Number of BLDC motor poles	–

The real-time performance of each motor driver was verified on the test stand shown in figure 4.8. This test stand was used to test the motor controllers out prior to designing the Bobble-Bot assembly. A sample of the results from these early tests is shown in figure 4.9. These results show that the motor drivers performed as expected.

These tests gave confidence in the use of the driver software and hardware for the full Bobble-Bot assembly.

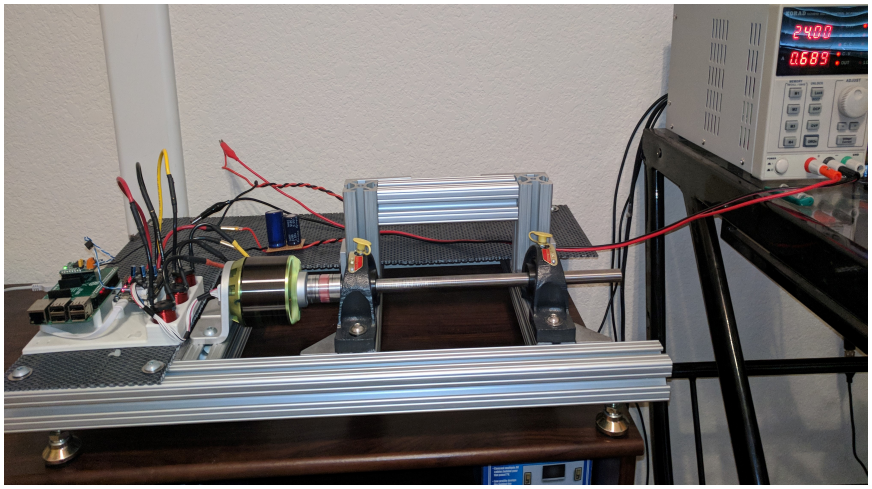


Figure 4.8: *Motor Test Stand*

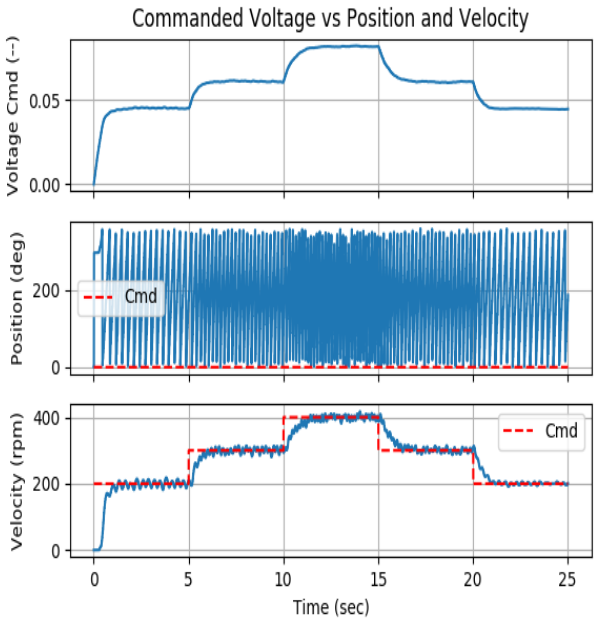


Figure 4.9: *Motor Driver Verification*

4.2.3 IMU and State Estimation

The last device driver executed within the real-time loop is the IMU. The IMU is actually composed of two separate sensors, the ADXL-345 (2015) accelerometer and the ITG-3200 (2011) gyroscope. A summary of relevant sensor specifications is provided in table 4.3. The readings from these two sensors are fused together using an orientation filter to produce an absolute orientation and orientation rate estimate. While the orientation filter produces a full orientation and orientation rate estimate, only the tilt, tilt rate, and turning rate are needed. During each cycle of the real-time loop, the IMU driver reads three accelerometer measurements and three gyroscope measurements. These values are read off the I2C bus and then handed to the orientation filter. The end result is tilt, tilt rate, and turning rate supplied in to the balance controller at 250 Hz.

Table 4.3: *IMU Sensor Summary*

Parameter Name	**ITG-3200**	**ADXL-345**
Voltage Range	2.1 - 3.6 V	2.0 - 3.6 V
Temperature Range	-40 - 185 F	-40 - 185 F
Size	4mm x 4mm x 0.9mm	3mm x 5mm x 1mm
Digital Output	I2C	I2C
Full Scale Range	+- 2000 deg/s	+- 16 g
Noise	0.4 deg/s rms	2.2 LSB rms

The orientation filter used by Bobble-Bot processes raw accelerometer and gyroscope readings using an analytically derived, optimized, gradient-descent algorithm. This algorithm computes a quaternion derivative representing the direction of gyroscope measurement error and then removes it from the estimated orientation and

orientation rate. The filter algorithm was created by Sebastian O.H. Madgwick. The details of the filter design, experimental results, and source code can be found in Madgwick (2010). This filter was selected for Bobble-Bot's IMU driver because it provided the greatest performance while still maintaining a low computational load at the 250 Hz target loop rate. The filter is considerably easier to implement and tune over the more traditional Kalman and Extended Kalman filter options. This filter does not require any least squares curve fitting or complex tuning processes. In fact, for the Bobble-Bot application, the Madgwick filter has only one gain that must be tuned experimentally. The general observation from tuning this filter is that increasing the gain leads to faster gyroscope bias correction at the expense of higher sensitivity to noise in lateral accelerations. A value of 0.20 was settled upon for this application.

With the orientation filter tuned, all of the hardware device drivers needed by the balance control loop's read and write routines are ready for use by the balance controller. The next section will detail the implementation of a control algorithm that can compute a desired torque based on the state feedback estimates provided by the Bobble-Bot device drivers.

4.3 Control Algorithm

Bobble-Bot's balance control is implemented within a single class inheriting from an EffortJointInterface. The control law is executed during the real-time loop with a call to the ControllerManager update function. The EffortJointInterface and the ControllerManager are C++ classes defined within the ROS control library. ROS control is an open-source library found within the ROS ecosystem which is intended

for implementing real-time controllers capable of interacting with both hardware and simulation Chitta et al. (2017). The architecture of a generic robot that utilizes ROS control was outlined in section 3.5. The implementation of Bobble-Bot's balance control algorithm conforms to this architecture, and in doing so, it enables several useful features found within ROS control that simplify the verification process. One of the most useful of these features is provided by ROS control's ControllerManager class.

The ControllerManager follows the Singleton design pattern Gamma et al. (1995). It encourages developers to inherit from known base class controller types, like the EffortJointInterface. The developer can then implement their custom controller and compile it as a C++ shared library. The ControllerManager is instantiated and initialized once during the pre-allocation stage of the real-time loop. Any controller which implements one of the standard interfaces can then be registered at run-time during this initialization stage. Once all potential controllers are registered, the update function of the ControllerManager is then called once during each cycle of the update loop.

The ControllerManager provides a convenient API which can be used to start, stop, restart, and swap controllers in and out of the system at run-time. This abstraction enables many useful features. For example, Bobble-Bot's on-the-fly tuning feature is accomplished by invoking methods within the ControllerManager API. To do this, the non real-time state machine commands the robot to the idle state and invokes a ControllerManager method to temporarily suspend the balance controller. It then uses the ROS parameter server to reload the controller configuration file. After the file has been loaded, the non real-time state machine once again uses the ControllerManager

API to restart the controller it had previously suspended. Once the controller has been restarted, the non real-time state machine can command the robot back into the balance state. The end result is a safe way to tune the controller without having to manually restart the robot. All that a developer has to do to gain this sort of flexibility is have their particular controller implement a standard interface defined within the ROS control library. The rest comes for free thanks to the ControllerManager.

Beyond ROS control, Bobble-Bot's control algorithm also makes use of two other open-source libraries that are worth briefly mentioning. The first is a generic PID controller implementation called MiniPID. MiniPID is a self contained and efficient C++ PID controller implementation that is designed to be used on any embedded system requiring PID control Sheadel (2015). In the case of cascaded PID controllers, like Bobble-Bot's balance controller, the developer simply creates and configures multiple instances of the MiniPID class and feeds the output signal of one PID controller object into the input signal of another. Bobble-Bot uses three instances of the MiniPID controller class in order to implement the three cascaded PID controllers depicted in the balance control block diagram shown in figure 3.11 from section 3.5. The second open-source library used within Bobble-Bot's balance controller is a generic digital filter found in Moore (2017). This library contains a C++ implementation of the generalized 1D digital filter shown in figure 4.10. This library can be used to implement simple low pass filters (LPF) or others like the rolling average, Butterworth, Bessel, and Chebyshev filters.

The library works by statically allocating the internal input and output buffers used by the digital filter. By default the maximum size of each buffer is twenty, but this default value is easy to change if necessary. Bobble-Bot's controller uses

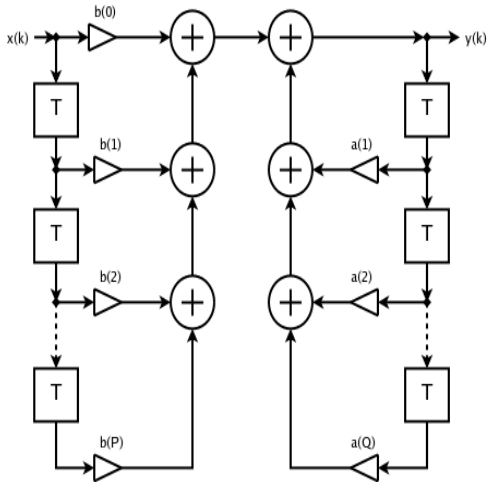


Figure 4.10: *General 1D Digital Filter*

this library to implement a generic low pass filter on all state feedback variables fed into the controller. This is done to cut out high frequency changes to the controller input signals in order to reduce undesirable system vibrations. The low pass filter implementation is given by the equation below, where x_0 is the current sensed value, y_0 is the filtered output, y_1 is the previous filter output, and α is the LPF gain.

$$y_0 = x_0(1.0 - \alpha) + y_1\alpha$$

MiniPID and the digital filter libraries work together to implement the underlying mathematics captured in the balance controller block diagram. In the real system, it is not enough for the controller to simply execute this math during every update cycle. The controller also needs to operate in a few other states in order to facilitate debugging and keep the hardware safe. This is accomplished by having the balance controller also manage a simple internal state machine. Figure 4.11 depicts the different states the controller can be in.

As can be seen from figure 4.11, the balance controller's internal state machine runs in parallel with a higher level state machine operating within the non real-time Python

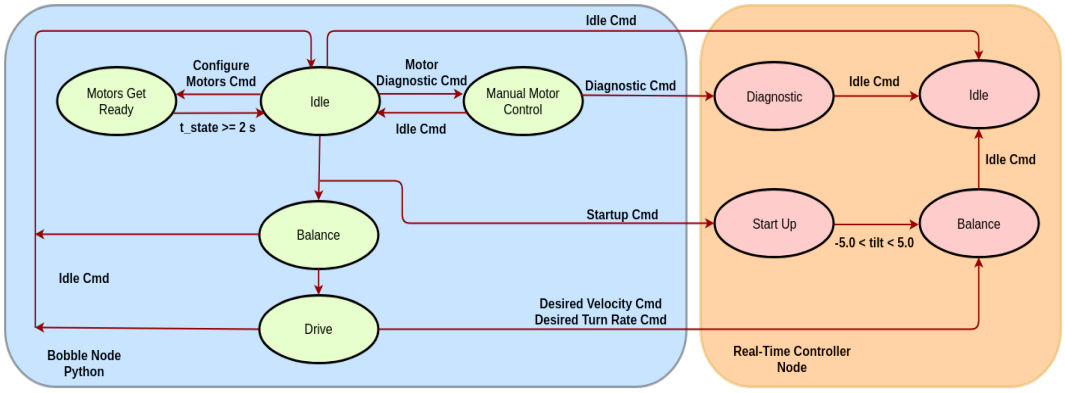


Figure 4.11: *Controller States*

executive. The non real-time state machine is implemented following a generalized state pattern in object oriented Python Gamma et al. (1995). The balance controller's internal state machine is implemented as a simple C++ switch statement that runs during the controller's update function. The higher level state machine communicates with the balance controller state machine by using ROS's publish/subscribe pattern. The best practices for real-time programming detailed in section 4.1.3 are employed in order to ensure that this communication channel is real-time capable. These state machines work together to manage the balance controller's internal state to keep the hardware and operator safe while promoting ease of use and maximizing flexibility. Table 4.4 lists a summary of the most crucial states and provides a short description for each.

Table 4.4: *Controller States*

State	**Description**
Idle	Motors powered on but inactive
Motors Get Ready	Uses Python API to send motor configs
Diagnostic	Enable manual controlling of motors
Start Up	Start balance controller and activate it when safe
Balance	Execute balance control law desired states = 0
Drive	Execute balance control law using desired states

Defining these controller states has three primary advantages. The first is the ability to gracefully handle some safety concerns. Defining an idle state that the robot can be commanded into from any other state allows for the operator to effectively shut the motors down at any time. This can be used if the system begins to operate erratically and jeopardizes the safety of the robot or its environment. Likewise, defining a transitionary start-up state allows the balance controller to be primed but to remain within an inactive state until the robot reaches a tilt angle in which it can be safely balanced. Secondly, the balance control state machine affords greater flexibility when extending the robot's basic capabilities. As an example, re-tuning the controller on-the-fly would not be possible without a mechanism like this one. Additionally, it is likely that these states will be reused during the implementation of autonomous navigation algorithms. These algorithms are traditionally more computationally expensive, and they would likely be implemented within the non real-time state machine. While these routines are running, however, the robot would still be expected to perform its more critical function of maintaining its balance and tracking desired state commands. The state machine provides an easy mechanism to develop,

test, and introduce these types of capabilities without posing extra risk. Finally, the introduction of a diagnostic state simplifies the robot's maintenance burden. The diagnostic state can be used to test out new hardware and software features in an easy and safe way. One common use of the diagnostic state is to check out motor, controller, and filter performance while the robot is placed on a test stand. For these reasons, and many more, the state machine implementation makes Bobble-Bot a safer, more robust, and easily extended platform for robotics and controls development.

The last important aspect of the balance controller implementation involves tuning several critical controller performance gains. Table 4.5 lists a few of the most important parameters. All PID controllers must be tuned for the system that they are controlling, and Bobble-Bot is no exception. Rise time, settling time, amount of over/under-shoot, and steady state error are some of the most common control performance metrics used to evaluate potential control gains Tay et al. (1997). Each PID controller used within Bobble-Bot's balance controller has different requirements on each metric. Robust tilt control is clearly important in this application, but it needs to be carefully weighed against the robot's overall maneuverability. This trade-off is accomplished by careful selection of the filter and control gains used by the balance controller. Of course, Bobble-Bot's inherent unstable dynamics further complicates this tuning process. The next two sections are dedicated to determining suitable control gains in a way that minimizes the risk to damaging the system.

Table 4.5: *Select controller configuration parameters*

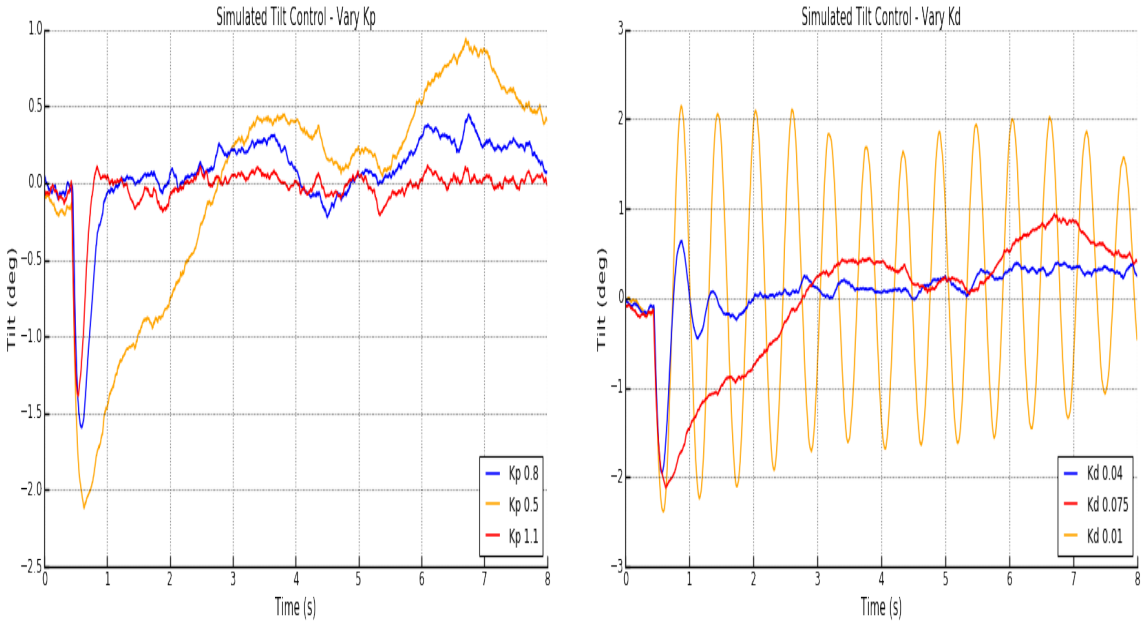
Parameter Name	**Description**
Madgwick Gain	State estimation filter
Tilt LPF	Tilt LPF gain
Tilt-dot LPF	Tilt-dot LPF gain
Turn rate LPF	Turn rate LPF gain
Wheel Velocity LPF	L/R wheel velocity LPF gain
Velocity Control Kp	Proportional gain, velocity
Velocity Control Ki	Integral gain, velocity
Tilt Control Kp	Proportional gain, tilt
Tilt Control Kd	Derivative gain, tilt
Turning Control Kp	Proportional gain, turning
Turning Control Ki	Integral gain, turning
Turning Control Kd	Derivative gain, turning

4.4 Control Tuning

The proceeding sections in this chapter have detailed an approach in which implementation and verification of the hardware systems has been done independently from the software implementation of the control law. This was accomplished by way of a modular software and hardware architecture that carefully considered the challenges posed when integrating a prototype controller together with hardware devices on an unstable platform. At the end of the controller design in section 3.5, a simulation was proposed as a method of safely verifying the controller implementation entirely in software. This was proposed with the goal of minimizing the risk of damaging the

hardware when the entire system is tested together for the first time. This section covers the results from applying this approach with the Gazebo based Bobble-Bot simulator discussed in section 3.6. The balance controller implementation covered in the previous section is run alongside the simulator. Test data is collected from each run of the simulation. An analysis is then carried out with the goal of evaluating different control gain selections and their impact on tilt, velocity, and turning control performance. The state machine architecture and real-time monitoring software are used repeatedly throughout this process. These tools allow for controller gains to be modified and analyzed as the simulation is running. A summary of the end results from this exercise are provided in the sections that follow.

4.4.1 Simulated Tilt Control



(a) Tilt Response Kp Tuning *(b) Tilt Response Kd Tuning*

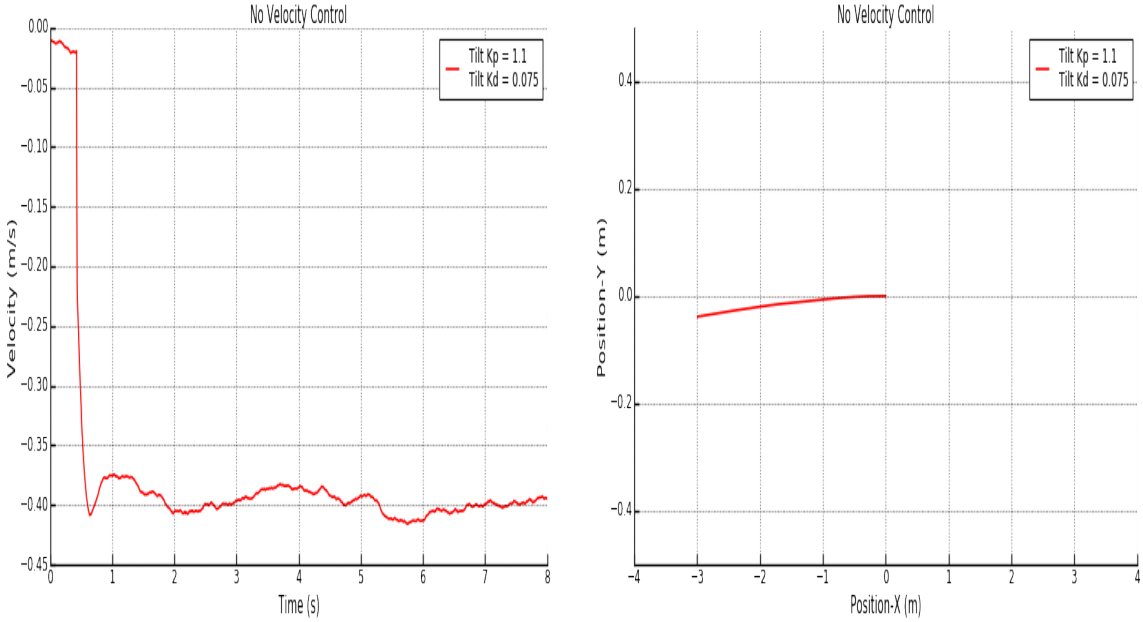
Figure 4.12: *Tilt Control Tuning*

Figure 4.12 summarizes the results from six different simulation runs. An impulse force in the -X direction is applied a half a second into each simulation run. The tilt controller takes over and does its job to reject the disturbance in order to keep the robot balanced. Figure 4.12 shows how the selection of different TiltControlKp and TiltControlKd values effects the performance. Smaller values of Kp lead to a much longer settling time. This is undesirable. Kp values that are too high are not likely achievable on the real robot, but the simulator predicts that they work well in the simulation environment. Large values of TiltControlKp result in large and rapid changes to the resulting torque command. This shows that the current motor model in the simulator could benefit from an increase in model fidelity. For the final tuning during real hardware tests, the safest approach is to start with smaller stable values and ramp up the Kp value until the system approaches instability.

The second half of the tests were dedicated to tuning TiltControlKd in the simulation. As expected, the simulator shows that increasing Kd has the advantage of reducing the settling time at the expense of additional overshoot. The end result of this analysis is a prediction that a TiltControlKp value of 1.0 and a TiltControlKd value of 0.05 will result in stable tilt control while also being achievable with the real motors. These values will serve as our starting tilt control gains for the first hardware tests.

Figure 4.13 shows the robot position and velocity response during one of the simulated tilt control tests. The velocity plot shows that the -X impulse force results in a -0.4 m/s velocity in the X direction. The velocity controller gains were set to zero for these tests, so this result is expected. The position plot shows the robot's trajectory in X, Y world co-ordinates when only the tilt controller is active. The

end result from this analysis is that the tilt controller is performing as expected. It is able to keep the robot upright, but it is not sufficient to bring the robot back to rest following an impulse force. To accomplish that goal, we must turn to tuning the velocity controller. The next section shows the results from that effort.



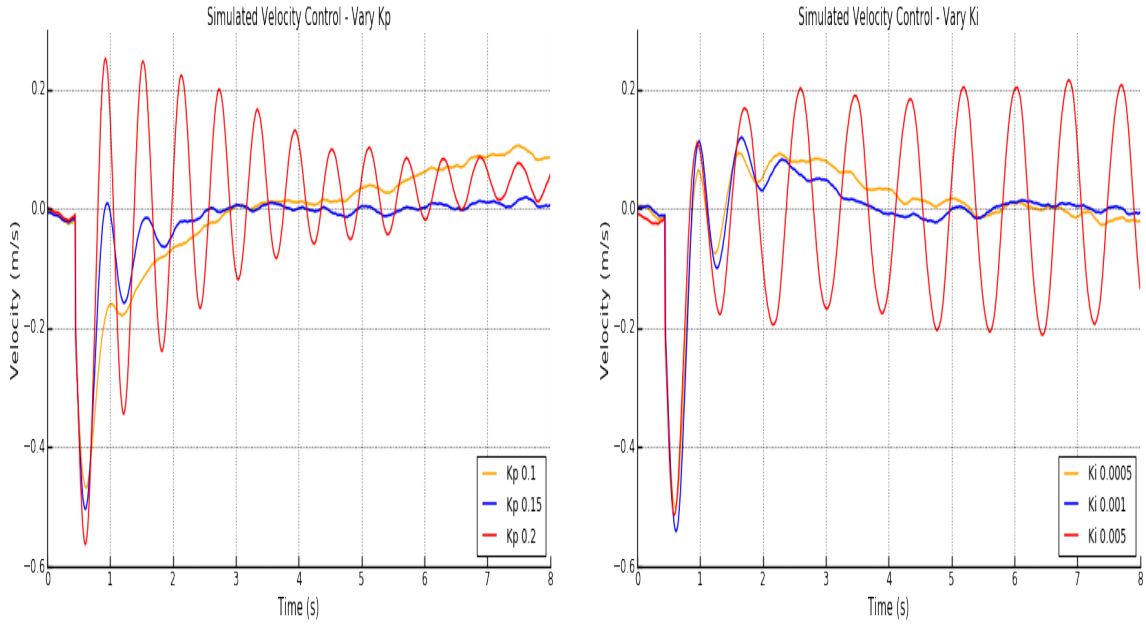
(a) Backward Velocity *(b) World Position X, Y*

Figure 4.13: *Tilt Control Without Velocity Control*

4.4.2 Simulated Velocity Control

Following the same testing procedure discussed in the previous section, figure 4.14 shows the results from tuning the velocity controller in the simulator. This time, we analyze the robot's velocity in response to the same -X impulse force. In all simulation runs we see the impulse force results in a velocity around -0.4 m/s in the -X direction. However, unlike the results from the previous section, this resulting velocity is controlled back to a resting state. This is the expected result as the job of the velocity controller during balance mode is to try to control the robot to stay at

rest. This is accomplished by setting the VelocityControlKp and VelocityControlKi values appropriately. Figure 4.14 shows the effect of selecting different values for these gains. The end result of this analysis is a prediction that a VelocityControlKp value of 0.15 and VelocityControlKi value of 0.001 are likely to be good starting values during hardware testing.

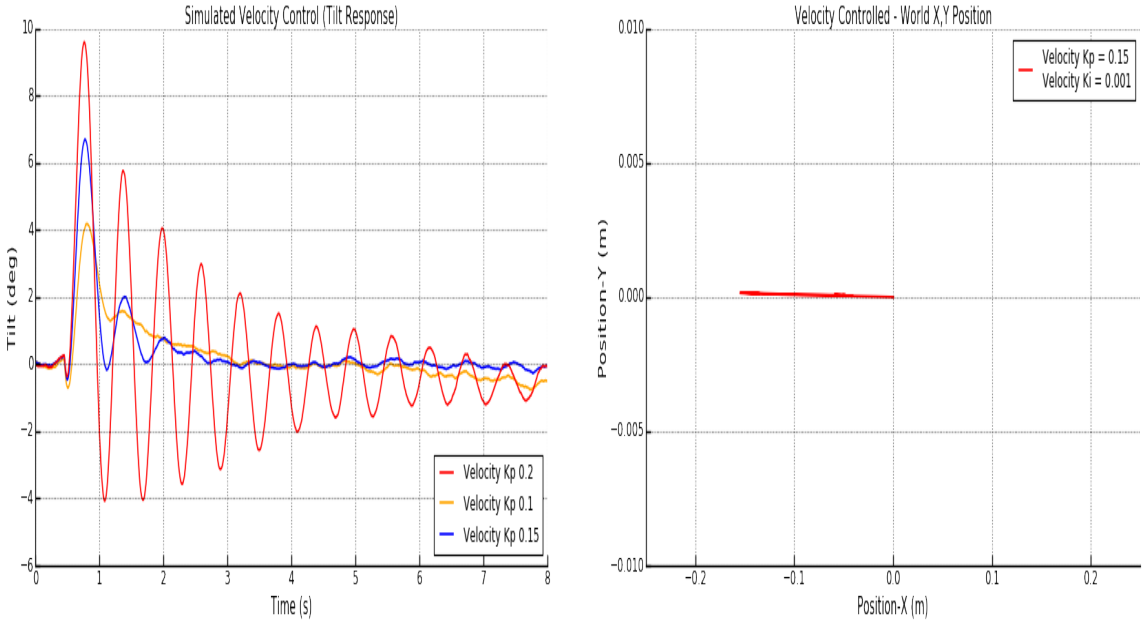


(a) *Velocity Response Kp Tuning* ***(b)*** *Velocity Response Ki Tuning*

Figure 4.14: *Velocity Control Tuning*

The tilt response plot in figure 4.15 shows how the velocity controller achieves its goal of keeping the robot at rest. In response to the impulse force, the controller modulates the desired tilt command sent into the tilt controller. This has the effect of initially tilting the robot forwards in response to the impulse in the -X direction. The dynamics of the TWIP system show that this will cause an acceleration that will fight against the -X impulse force and eventually bring the system back to rest. The controller's integral gain allows the robot to return close to its original position by

closing out the steady state error. This can be seen by examining the position plot on the right side of figure 4.15. The robot travels about 15 cm in the -X direction before returning back towards the origin.



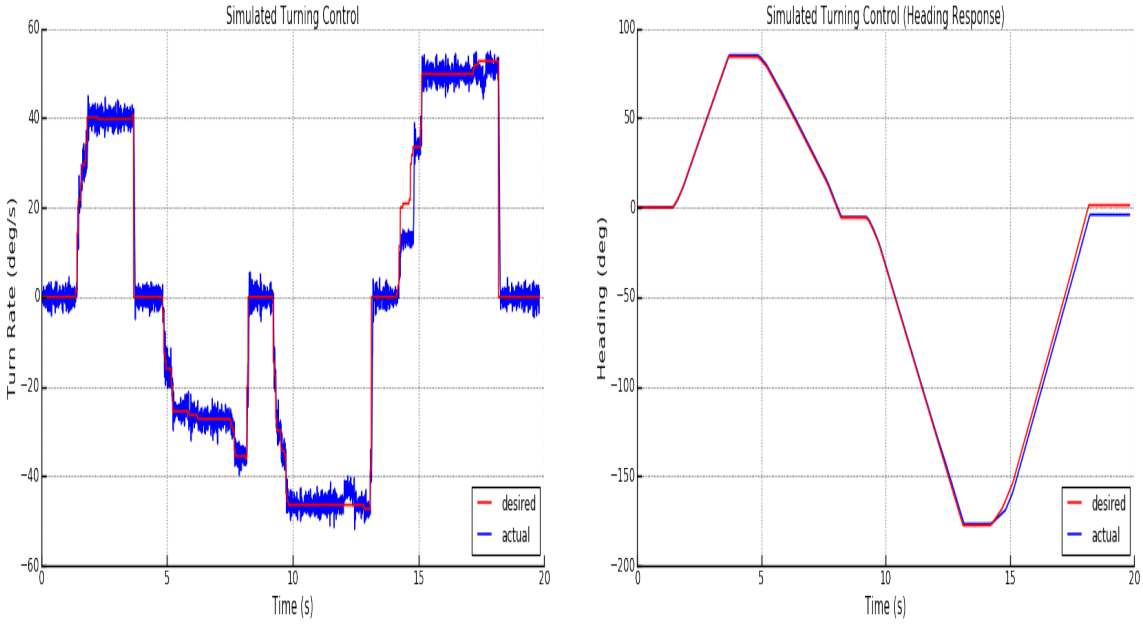
(a) Tilt Response *(b) World Position X, Y*

Figure 4.15: *Velocity Control*

This analysis has used simulation to verify the performance of the proposed velocity and tilt controller design given in section 3.5. The results show that this design works to keep Bobble-Bot balancing in place even when perturbed by a disturbance force. In much the same way, the controller is also capable of tracking a desired forward and backward reference velocity command. However, the tilt and velocity controllers are not sufficient to enable turning. The next section describes a simulation based approach to tuning the final controller used by Bobble-Bot, the turning controller.

4.4.3 Simulated Turning Control

The last step in the simulation based validation of the balance controller is verifying the turning controller depicted in the block diagram shown in figure 3.11. The turning controller is a PID controller acting on the turn rate feedback signal provided by the gyroscope. This controller outputs a positive and negative bias to the left and right motor torques respectively. These two torque biases result in a motor velocity differential that turns the robot left and right. The TurningControlKp, Ki, and Kd gains were tuned experimentally in simulation. Figure 4.16 shows the resulting performance after tuning all three of these gains.

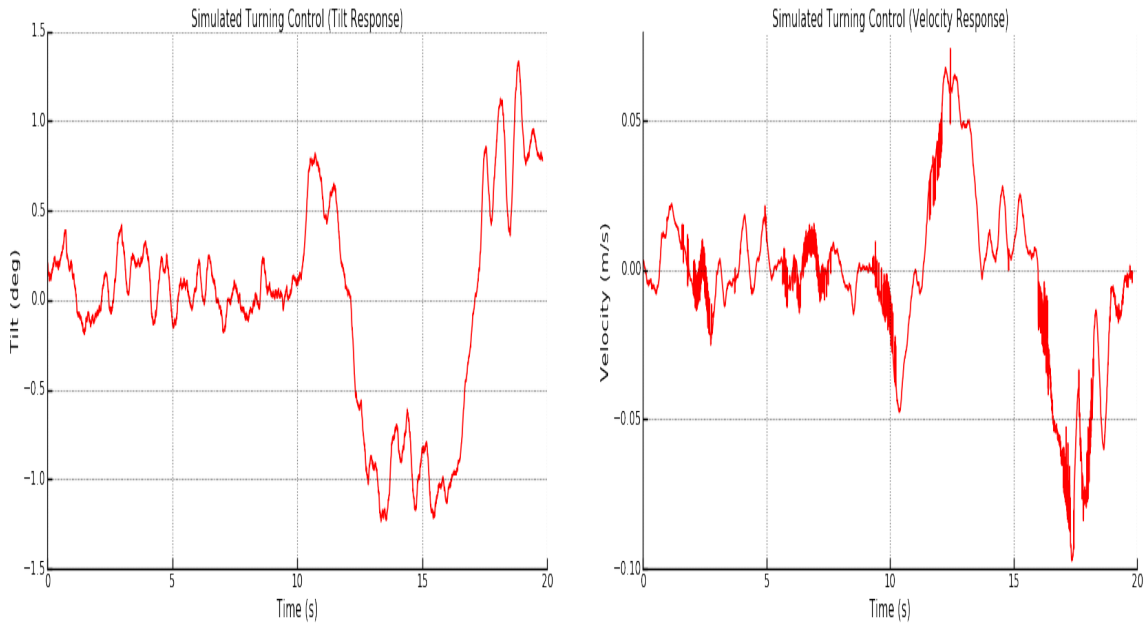


(a) Turn Rate Control *(b) Heading Control*

Figure 4.16: *Turning Controller Performance*

The simulation run used to generate the data shown in figures 4.16 and 4.17 is the result of commanding the robot to turn left and right a few times in succession. The turn rate command and the resulting desired heading are shown in red in the

left and right plots in figure 4.16. The resulting turn rate and heading response are shown in blue. For this simulation run, the robot's desired velocity command was continually set to zero. Despite this, figure 4.17 shows that the simulated robot did pick up a little bit of velocity and tilt during the turning manuevers. A little wobbling during a turn is expected. The tilt and velocity controllers remain active during the turn, and the plots show that they successfully do their job to keep the robot mostly still and balanced throughout the turn. The result of this analysis is a prediction that the following turning controller gains should serve as the selected values during initial hardware tests: TurningControlKp is 1.0, TurningControlKi is 0.01, TurningControlKd is 0.085.



(a) *Tilt Response* ***(b)*** *Velocity Response*

Figure 4.17: *Tilt/Velocity While Turning*

The turning control analysis described above concludes the initial verification of the balance controller in its entirety. The simulator has helped to predict suitable

starting values to use for the most important control gains during hardware testing. The simulator has also helped to give credibility to the balance control design given in section 3.5. The final step is to compile the balance controller on the RPi and run it alongside the rest of the hardware. The next section will summarize the results from a preliminary integrated hardware and software test.

4.4.4 Validation with Hardware Testing

This chapter has been dedicated to the implementation of the design provided in chapter two. The real test of the entire approach comes when finally bringing all of the different hardware and software components together into their final configuration. The first tests done on the real Bobbble-Bot exactly mimic the simulation based tests performed in the previous section. In this way, we can analyze the results from the hardware test and compare them with the simulator to make refinements to improve both the hardware and the simulator simultaneously. The end goal is a working Bobble-Bot whose performance is sufficiently approximated by the simulator in order to enable continued reliance on simulation as the initial proving ground for any future additions to Bobble-Bot's control and navigation capabilities.

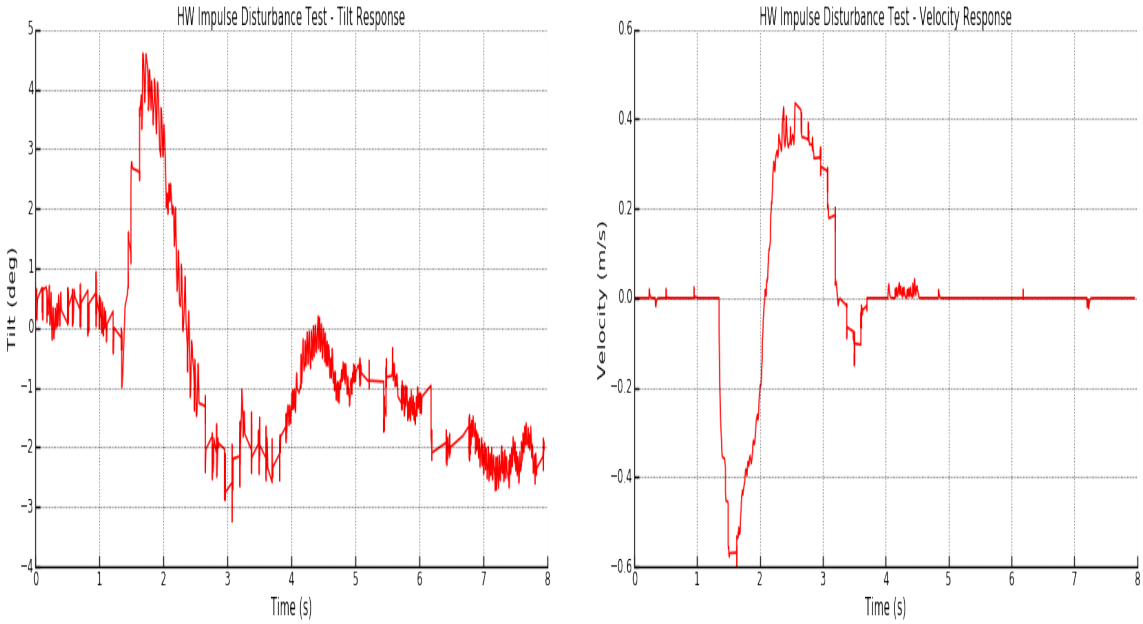
The first test done on the real Bobble-Bot mimics the -X impulse force test carried out initially in simulation. This test will verify the control gains predicted by the simulator. Those control gains are repeated here in table 4.6. Like in simulation, the state machine is used once again during this test to load these configuration values at run time. This allows for rapid tuning of the system in real time.

Table 4.6: *Control Gains for HW Test*

Control Gain	**Value**
Tilt Control Kp	1.0
Tilt Control Kd	0.05
Velocity Control Kp	0.15
Velocity Control Ki	0.001
Turning Control Kp	1.0
Turning Control Ki	0.01
Turning Control Kd	0.085

As in simulation, the Xbox controller described in section 3.3 is used to issue the necessary commands to prepare the robot for the impulse test. Unlike simulation, however, there is a unique hardware only test procedure that is followed prior to the impulse test that is used to verify the motors are operating as expected before attempting to enter the balance state. As described in section 4.3, the robot begins in the idle state. A controller button is then pressed to enter the Motors Get Ready state and load a motor configuration file onto each BLDC motor controller. Once that step is completed the robot automatically returns to the idle state. Bobble-Bot is then placed on a test stand, and each motor is manually commanded using the controller's left and right joysticks. The test operator visually verifies that the motors produce the expected torque response. Once they are satisfied, they issue a command to return the robot back to the idle state. Bobble-Bot is now ready to safely enter the start-up state. At first, the robot is assisted by holding the robot up-right. The operator then issues the balance command to enter the balance state. In response to this command, the real-time state machine modes the balance controller into the start-up state. The

balance controller remains in this state and keeps the motors inactive until it senses that it is in an assisted up-right position. At the time of this writing, Bobble-Bot has no ability to up-right itself within the start-up state, although that is a desirable feature for future work. Once the robot detects that it is in a safe initial condition, it activates the balance controller and begins sending torque commands to each motor. If all goes well, the operator can stop assisting the robot as it will begin to balance on its own. To initiate the -X impulse test, the test operator sends a command to ROS to begin logging data, and then gently pushes the robot in the -X direction. Figure 4.18 shows the results of carrying out this test procedure using the control gains given in table 4.6.



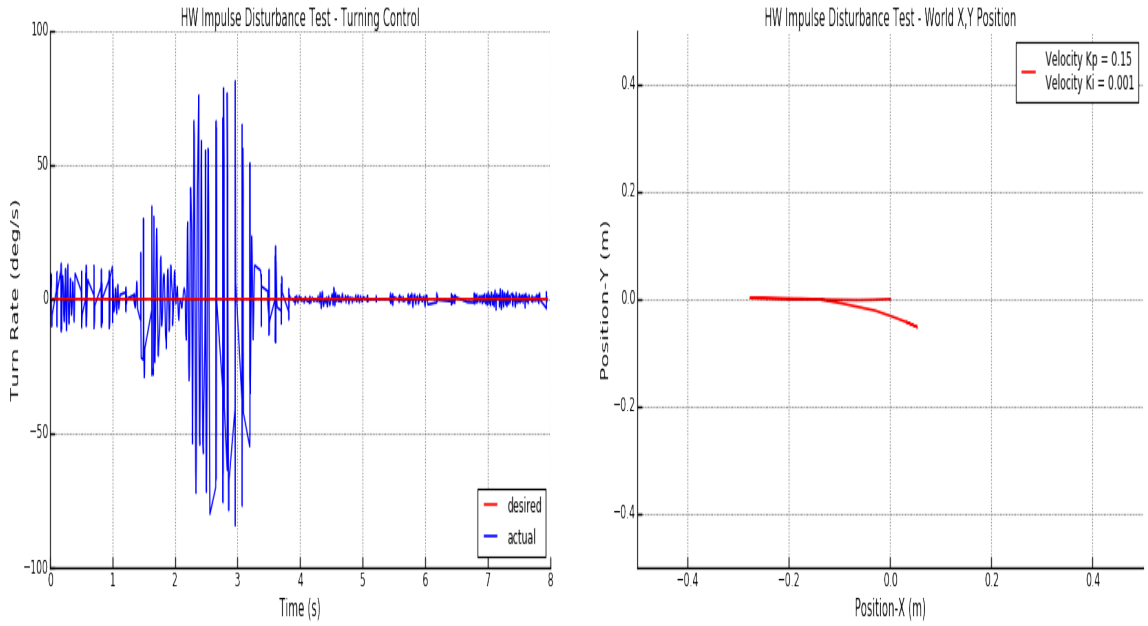
(a) *Tilt Response* ***(b)*** *Velocity Response*

Figure 4.18: *Tilt/Velocity During HW Impulse Test*

Figure 4.18 shows the tilt and velocity response from this test. At first glance, we see that the simulation has done a very good job at predicting viable control gains.

The tilt and velocity response are qualitatively very similar to what was observed in the previous section. Like in simulation, the -X impulse disturbance causes the robot to reach a maximum tilt angle of about 5 degrees and a maximum velocity of around -0.4 m/s. A more detailed comparison of the hardware versus the simulator will be given in the conclusion of this work. However, the results shown here are encouraging. Figure 4.19 shows the action of the turning controller during this test and also the resulting position of the robot in world co-ordinates. This shows that the turning controller is successfully doing its job despite the presence of considerably larger noise within the real gyroscope's turn rate measurement. This indicates that perhaps greater filtering of this signal could lead to better performance. The world position plot shows that the -X impulse force caused the real robot to travel about 10cm farther than what was predicted by the simulation, although the final steady state error magnitude in position is comparable. There are a variety of potential reasons for this difference including differences in the applied impulse force magnitude and surface friction. Despite these differences, the results still indicate that the simulator has served its purpose.

The second hardware test, and final verification result presented in this chapter, is a repeat of the turning test that was done in simulation in the previous section. BobbleBot is commanded to turn left and right in place. Figure 4.20 shows the desired and actual turn rate and heading. This result shows that the turning controller did not perform as well as the simulator predicted it would. The reason for this can be seen by inspecting the noise magnitude on the actual sensed turn rate in the left plot of figure 4.20. This plot shows that the controller is causing the turn rate to achieve the desired value on average. This results in a turn that the robot operator perceives as smooth and responsive. However, for precision driving, the controller is actually

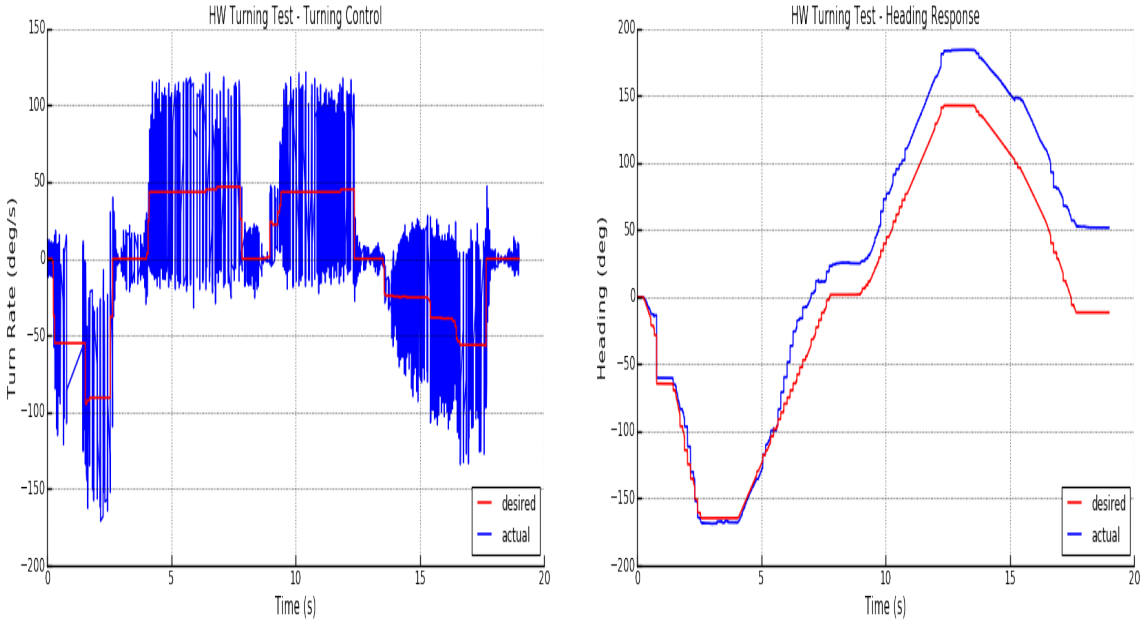


(a) Turn Rate Control *(b) Position Control*

Figure 4.19: *Turn Rate/Position During HW Impulse Test*

accumulating some steady state error when you look at the changes in the resulting heading signal. It is important to keep in mind that the current implementation of the turning controller is entirely based on turn rate. The controller does not have any sensor or make any estimate of what the absolute heading might be. In fact, the plot on the right side of figure 4.20 is generated entirely during post processing of the turn rate command and response data that was logged during the test. The plot on the right is the result of integrating the desired and actual signals shown in the plot on the left. There's a lot of noise in the turn rate signal, so this integration is likely building up a certain amount of error. The result of this analysis is that investing some time in additional filtering on the gyroscope turn rate signal would likely result in an improved system capable of more precisely tracking turn rate. Adding absolute position and heading sensors would also help solve this problem. In the end, this

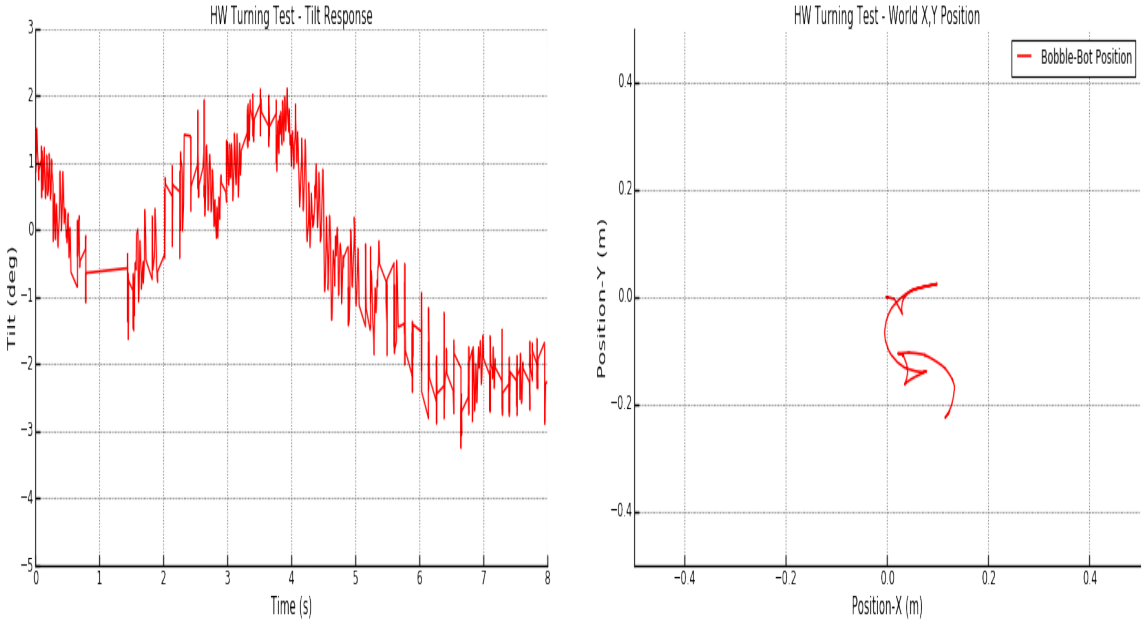
result was deemed acceptable because Bobble-Bot does not have any need to control absolute heading at this time.



(a) *Turn Rate Control* ***(b)*** *Heading Control*

Figure 4.20: *Turn Rate/Heading Control During HW Turning Test*

Figure 4.21 shows the tilt and world position during the turns. The simulation predicted that the tilt controller would continue to control the robot's tilt angle to within +- 1.5 degrees during the turn. The hardware results show the tilt remained between +- 2.5 degrees during all the turning. The robot was commanded to turn in place so we do not expect to see much movement in the X,Y plane during the turn. The right hand plot in figure 4.21 shows that the robot remained within an approximately 20 x 20 cm grid during all of the turning. This is an indicator that both the tilt and velocity controllers were doing their job of restricting the robot's motion to a mostly pure turn about the Z axis.



(a) *Tilt While Turning* ***(b)*** *Position While Turning*

Figure 4.21: *Tilt and Position During HW Turning Test*

The hardware turning test concludes the initial validation of the Bobble-Bot balance controller. Taken all together, the simulation and hardware verification sections of this chapter show that the balance controller is working as expected in both hardware and simulation. These results help to establish a great deal of confidence in the correctness of the balance control implementation that was provided in this chapter. The final chapter will present a more direct comparison of simulation and hardware before giving some concluding remarks and listing plans for future work.

5

CONCLUSION

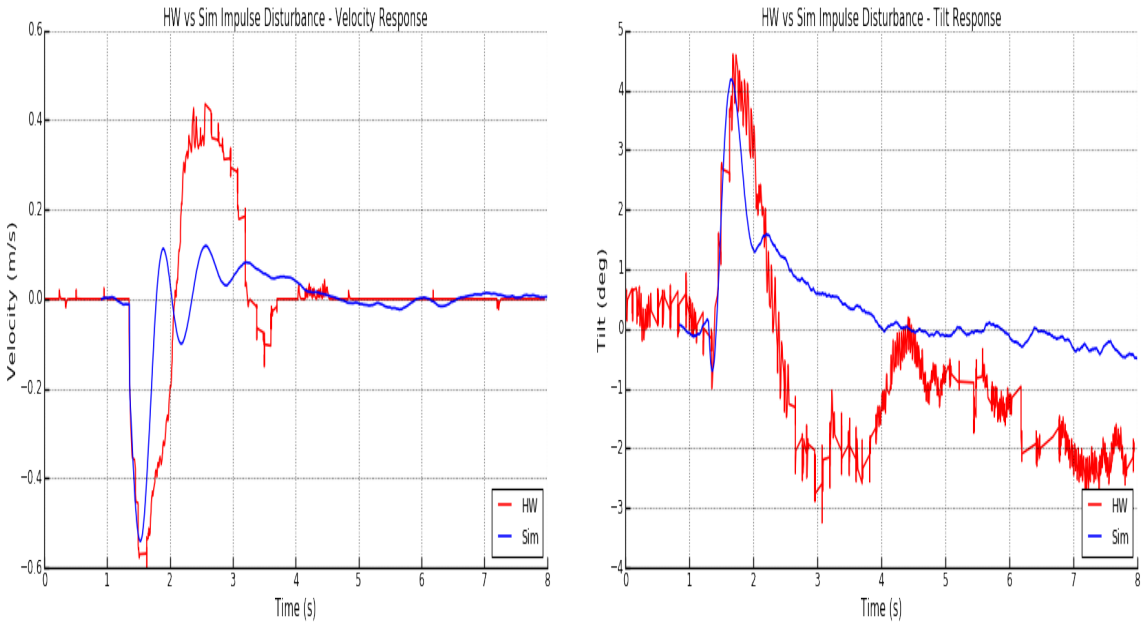
This paper has focused on the theory, design, and implementation of a high bandwidth, real-time, feedback controller for an unstable system. The sections progressed linearly from theory to design to implementation and finally validation. The actual development process was a constant cyclic iteration between these steps as the design was continually refined based on results from simulation and hardware tests. The design presented in chapter three captures the list of final assumptions, requirements, component selection, and software modules that were converged upon as the problems became better understood and ultimately solved. Real-time control played a critical role in achieving a satisfactory level of control performance that enabled a robust balance controller capable of maintaining balance while driving. As such, it was the main focus of this work. Chapter four shows how achieving real-time performance with the Linux OS is a delicate balancing act between software architecture, communication and message passing infrastructure, hardware component selection, device driver implementation, and of course a real-time capable OS kernel with a prioritized scheduler and memory locking mechanisms.

Perhaps the second most important element to achieving the final result was the development of a simulator that could be used to directly test the actual source code implementation of the balance controller. As shown in chapter four, this simulation was instrumental in determining suitable control gains that keep the system stable throughout the various driving conditions it may encounter. Without the simulation, it would have been challenging to verify the control design and gain selection without damaging hardware and slowing down the pressing development schedule. The next section presents a final comparison between simulation and reality. The results further

demonstrate how reliable the simulator has become in its ability to track the real performance of Bobble-Bot. The simulation was worth the effort it took to develop, and it will continue to be used to develop and test future Bobble-Bot capabilities.

5.1 Hardware vs Simulation

Figure 5.1 shows the real system's impulse response as compared with the the result seen in the simulation. The comparison matches quite well considering that the magnitude of the applied impulse was not measured during the hardware test.



(a) *Velocity Response* ***(b)*** *Tilt Response*

Figure 5.1: *Simulation vs Hardware Impulse Response*

The hardware versus simulation impulse response test above was done while Bobble-Bot was placed in balance mode. The next analysis that was performed was a world position comparison between hardware and simulation when the robot was placed in drive mode. To do this test, a 1 m x 1 m square was marked in tape on a concrete

floor. Bobble-Bot was then manually driven slowly and carefully to attempt to follow the perimeter of the square path as closely as possible. During this driving test, the commands sent from the Xbox controller to the robot were logged to a data file. The robot's response during the driving tests were also logged to a data file. This response data was then analyzed and the robot's world position was plotted. Once a satisfactory result was obtained, the commands from the hardware drive test were played back into the simulation. The simulated robot's response to these commands was then logged to a data file. Figure 5.2 shows the resulting comparison from performing this hardware and simulator test.

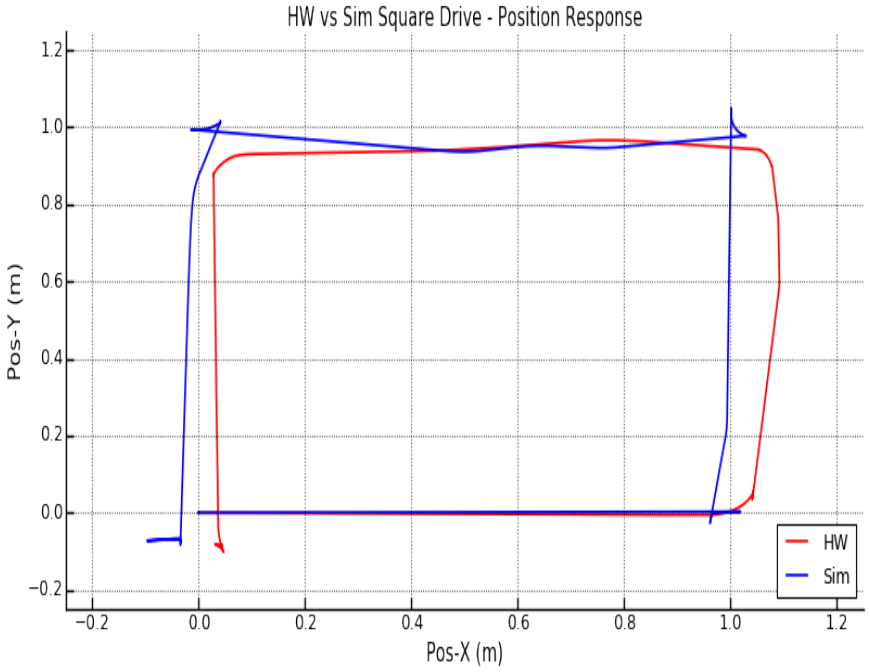


Figure 5.2: *Simulation vs Hardware Square Drive Comparison*

The simulation has an ideal model of the friction force between the wheels and the floor. Furthermore, the simulation's motor model has room for improvement. The simulator also applies Gaussian noise to its simulated IMU measurements. For these reasons, the controller performs a bit differently in simulation than it does in reality

despite being given the same commands. Despite these differences, the comparison shows how well the simulation approximates reality. These tests show that both the simulation and hardware is performing as designed. This hardware versus simulation validation effort has established confidence in the simulator as a tool for future Bobble-Bot controls development. The plan for future work is given in the next and final section of this work.

5.2 Future Work

Without carefully considering and addressing all the challenges associated with real-time programming, Bobble-Bot's balance controller would never have worked within the Linux OS. Running the balance controller in Linux and on the RPi allows the robot to fully leverage the power and capability of the ROS infrastructure. The author is forever grateful to the collection of developers, authors, and maintainers that contribute to ROS. Please consult the references provided in the bibliography for more information on this outstanding community. Follow on work will continue with the primary goal of contributing Bobble-Bot's simulation and source code back to the community it benefited from. The following list provides some additional items that are also planned as future work.

- Migrate Bobble-Bot to ROS2 and use it as a test bed to explore the advancements in real-time control support promised by ROS2.
- Further hardware versus simulation analysis.
- Use ROS libraries to integrate a SLAM capability.
- Release the simulator to the ROS and Gazebo open source community.
- Increase simulation fidelity with the addition of a more detailed motor model

Control of unstable systems is a challenging problem that typically leads to higher cost robotic systems. The ever growing need for high bandwidth, time-critical, control loops will continue to drive innovation in real-time controls. Options that can bring the cost down will lead to new innovations in robotics as it will reduce the cost of experimentation and prototype development. It is the author's sincere hope that the example of Bobble-Bot may help others trying to solve similar technical challenges on a limited budget. If nothing else, it was a fun problem with many ups and downs along the way.

BIBLIOGRAPHY

ADXL-345. Analog Devices ADXL345 Data Sheet, 2015. URL https://www.analog.com/media/en/technical-documentation/data-sheets/adxl345.pdf.

Wei An and Yangmin Li. Simulation and control of a two-wheeled self-balancing robot, 12 2013.

AS-5047D. Datasheet for BobbleBot's wheel position sensor, 2016. URL http://ams.com/eng/Products/Magnetic-Position-Sensors/Angle-Position-On-Axis/AS5047D.

Ahmad Azar, Hossam Hassan Ammar, Mohamed Hesham Barakat, Mahmood Abdallah Saleh, and Mohamed Abdallah Abdelwahed. Self-balancing robot modeling and control using two degree of freedom pid controller, 01 2019.

Mete Balci. Latency of Raspberry Pi 3 on Standard and RT Linux 4.9 Kernel, 2017. URL https://metebalci.com/blog/latency-of-raspberry-pi-3-on-standard-and-real-time-linux-4.9-kernel/.

Olfa Boubaker. The inverted pendulum: A fundamental benchmark in control theory and robotics. *International Conference on Education and e-Learning Innovations*, 2012. doi: 10.1109/iceeli.2012.6360606.

Olfa Boubaker. The inverted pendulum: history and survey of open and current problems in control theory and robotics, 12 2017.

Lukas Bulwahn. Real-Time Linux Continues Its Way to Mainline Development and Beyond, 2018. URL https://www.linuxfoundation.org/blog/2018/09/real-time-linux-continues-its-way-to-mainline-development-and-beyond.

John M. Carson, Behcet Acikmese, Lars Blackmore, and Aron A. Wolf. Capabilities of convex Powered-Descent Guidance algorithms for pinpoint and precision landing. *2011 Aerospace Conference*, 2011. doi: 10.1109/aero.2011.5747244.

Arnoldo Castro. Modeling and Dynamic Analysis of a Two-Wheeled Inverted-Pendulum. Master's thesis, Georgia Institute of Technology, 2012.

Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Fernández Perdomo. ros_control: A generic and simple control framework for ros. *The Journal of Open Source Software*, 2017. doi: 10.21105/joss.00456. URL `http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf`.

Barbara Faccini. Four Minutes, 23 Seconds : Flight AF447, 2013. URL `http://understandingaf447.com/extras/18-4_minutes__23_seconds_EN.pdf`.

Davide Faconti. PlotJuggler Timeseries Visualization in ROS., 2018. URL `https://github.com/facontidavide/PlotJuggler`.

RPi Foundation. Respberry Pi Model 3B Data Sheet, 2016. URL `https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdfD`.

Tobin Fricke. State Space Intro Talk, 2012. URL `https://github.com/tobin/statespace-intro-talk`.

Adrian Gambier. Real-time control systems: a tutorial, 08 2004.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

Da-Wei Gu, Petko Petkov, and M.M. Konstantinov. Robust control of self-balancing two-wheeled robot, 01 2013.

Alexander Gutierrez. Installing ROS Kinetic on the Raspberry Pi, 2018. URL `http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Kinetic%20on%20the%20Raspberry%20Pi`.

Oliver Hartkopp. SocketCAN Library, 2015. URL `https://github.com/linux-can`.

Juang Hau-Shiue and Kai-Yew Lum. Design and control of a two-wheel self-balancing robot using the arduino microcontroller board, 06 2013.

ITG-3200. InvenSense ITG-3200 Data Sheet, 2011. URL `https://www.invensense.com/wp-content/uploads/2015/02/ITG-3200-Datasheet.pdf`.

A. S. Katariya. Optimal State-Feedback and Output-Feedback Controllers for the Wheeled Inverted Pendulum System. Master's thesis, Georgia Institute of Technology, 2010.

Jackie Kay. Proposal for Implementation of Real-time Systems in ROS 2, 2015. URL `https://design.ros2.org/articles/realtime_proposal.html`.

Jackie Kay. Introduction to Real-Time Systems, 2018. URL `https://design.ros2.org/articles/realtime_background.html`.

Jackie Kay and Adolfo Rodriguez Tsouroukdissian. Real-Time Control in ROS and ROS 2.0. Open Source Robotics Foundation, 2015.

Sangtae Kim and SangJoo Kwon. Dynamic modeling of a two-wheeled inverted pendulum balancing mobile robot, 2015.

Phil Koopman. A Case Study of Toyota Unintended Acceleration and Software Safety, 2014. URL https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf.

Linux-Foundation. SocketCAN Documentation, 2012. URL https://www.kernel.org/doc/Documentation/networking/can.txt.

Linux-Foundation. How To Build a Simple RT Application, 2017. URL https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/application_base.

K. Lundberg and T. Barton. History of Inverted-Pendulum Systems. volume 42, pages 131–135. IFAC Proceedings Volumes, 2010.

Sebastian O.H. Madgwick. An efficient orientation filter for inertial and inertial/-magnetic sensor arrays, 2010. URL http://x-io.co.uk/res/doc/madgwick_internal_report.pdf.

Micrium. What is an RTOS, 2015. URL https://www.micrium.com/rtos/what-is-an-rtos/.

Microchip. Microchip MCP-2515 Data Sheet, 2012. URL http://ww1.microchip.com/downloads/en/devicedoc/21801e.pdf.

MikroBus. MikroBus Click Shield Data Sheet, 2014. URL https://www.mikroe.com/mikromedia-3-mikrobus-shield.

Mike Moore. GenericFilter, 2017. URL https://github.com/mike-moore/filter_tools.

Dave Mosher. SpaceX Reusable Rocket Costs and Profits, 2017. URL https://www.businessinsider.com/spacex-reusable-rocket-launch-costs-profits-2017-6.

Marco Di Natale. An Introduction to Real-Time Operating Systems and Schedulability Analysis, 2014. URL https://inst.eecs.berkeley.edu/~ee249/fa07/RTOS_Sched.pdf.

Huseyin Oktay Erkol. Optimal pid controller design for two wheeled inverted pendulum. *IEEE Access*, PP:1–1, 11 2018. doi: 10.1109/ACCESS.2018.2883504.

OSRF. Maintainers of ROS, 2018. URL https://github.com/osrf.

Manuel Perez-Polo, Manuel Perez Molina, Francisco gil chica, and José Angel Berná Galiano. Stability and chaotic behavior of a pid controlled inverted pendulum subjected to harmonic base excitations by using the normal form theory. *Applied Mathematics and Computation*, 232, 04 2014. doi: 10.1016/j.amc.

Derry Pratama, Eko Binugroho, and Fernando Ardilla. Movement control of two wheels balancing robot using cascaded pid controller, 09 2015.

Micho Radovnikovich. Teeterbot : A self-balancing robot simulation model for ROS/Gazebo, 2017. URL https://github.com/robustify/teeterbot.

Desna Riattama, Eko Binugroho, R.s Dewanto, and Dadet Pramadihanto. Pens-wheel (one-wheeled self balancing vehicle) balancing control using pid controller, 09 2016.

RoboSavvy. RoboSavvy Self-balancing robotic platform, 2017. URL http://wiki.ros.org/Robots/RoboSavvy-Balance.

Frank Rowand. Using and Understanding the Real-Time Cyclictest Benchmark, 2013. URL https://events.static.linuxfound.org/sites/events/files/slides/cyclictest.pdf.

RPi-Foundation. Bobble-Bot's RT Kernel, 2018. URL https://github.com/raspberrypi/linux/tree/rpi-4.14.y-rt.

Thomas Bewley Saam Ostovari, Nick Morozovsky. The Dynamics of a Mobile Inverted Pendulum (MIP), 2013. URL http://renaissance.ucsd.edu/courses/mae143c/MIPdynamics.pdf.

Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles*, pages 621–635. 01 2018. ISBN 978-3-319-67360-8. doi: 10.1007/978-3-319-67361-5_40.

Dan Sheadel. MiniPID, 2015. URL https://github.com/tekdemo/MiniPID.

SpaceX. SpaceX Falcon Heavy Side Boosters Landing Simultaneously at Kennedy Space Center. URL https://www.youtube.com/watch?v=u0-pfzKbh2k.

Tacon. Tacon 96M608 Data Sheet, 2012. URL https://www.hobbypartz.com/96m608-bigfoot160-5335-245kv.html.

Bryant Tan and Tim Wheeler. Optimal Control for Landing Rockets, 2014. URL http://timallanwheeler.com/aboutme/writeups/TAN_WHEELER_RocketLanding.pdf.

Teng-Tiow Tay, Iven Mareels, and John B. Moore. *High Performance Control (Systems & Control: Foundations & Applications)*. Birkhäuser, 1997. ISBN 9780817640040. URL https://www.researchgate.net/publication/235683246_High_Performance_Control.

Brian Thorne. Python-CAN Library, 2018. URL https://github.com/hardbyte/python-can.

Turnigy. Turnigy 5000 mAh 14.8 V Li-Po Data Sheet, 2014. URL https://hobbyking.com/en_us/turnigy-5000mah-4s1p-14-8v-20c-hardcase-pack-1.html.

Yorihisa Yamamoto. NSTway-GS Model Based Design, 2018. URL http://www.pages.drexel.edu/~dml46/Tutorials/BalancingBot/files/NXTway-GS%20Model-Based_Design.pdf.

GLOSSARY

API Application Programming Interface. 35, 37, 65, 70, 77, 78, 81, 109

ARM Advanced RISC Machine. 57, 109

BLDC Brushless DC Electric Motor. 24–26, 37, 40, 54, 69, 109

CAD Computer Aided Design. 30, 109

CAN Controller Area Network. viii, x, 25, 26, 41, 68–73, 109

CG Center of Gravity. x, 4, 5, 28, 29, 51–54, 109

CLI Command line interface. 109

CRC Cyclic Redundancy Check. 109

GUI Graphical user interface. 10, 109

I/O Input/Output. 10, 61, 62, 64, 65, 68–70, 109

I2C Inter-Integrated Circuit. 26, 75, 109

IDM Input Device Manager. 32, 109

IMU Inertial measurment unit. viii, 28, 39, 41, 51, 68, 75, 76, 100, 109

LiPo Lithium Polymer battery. 27, 109

LPF Low Pass Filter. 78, 83, 109

MBD Model Based Design. 11, 109

OS Operating System. 56, 60, 63, 98, 101, 109

OSRF Open-Source Robotics Foundation. 33, 109

PCB Printed Circuit Board. 109

PDU Power Distribution Unit. 24, 26, 27, 109

PID Proportional-Integral-Derivative. 7, 38, 42, 54, 109

POSIX Portable Operating System Interface. 8, 62, 63, 65, 109

PREEMPT_RT Linux Preemptive Real-Time kernel patch set. 10, 13, 35, 56, 62, 109

RAM Random Access Memory. 59, 63, 65, 66, 109

RISC Reduced Instruction Set Computer. 57, 109

ROS Robot Operating System. v, 1, 7, 11–14, 24, 26, 33, 35–39, 43, 45, 48–50, 56, 64, 65, 67, 76–78, 80, 93, 101, 109

ROS2 Robot Operating System v 2.0. v, 12, 109

RPi Raspberry Pi. Low-cost hobbyist embedded Linux computer.. 25, 26, 35, 39–41, 43, 54–59, 67, 69, 70, 72, 91, 101, 109

RPi Raspberry Pi. 109

RTOS Real-Time Operating System. 10, 12, 24, 109

SLAM Simultaneous Localization and Mapping. 49, 101, 109

SoC System on a chip. 26, 109

SSH Secure Shell. 109

TWIP Two Wheeled Inverted Pendulum. 1, 3–8, 11, 12, 28, 87, 109

URDF Universal Robot Description Format. 39, 109

USB Universal Serial Bus. 25, 109

YAML YAML Ain't Markup Language. 38, 109

6

APPENDIX

6.1 Bobble-Bot Chassis Mass Properties

This next snippet describes the mass properties used for Bobble-Bot's chassis in the simulator.

Listing 6.1: *BobbleBot Mass Properties from URDF*

```
<link name="bobble_chassis_link">
  <inertial>
    <origin xyz="0.0 0.0 0.180" rpy="0.0 0.0 0.0"/>
    <mass value="2.043"/>
    <inertia ixx="0.03" ixy="0.0" ixz="0.0" iyy="0.03" iyz="0.0
       " izz="0.03"/>
  </inertial>
  <collision>
    <origin xyz="0.0 0.0 0.130" rpy="0.0 0.0 0.0"/>
  <geometry>
    <box size="0.1 0.11 0.185" />
  </geometry>
    <contact_coefficients kd="1.0" kp="1000.0" mu="0"/>
  </collision>
</link>
```

6.2 Bobble-Bot Wheels Mass Properties

This next snippet describes the mass properties used for Bobble-Bot's wheels in the simulator.

Listing 6.2: *BobbleBot Mass Properties from URDF*

```
<link name="left_wheel_link">
  <inertial>
    <mass value="0.084"/>
    <inertia ixx="0.00035" ixy="0.0" ixz="0.0" iyy="0.00035"
       iyz="0.0" izz="0.00035"/>
  </inertial>
  <collision>
    <origin xyz="0.0 0.0 0.0" rpy="0.0 1.5707 1.5707"/>
    <geometry>
      <cylinder radius="0.05" length="0.05" />
    </geometry>
    <contact_coefficients kd="1.0" kp="1000.0" mu="0"/>
  </collision>
</link>
```

6.3 Python Double Pendulum

The code below provides an example of simulating a double pendulum using NumPy, SciPy, and Matplotlib. It is a good starting point for simple dynamics models from state space equations.

Listing 6.3: *Python Double Pendulum Dynamics Example*

```
# Double pendulum formula translated from the C code at
# http://www.physics.usyd.edu.au/~wheat/dpend_html/solve_dpend.c
from numpy import sin, cos
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as integrate
import matplotlib.animation as animation

  def derivs(state, t):
    dydx = np.zeros_like(state)
    dydx[0] = state[1]
    del_ = state[2] - state[0]
    den1 = (M1 + M2)*L1 - M2*L1*cos(del_)*cos(del_)
    dydx[1] = (M2*L1*state[1]*state[1]*sin(del_)*cos(del_) +
    M2*G*sin(state[2])*cos(del_) +
    M2*L2*state[3]*state[3]*sin(del_) -
    (M1 + M2)*G*sin(state[0]))/den1
    dydx[2] = state[3]
    den2 = (L2/L1)*den1
    dydx[3] = (-M2*L2*state[3]*state[3]*sin(del_)*cos(del_) +
    (M1 + M2)*G*sin(state[0])*cos(del_) -
```

```
22      (M1 + M2)*L1*state[1]*state[1]*sin(del_) -
23      (M1 + M2)*G*sin(state[2]))/den2
24    return dydx
25
26    def init():
27      line.set_data([], [])
28      time_text.set_text('')
29      return line, time_text
30
31    def animate(i):
32      thisx = [0, x1[i], x2[i]]
33      thisy = [0, y1[i], y2[i]]
34      line.set_data(thisx, thisy)
35      time_text.set_text(time_template % (i*dt))
36      return line, time_text
37
38 G = 9.8  # acceleration due to gravity, in m/s^2
39 L1 = 1.0  # length of pendulum 1 in m
40 L2 = 1.0  # length of pendulum 2 in m
41 M1 = 1.0  # mass of pendulum 1 in kg
42 M2 = 1.0  # mass of pendulum 2 in kg
43 # create a time array from 0..100 sampled at 0.05 second steps
44 dt = 0.05
45 t = np.arange(0.0, 20, dt)
46 # th1 and th2 are the initial angles (degrees)
47 # w10 and w20 are the initial angular velocities (degrees per
       second)
48 th1 = 120.0
```

```
w1 = 0.0
th2 = -10.0
w2 = 0.0
# initial state
state = np.radians([th1, w1, th2, w2])
# integrate your ODE using scipy.integrate.
y = integrate.odeint(derivs, state, t)
x1 = L1*sin(y[:, 0])
y1 = -L1*cos(y[:, 0])
x2 = L2*sin(y[:, 2]) + x1
y2 = -L2*cos(y[:, 2]) + y1
fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2),
    ylim=(-2, 2))
ax.grid()
line, = ax.plot([], [], 'o-', lw=2)
time_template = 'time = %.1fs'
time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)
ani = animation.FuncAnimation(fig, animate, np.arange(1, len(y)
    ),
interval=25, blit=True, init_func=init)
# ani.save('double_pendulum.mp4', fps=15)
plt.show()
```

6.4 RasPi CAN Driver Configuration Files

Listing 6.4: *Additions to /boot/config.txt*

```
1 dtparam=spi=on
2 dtoverlay=mcp2515-can0,oscillator=10000000,interrupt=6
```

Listing 6.5: *Additions to /etc/network/interfaces*

```
1 auto can0
2 iface can0 inet manual
3         pre-up /sbin/ip link set $IFACE type can bitrate 833333
              triple-sampling on
4         up /sbin/ifconfig $IFACE up
5         down /sbin/ifconfig $IFACE down
```