1-30-2012

The development of a robotic test bed with applications in Q-learning

Titus Appel

Titus James Appel

Candidate

Electrical and Computer Engineering

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Rafael Fierro , Chairperson

Ron Lumia

Meeko Oishi

The Development of a Robotic Test Bed with Applications in Q-Learning

by

Titus James Appel

B.S., Electrical Engineering, Kettering University, 2009

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Electrical Engineering

The University of New Mexico

Albuquerque, New Mexico

December, 2011

Dedication

To my wife, for supporting and encouraging me
to work hard and finish this endeavor.

Acknowledgments

This work was sponsored by the DOE University Research Program in Robotics (URPR), Grant #DE-FG52-04NA25590, awarded to the UNM Manufacturing Engineering Program.

I would like to thank God, for giving me the wisdom and strength to complete this project and my degree. I would also like to thank my wife for her support and encouragement throughout my studies. I am grateful for my advisor, Professor Rafael Fierro, for his support and advice during my studies as well as the MARHES lab for their help and encouragement. I truly appreciate Professor Ron Lumia for his help and critical suggestions on the learning part of the thesis along with Professor Meeko Oishi for agreeing to be on the thesis committee. I want to thank my family for encouraging me through my education. Lastly, I am indebted to Sandia National Laboratories, especially Organization 6623, for providing the financial support for my master's degree through NPSC, the National Physical Science Consortium.

The Development of a Robotic Test Bed with Applications in Q-Learning

by

Titus James Appel

ABSTRACT OF THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Electrical Engineering

The University of New Mexico

Albuquerque, New Mexico

December, 2011

The Development of a Robotic Test Bed with Applications in Q-Learning

by

Titus James Appel

B.S., Electrical Engineering, Kettering University, 2009

M.S., Electrical Engineering, University of New Mexico, 2011

Abstract

In this work, we show the design, development, and testing of an autonomous ground vehicle for experiments in learning and intelligent transportation research. We then implement the Q-Learning algorithm to teach the robot to navigate towards a light source. The vehicle platform is based on the Tamiya TXT-1 chassis which is outfitted with an onboard computer for processing high-level functions, a microcontroller for controlling the low-level tasks, and an array of sensors for collecting information about its surroundings.

The TXT-1 robot is a unique research testbed that encourages the use of a modular design, low-cost COTS hardware, and open-source software. The TXT-1 is designed using different modules or blocks that are separated based on functionality. The different functional blocks of the TXT-1 are the motors, power, low-level controller, high-level controller, and sensors. This modular design is important when considering upgrading or maintaining the robot.

The research platform uses an Apple Mac Mini as its on-board computer for handling high-level navigation tasks like processing sensor data and computing navigation trajectories. ROS, the robot operating system, is used on the computer as a development environment to easily implement algorithms to validate on the robot. A ROS driver was created so that the TXT-1 low-level functions can be sensed and commanded. The TXT-1 low-level controller is designed using an ARM7 processor development board with FreeRTOS, OpenOCD, and the CodeSourcery development tools. The RTOS is used to provide a stable, real-time platform that can be used for many future generations of TXT-1 robots. A communication protocol is created so that the high and low-level processors can communicate. A power distribution system is designed and built to deliver power to all of the systems efficiently and reliably while using a single battery type. Velocity controllers are developed and implemented on the low-level controller. These control the linear and angular velocities using the wheel encoders in a PID feedback loop. The angular velocity controller uses gain scheduling to overcome the system's nonlinearity. The controllers are then tested for adequate velocity response and tracking.

The robot is then tested by using the Q-Learning algorithm to teach the robot to navigate towards a light source. The Q-Learning algorithm is first described in detail, and then the problem is formulated and the algorithm is tested in the Stage simulation environment with ROS. The same ROS code is then used on the TXT-1 to implement the algorithm in hardware. Because of delays encountered in the system, the Q-Learning algorithm is modified to use the sensed action to update the Q-Table, which gives promising results. As a result of this research, a novel autonomous ground vehicle was built and the Q-Learning source finding problem was implemented.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter describes the motivation of creating a prototype for a testbed of modular robotic platforms and testing it in a learning application. It also provides a literature review of developed mobile robotic platforms and Q-Learning and robotics. Then the contributions of this thesis are explained and an overview of the chapters are given.

1.1 Motivation

In recent years, there have been great improvements in sensor technology and computing. Sensors have become smaller, more accurate, and inexpensive while breakthroughs in computing technologies have offered smaller, faster, and more efficient computers for a lower price tag.

Additionally, there have been great pushes to research intelligent vehicles. For example, DARPA launched the Grand Challenge in 2004 [1] and the Urban Challenge [2] in 2007, competitive races in which teams built autonomous cars capable of navigating on dirt roads and urban cities. The projects' goal was to spur autonomous

(a) (b)

Figure 1.1: DARPA Challenge vehicles. Stanley (1.1(a)) is shown on the left, while Boss (1.1(b)) is shown on the right.

vehicle technology for the military. The winning team was awarded prize money for their research. In 2005, Stanley, from Stanford University, won the Grand Challenge. In 2007, Boss, from Tartan Racing, won the Urban Challenge.

In 2010, Google announced it was testing autonomous Prii throughout California. According to [3], Google has 7 test cars that have each driven 1,000 miles without human intervention. This Google project is led by Sebastian Thrun, who led the

Figure 1.2: An image of one of Google's Prius autonomous test vehicles.

2005 Stanford team to win the Grand Challenge. He is followed by 15 engineers

devoted to developing autonomous cars.

Another research initiative is the Safe Road Trains for the Environment project (SARTRE) [4], funded by the European Commission. The idea behind this project is to create autonomous highway convoys of personal vehicles which results in significant safety, comfort, and environmental benefits. The lead vehicle in the convoy is a professional driver. The vehicles autonomously following are mainly personal vehicles that can join or leave the train at any time.

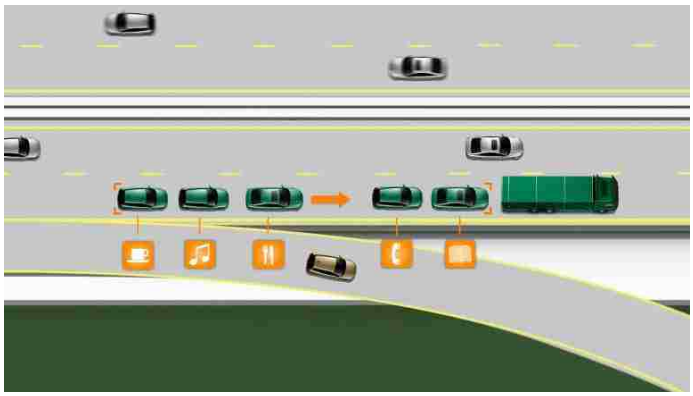


Figure 1.3: An illustration of the concept behind SARTRE.

With the recent push in intelligent ground vehicles and low-cost hardware, a novel research platform is needed in the MARHES laboratory to help test smart car algorithms. Therefore, this thesis focuses on the design and development of an inexpensive mobile robotic platform prototype for the testing of autonomous vehicle algorithms. The prototype vehicle developed in this thesis is a novel platform which is the heart of the ground vehicle testbed in the MARHES laboratory. The Multi-Agent, Robotics, Hybrid, and Embedded Systems (MARHES) laboratory testbed is a heterogeneous robotic testbed incorporating both ground and aerial autonomous vehicles. The MARHES testbed consists of the 10 TXT-1's in which one is the prototype built, 5 MobileRobots Pioneer 3-AT ground vehicles, 3 AscTec Hummingbird quadrotors, and a Dragonflyer X-Pro quadrotor. These platforms are used in the de-

sign and testing of controllers, multi-robot coordination algorithms, sensor networks, and learning algorithms.

The TXT-1 platform is used mainly in the verification of autonomous vehicle, multi-robot coordination, and learning algorithms. The TXT-1 can be thought of as an electric car. Therefore, algorithms used for intelligent highway and city driving, military convoys, search and rescue, intelligent transportation, etc. can be verified on this vehicle. Robotics and learning is another large area of research. Robots can learn behaviors and how to interpret their environment in order to become more autonomous. The TXT-1 prototype is used to verify a simple learning algorithm.

1.2 Literature Review

An overview of literature relating to mobile robot platforms and Q-Learning and robotics will be provided in this section.

1.2.1 Mobile Robot Platforms

Mobile robot platforms have been developed by universities and companies around the world. There are several companies in the robotics field. A few of these include MobileRobotics, Dr. Robot Inc., and Kiva Systems. MobileRobotics produces the widely used Pioneer platforms. The Pioneer 3-AT [5] is an all-wheel differential drive robot capable of driving outdoors. It is a medium-sized platform capable of driving 0.8 m/s with a payload of 30kg and 3 batteries. The P3-AT includes encoders to sense the vehicle's position and velocity; however, other sensors and an on-board computer are add-ons. The platform weighs 12kg, which is fairly heavy, and can run for 3 hours without any additional sensors.

Dr. Robot Inc. produces several ground robots. The Jaguar platform [6], an indoor/outdoor platform, is water-resistant, capable of climbing 45° stairs or slopes, and able to flip over. The Jaguar has GPS, IMU, laser scanner, and cameras included for its sensors. Since the Jaguar uses a wireless router to communicate with it, there is no on-board computer on the platform.

Kiva Systems [7] has pioneered the use of automated material handling systems in store warehouses. The Kiva robots are used to bring products on shelves to packers to increase their productivity and decrease the cost of operations at large retail warehouses. The Kiva robots can carry up to 3000 lbs. and are controlled by a central operations server. These robots are used by the following companies to fulfill orders: Crate & Barrel, Dillards, GAP, Old Navy, Staples, and Walgreens.

Universities create their own mobile robot platforms, because they are used as inexpensive alternatives to buying a commercial mobile robot. The Clodbuster III [8] is an example of a university mobile robot from the Grasp Laboratory at the University of Pennsylvania. The Clodbuster is developed on a Tamiya Clodbuster chassis and has an on-board Pentium III 850MHz laptop and low-level FPGA sensor board. The Clodbuster uses an omnidirectional camera, 12 infrared sensors, and a 2-axis accelerometer to sense its position and surroundings. The platform also uses three batteries to power the system: a laptop battery, 9.6V battery to power the sensor board and camera, and a 7.2V battery to power the vehicle's motors.

Several platforms are based on the Tamiya TXT-1 monster truck chassis like the robot being described in this thesis. For example, the previous work done on the MARHES TXT-1 testbed was completed at Oklahoma State University [9, 10]. The first MARHES TXT-1 developed was a low-cost mobile robot platform with an on-board laptop computer and commercial data acquisition system [9]. Later, in [10], a team of TXT-1's was built for the verification of multi-robot coordination navigation controllers. This robot used a PC104 computer with a few custom microcontroller

boards and a CAN communication network for sensor communication.

Other universities [11, 12, 13, 14] used the TXT-1 chassis in their robotic testbeds. In [11], the TXT-1 chassis is used to create an autonomous mobile robot. A Mini-ITX computer is used with Matlab Real-Time Workshop to run the velocity controller algorithm. Then a hybrid navigation controller is developed which combines potential field and obstacle avoidance controllers for waypoint following. In [12], the TXT-1 chassis is used in conjunction with Matlab and a miniPC to implement a Kalman Filter for tracking the robot's position and orientation. Another TXT-1 platform uses the robots to verify multi-robot coordination algorithms in [13]. In this work, the TXT-1 chassis is used with a 700 MHz computer and 20 MHz microcontroller to control the robot. For this platform, the batteries are stored inside the tires to conserve space and lower the vehicle's center of gravity. The robot uses encoders and vision sensors for navigation and position tracking. In [14], the TXT-1 chassis is used to implement the simultaneous localization and mapping (SLAM) algorithm to determine the trajectory of the robot and map the local environment. The implemented SLAM algorithm uses a stereo camera to obtain the localization and mapping estimates.

1.2.2 Q-Learning and Robotics

Q-Learning has been used in several robot hardware experiments on a wide variety of platforms, including RoboCub players [15, 16], Lego Mindstorms NXT [17], and a self-steering automobile [18]. As is commonly reported, implementation on physical hardware posed some significant challenges [17, 19]. Even navigation and tracking tasks were overwhelmingly popular among robot Q-Learners (as opposed to more complex tasks, such as grasping and manipulation), the robots and their physical environments provided unavoidably noisy learning problems. Such problems can, in

principle, be solved with Q-Learning, but in practice, the learning time required can exceed the mean time between failure for most robot actuators.

One way the problems were kept tractable was to limit the number of sensory channels (inputs) and actions (outputs) available to the agent. In some cases, these were kept coarsely discretized, to minimize the number of states that must be visited [15, 17, 20]. Where inputs and outputs were continuous, function approximators were used [16, 18, 19]. These were multi-layer perceptrons that extrapolated values from previously visited state-action combinations to cover the entire state-action space. Another effective strategy was to pair the Q-Learner with either a higher level mediating control law [15] or with lower level sensory and action primitives and heuristics [17, 20]. Using these strategies, these Q-Learners were able to achieve a broad set of learning objectives across a rich sampling of available robot platforms, illustrating the generality of Q-Learning in robotic applications.

1.3 Contributions

This work presents a novel robotic platform with many advantages over traditional commercial robotic vehicles. The robot is designed with flexibility in mind, which allows different configurations of sensors to be used for different applications. The design also implements a hierarchical architecture that allows components such as the sensors and the on-board computer to be replaced or upgraded without any significant development time. Only open-source or freeware software is used in the creation of the software of the robot. COTS (commercial off the shelf) parts are used when possible. This reduces the cost and the ease of finding parts for the vehicle. It also has a significant amount of processing power located on the vehicle which provides truly autonomous behavior as opposed to robots controlled by a central computer. The designed prototype is a novel vehicle designed to be lightweight and

low-cost in comparison to commercially available vehicles, while being able to test autonomous navigation algorithms. We implemented the Q-Learning algorithm on the TXT-1 prototype in a source navigation problem to verify the platform's use for testing algorithms.

1.4 Thesis Outline

The organization of this thesis is as follows: Chapter 2 discusses the models used, the velocity limits, and the odometry calculations. The parts and design of the TXT-1 robotic platform are explained in detail in Chapter 3. The linear and angular velocity controllers and their responses are introduced in Chapter 4. Chapter 5 describes Q-Learning and the results of the completed experiment. Concluding remarks of this thesis and any resulting work are described in Chapter 6.

Chapter 2

Kinematic Modeling

In this chapter, the kinematics of the mobile robot are derived. Two different models are analyzed: the unicycle and the bicycle models. Then the validity of using the unicycle model for the bicycle model is shown. The odometry calculations and the velocity limits for a bicycle vehicle with steering are also shown. In this study, we assume that the world is perfect and the vehicle does not suffer from wheel slippage and tire deformations.

2.1 Unicycle Kinematic Model

The unicycle model is a simple nonholonomic model that represents the vehicle as a rolling disk as shown in Figure 2.1. Assuming that there is no wheel slip and tire deformation, the nonholonomic constraint,

$$\dot{x}\sin\theta - \dot{y}\cos\theta = 0, \tag{2.1}$$

holds, where $(x,\ y)$ are the Cartesian coordinates of the local frame of the robot and $\theta \in [-\pi, \pi]$ is the orientation of the vehicle to the positive x-axis of the initial local

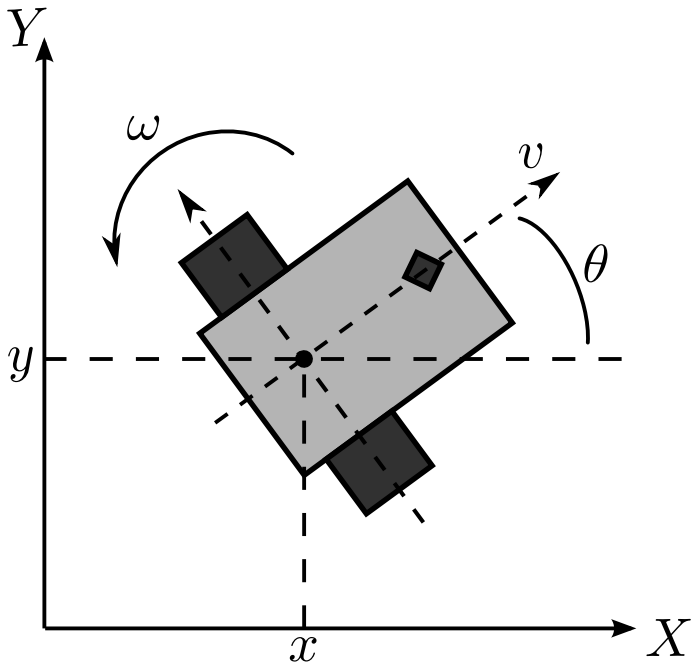


Figure 2.1: The unicycle kinematic model.

frame of the robot. The x-axis of the vehicle frame is orthogonal to the wheel axle, which is the y-axis of the vehicle frame. The positive x-axis points toward the front of the vehicle while the positive y-axis points to the vehicle's left side. Following the right-hand rule, positive θ is in the counterclockwise direction. The nonholonomic constraint limits the vehicle from motion along its y-axis. The unicycle kinematic model is

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}, \tag{2.2}$$

where v is the linear velocity of the vehicle along its x-axis and ω is the angular velocity of the vehicle around its instantaneous center of curvature (ICC). The ICC is the center point of the circle being transcribed by the vehicle at a certain time. In this model, the two wheels on the rear axle can be modeled as one wheel in the

midpoint of the axle, making the vehicle a unicycle.

2.2 Bicycle Kinematic Model

The car-like vehicle is shown in Figure 2.2. In this model, the vehicle has a fixed rear axle with wheels in the front for steering. Like the unicycle model, the front and rear sets of wheels can be modeled as single wheels positioned on the midpoint of their respective axes, making the vehicle model a bicycle. In this model, $(x,\ y)$ are the Cartesian coordinates of the midpoint of the rear axle, θ is the orientation of the vehicle to the x-axis, and ϕ is the steering angle of the front wheels to the vehicle frame's x-axis. The nonholonomic constraints for the front and rear wheels are

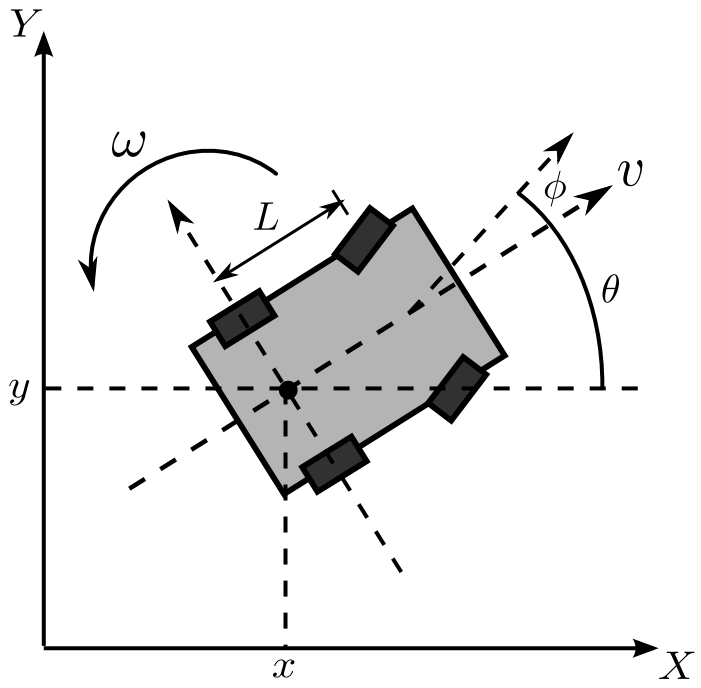


Figure 2.2: The bicycle kinematic model.

$$\dot{x}_f \sin(\theta + \phi) - \dot{y}_f \cos(\theta + \phi) = 0, \tag{2.3}$$

$$\dot{x} \sin\theta - \dot{y} \cos\theta = 0, \tag{2.4}$$

where $(x_f,\ y_f)$ are the Cartesian coordinates of the midpoint of the front axle. The midpoint of the front axle is found using

$$\begin{aligned} x_f &= x + L\cos\theta, \\ y_f &= y + L\sin\theta, \end{aligned} \tag{2.5}$$

where L is the length between the front and rear axles. Then the velocity of the front axle midpoint is found by taking the derivatives of (2.5) to obtain

$$\begin{aligned} \dot{x}_f &= \dot{x} - L\dot{\theta}\sin\theta, \\ \dot{y}_f &= \dot{y} + L\dot{\theta}\cos\theta. \end{aligned} \tag{2.6}$$

Then $\dot{\theta}$ can be solved by first substituting (2.6) into (2.3) to obtain

$$(\dot{x} - L\dot{\theta}\sin\theta)\sin(\theta + \phi) - (\dot{y} + L\dot{\theta}\cos\theta)\cos(\theta + \phi) = 0, \tag{2.7}$$

which reduces to

$$\dot{x}\sin(\theta + \phi) - \dot{y}\sin(\theta + \phi) - \dot{\theta}L\cos\theta = 0. \tag{2.8}$$

Next $\dot{\theta}$ is solved by using the sum-difference trigonometry identities and (2.4) to obtain

$$\dot{\theta} = \frac{v\tan\phi}{L}, \tag{2.9}$$

where the linear velocity is

$$v = \dot{x}\cos\theta + \dot{y}\sin\theta. \tag{2.10}$$

The resulting bicycle kinematic model is

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ \frac{\tan\phi}{L} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ u \end{bmatrix}, \tag{2.11}$$

where u is the steering velocity input. The bicycle model has a singularity at $\phi = \pm\frac{\pi}{2}$ which occurs when the steering wheels of the vehicle are perpendicular to the direction of the linear velocity. This does not usually occur because vehicles are not designed to operate this way. In this case, the vehicle is not able to move because the front wheels are orthogonal to the direction of the force of the rear wheels. This restriction limits the vehicle from turning in place while having a linear velocity of 0. The inputs to the bicycle model are the linear and steering velocities.

2.2.1 Dual Steering Bicycle Model

A vehicle can have steering on both the front and the rear axles, which provides a smaller turning radius as compared to the normal bicycle model. With dual steering, the vehicle can reach the desired orientation quicker than with single steering. The addition of rear steering does not change the kinematic model from the single-steered bicycle model except by doubling the steering angle. In this model, it is assumed that the steering angles are equal in angle and opposite in direction to one another. The model is shown in Figure 2.3 and the kinematic model is given by

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 2\frac{\tan\phi}{L} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ u \end{bmatrix}. \tag{2.12}$$

2.2.2 Modeling the Bicycle Model as a Unicycle

In this work, a car-like vehicle with dual steering is developed, therefore the dual steering bicycle model should be used. However, since the steering velocity input, $\dot{\phi}$, of the bicycle model is difficult to control, it would be more practical to control the

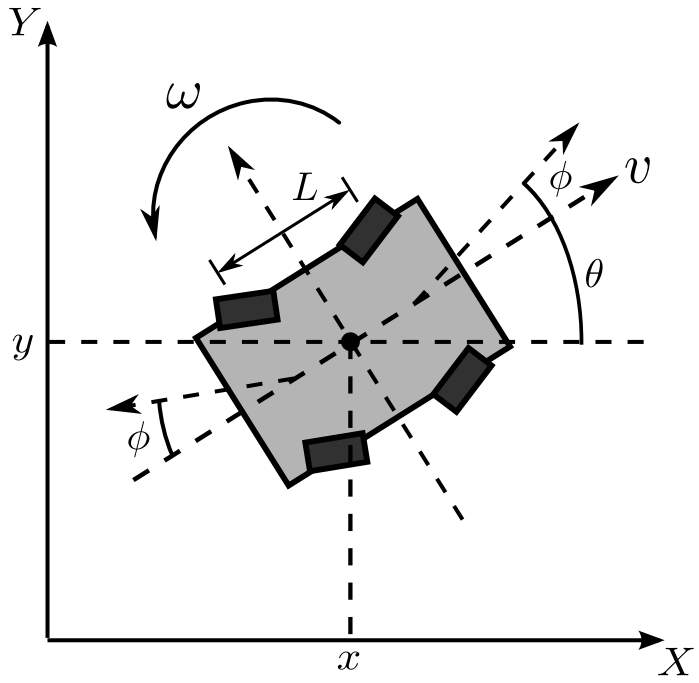


Figure 2.3: The dual steered bicycle kinematic model.

angular velocity, ω, of the vehicle. Therefore, the bicycle model is reduced to the unicycle model. Instead of controlling the steering velocity, $\dot{\phi}$, the angular velocity is controlled with

$$\omega = 2\frac{\tan\phi}{L}v, \tag{2.13}$$

which holds for $v \neq 0$.

2.2.3 Maximum Angular Velocity

Since there are limitations in the maximum steering angle of the car-like vehicle, the maximum reachable angular velocity, $\omega_{\max}$, is discussed. In real-life applications, the maximum angular velocity poses limits on the turning radius of the vehicle, which affects its path planning. For example, to parallel park, it is necessary that a car-

like vehicle plans a series of forward and backward maneuvers in order to achieve the desired final position of the vehicle. This is due to its minimum turning radius. Alternatively, a differential drive vehicle, which has a unicycle model and is capable of a zero turning radius, does not have the limitations of a car-like vehicle. Using the maximum steering angle allowed by the steering servos in the prototype vehicle, the minimum radius of curvature was found to be $R_{\min} = 0.56$ meters. The maximum angular velocity is limited by

$$-\omega_{\max} \leq \omega \leq \omega_{\max}, \tag{2.14}$$

where positive ω results in a counterclockwise rotation. The maximum angular velocity for a constant linear velocity, v, can then be calculated using

$$\omega_{\max} = \frac{v}{R_{\min}}. \tag{2.15}$$

Then by substituting Equation 2.15 into Equation 2.14, the angular velocity limits of the vehicle are obtained by

$$-\frac{v}{R_{\min}} \leq \omega \leq \frac{v}{R_{\min}}. \tag{2.16}$$

From Equation 2.16, the maximum angular velocity limits are obtained from the minimum radius of curvature. Figure 2.4 shows the limits of the angular velocities with their corresponding linear velocities.

2.3 Odometry Calculations

In this section, the wheel odometry of the vehicle is derived, which is how the vehicle calculates its position and velocity from knowing the rotational velocities and circumference of its wheels. Figure 2.5 shows a vehicle with the important variables labeled to derive the odometry calculations. It can be seen that the linear velocity

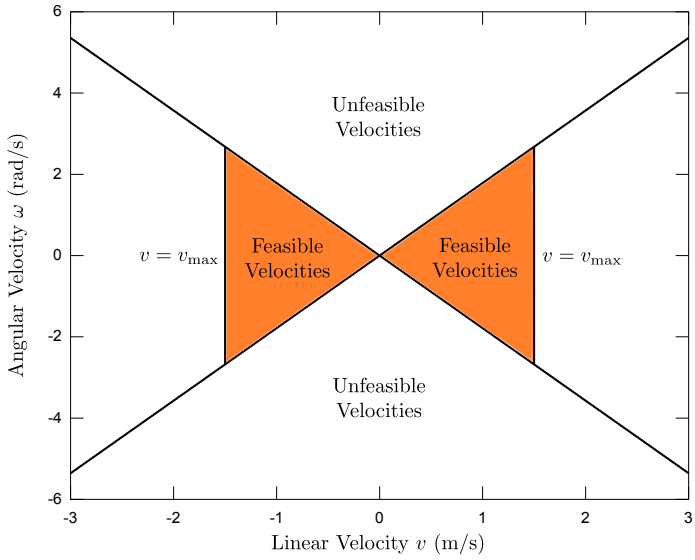


Figure 2.4: The reachable linear and angular velocities.

of the vehicle, v, can be calculated as follows

$$v = \frac{(r_L + \frac{W}{2})\theta}{\Delta t}, \tag{2.17}$$

where W is the width between the wheels,

$$W = r_R - r_L, \tag{2.18}$$

Δt is the change in time between measurements, and r_R and r_L are the radii to the ICC of the right and left wheels. Using (2.18), v becomes

$$v = \frac{r_L + r_R}{2}\frac{\theta}{\Delta t}. \tag{2.19}$$

It can also be seen that the rotational velocities of the right and left wheels, v_R and v_L, are

$$\begin{aligned} v_R &= \frac{s_R}{\Delta t} = \frac{r_R\theta}{\Delta t}, \\ v_L &= \frac{s_L}{\Delta t} = \frac{r_L\theta}{\Delta t}, \end{aligned} \tag{2.20}$$

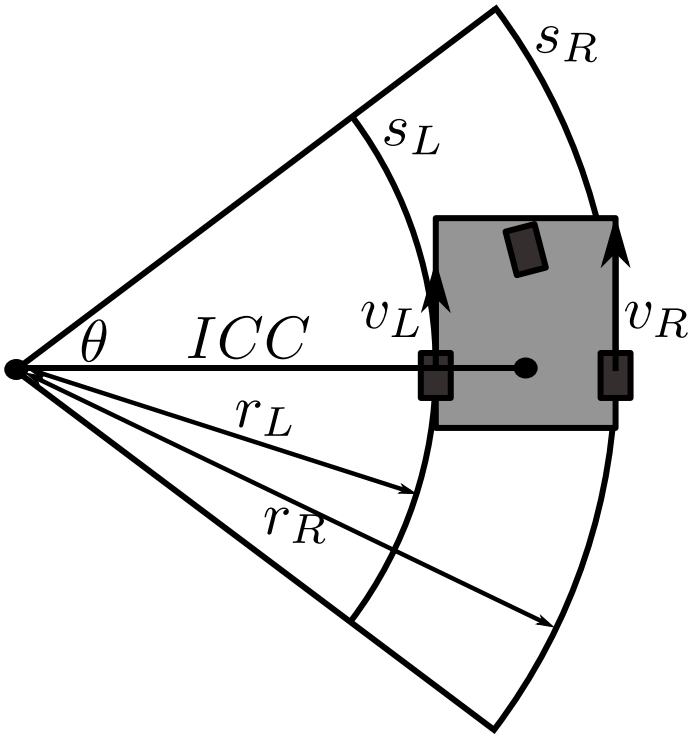


Figure 2.5: Calculating the wheel odometry.

where s_R and s_L are the distances traveled by the right and left wheels during the sampling interval, Δt. By substituting these into (2.19), the following linear velocity equation is obtained

$$v = \frac{v_R + v_L}{2}. \tag{2.21}$$

The angular velocity, ω, is calculated by observing that $\omega = \frac{\theta}{\Delta t}$. Then this is substituted into (2.20) to obtain

$$\begin{aligned} v_R &= r_R\omega, \\ v_L &= r_L\omega. \end{aligned} \tag{2.22}$$

These equations are then subtracted from each other to get

$$v_R - v_L = (r_R - r_L)\omega. \tag{2.23}$$

Next (2.18) is substituted to obtain the final odometry equation for the angular velocity,

$$\omega = \frac{v_R - v_L}{W}. \tag{2.24}$$

Equations (2.21) and (2.24) calculate the velocities of the vehicle from the left and right wheel speeds. By using the wheel speeds of the vehicle the robot can calculate its linear and angular velocities for the vehicle's velocity controller. Also, the vehicle can use its kinematic model to calculate its position and orientation in its local frame.

Chapter 3

TXT-1 Platform Description

3.1 System Overview

This chapter describes the system's hardware and software components. The TXT-1 prototype is part of a multi-vehicle testbed, of which there are 10 of these robots. The prototype is a novel robotic testbed that uses existing hardware from the previous designs [9, 10]. The existing platform, new enhancements, and the hardware and software are explained in detail in the following sections. Figure 3.1 shows the overall architecture of the system. As a part of this design, each robot has its own lower level controller to control velocities and other low-level functions. The lower level controller communicates with an on-board computer which handles high-level control tasks and sensor data processing. This design also leverages the use of a single battery to simplify maintenance, so a power distribution system is designed and implemented. The robotic development framework encourages code reuse and the use of simulation environments to test algorithms before implementing them in hardware. The newly designed, low-cost mobile robot is designed to help in the study and research of mobile robotics, sensor networks, cooperative control, cyber-physical

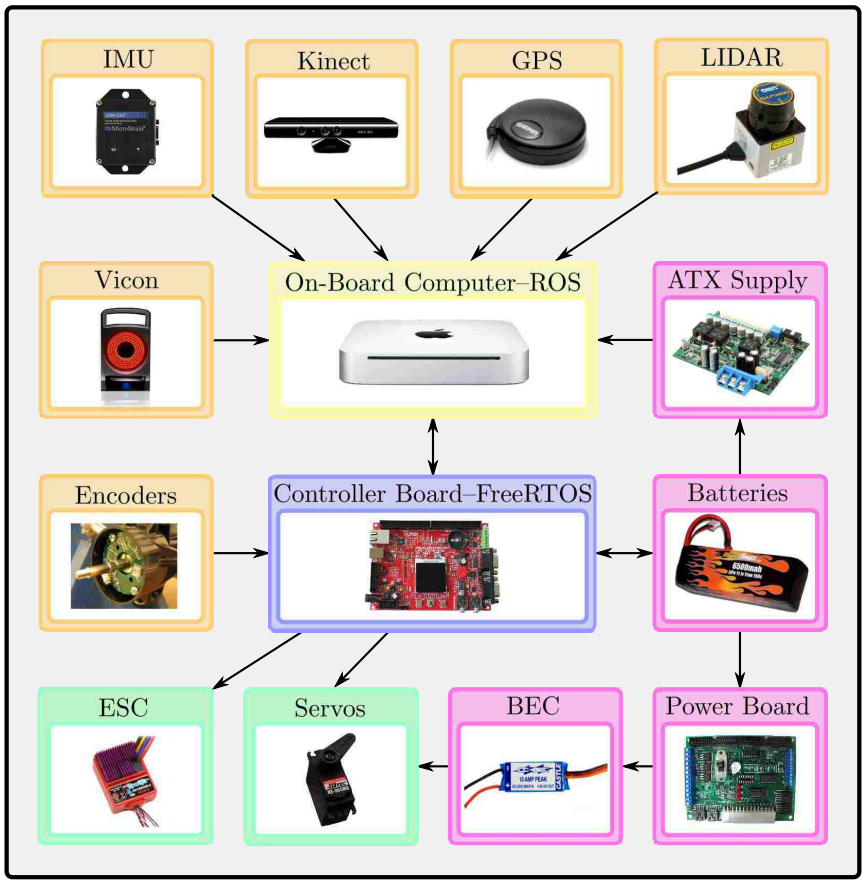


Figure 3.1: The overall architecture of the TXT-1 platform.

systems, learning algorithms and many other application areas.

3.2 Vehicle

To reduce the cost and the amount of custom parts, a commercial vehicle chassis is used. The chosen chassis is the Tamiya TXT-1 4x4 1:10 scaled remote control monster truck [21]. The TXT-1 is shown in Figure 3.2.

Figure 3.2: The commercial Tamiya TXT-1 4x4 R/C monster truck.

3.2.1 Specifications

Some of the specifications of the TXT-1 platform are discussed in this section. The TXT-1 vehicle has two motors with drive shafts that run in parallel to produce more torque at the wheels. It also has front and rear steering servos to decrease the turning radius of the platform. The vehicle has a cantilever suspension which allows the vehicle to perform well outdoors and drive over rough terrain. The TXT-1 is also small enough, with dimensions of 510mm long by 385mm wide by 297mm tall, to be used in an indoor laboratory environment. Since this vehicle is a car-like vehicle, the motor and steering angle inputs are controlled.

3.2.2 Modifications

In order to transform the chassis into a robot designed for scientific experiments, modifications are made to the chassis. First, the truck shell is removed and a plate is fabricated to attach the sensors, on-board computer, and other necessary devices to

the chassis. This plate was made in the previous version of the robot. It is a milled aluminum plate that is designed to be lightweight and flexible enough to secure a variety of sensors. The plate is 11 inches wide, 18 inches long, and weighs about 1.3 lbs. Threaded 4-40 screw holes are added to the plate to accommodate the current sensors and computer brackets.

The on-board computer is attached to the plate described above. In order to do this, a smaller plate was designed to secure the computer to the robot. The plate sandwiches the computer between it and the larger plate on the chassis, and keeps the computer from moving vertically. Then, plastic retaining fixtures are made on a 3-D printer to secure the computer from moving on the horizontal plane of the large plate. The plates and retaining fixtures are lined with rubber strips to absorb some vibrations and protect the computer from marring. This plate is also used to secure the lower level control board, power distribution boards, cameras, and the GPS and IMU mounting pole.

An additional Lexan plate is created to mount the PCBs used on the robot. The PCBs are mounted to the plate with plastic standoffs. The plate is mounted to the small plate for the computer. The camera is mounted to the front of the small plate with a custom fixture. The Microsoft Kinect can be mounted in place of the camera with a modified commercial mount. The PDP Kinect Sensor Mounting Clip is used to mount the Kinect. To modify it, first it is disassembled and the back arm is taken off and discarded. Then through holes are drilled in the mount so it can be secured to the small plate with screws. The GPS and IMU are mounted on a fixture on a pole, which is attached to the small plate above the robot, to protect the sensors from noise. The laser range finder is mounted to the large plate with a bracket plate. The inverter and an extra battery are mounted to the bottom of the large plate with metal straps lined with rubber strips. An additional battery is stored in the TXT-1's original battery compartment. The TXT-1 robot, with its custom

Figure 3.3: The TXT-1 Robot with the mounting hardware, plates and electrical systems.

machined mounting brackets, plates, and electrical hardware, is shown in Figure 3.3.

The wheel well of the TXT-1 requires modifications in order to accommodate the wheel encoders which can sense the wheel's speed and direction. The optical encoders are mounted in the front left and right wheel wells, which when used together, the linear and angular velocities can be calculated for odometry and the velocity controllers. Installation of the wheel encoders requires modifications to the wheel wells. These modifications are done on the previous version of the robot and are listed below. First, mounting holes are drilled into the back of the wheel well and standoffs are secured so the optical encoder PCB can be mounted inside the wheel

well. Then the inside part of the wheel well is machined to reduce its length by 0.5 inches. Next the PCB is mounted inside the wheel well on the installed standoffs and the patterned disc is fastened with a set screw on the wheel axle.

On the previous version of the robot, the optical encoders were poorly connected to the cable, resulting in unreliable electrical connections. The encoders come with a PCB connector and a connectorized cable from the manufacturer. However, these were modified in the previous version to allow the encoder to fit in the wheel well. The connector on the cable was cut off and the cable's wires were soldered directly to the PCB connector and routed through a hole in the back of the wheel well

(a) (b)

Figure 3.4: Images of the wheel encoder assembly and modifications done to the wheel well. 3.4(a) shows the modifications done previously. 3.4(b) shows the modifications done to fix the unreliability.

resulting in unreliable, unsecure connections to the encoders. Therefore, new cables are purchased and new connectors are soldered to the PCB's. The PCB with the connector does not fit in the wheel well, therefore, a notch is cut in the side of the wheel well to allow the connector to fit. Then the cable is routed tightly against the wheel well and along the chassis frame. The use of the connector greatly improves the reliability and ease of maintenance of the encoders. The wheel well modifications for the encoders are shown in Figure 3.4.

The electronic speed control (ESC) is reused from the previous version of the robot. The ESC used is the Novak Super Rooster [22], which is shown mounted on the TXT-1 in Figure 3.5. The ESC controls the speed and direction of the TXT-1's motors from a servo signal. The servo signal is a 50 Hz pulse-width modulated (PWM) signal with an adjustable duty on-time of 1-2 ms. At 1 ms, the motors move the robot at full speed in reverse. At 1.5 ms, the motors are stopped, and at 2 ms the motors move the vehicle forward at full speed.

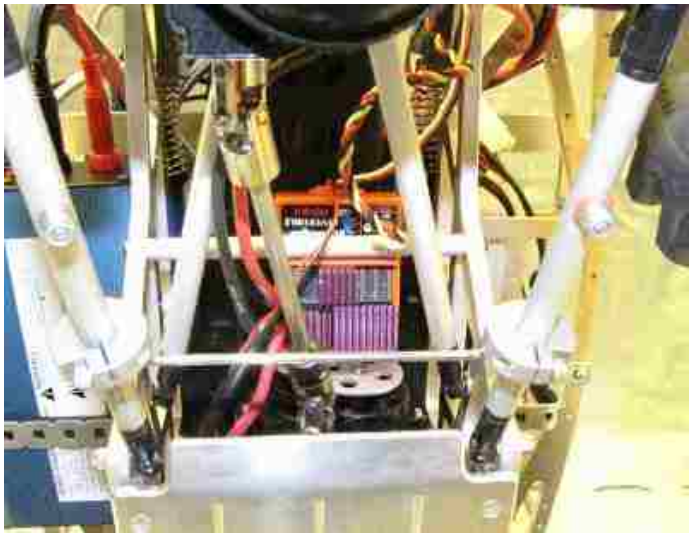


Figure 3.5: The Novak Super Rooster mounting location.

3.3 Low-Level Control Board

On a robot, there are many low-level tasks that need to be accomplished in order for the robot to operate. These tasks usually require some hardware that does not come on normal computers. Some of these tasks include the following: *a*) reading the encoders and calculating the robot's linear and angular velocities, *b*) control-

ling the steering and motor servo outputs, *c*) performing the PID velocity control loop, *d*) tracking the robot position, *e*) running battery measurements to calculate their remaining life, *f*) warning the user when the batteries are low or shutting down when unsafe, and *g*) communicating with a high-level controller. These tasks can be handled by connecting a few USB devices to the on-board computer. For example, Phidgets sells USB to encoder, analog input, servo, and digital output devices. However, the total price of these components becomes high in comparison to using a microcontroller in place of these devices. Some other benefits to using a microcontroller are its ability to do the following: *a*) flexibly choose inputs and outputs, *b*) run without the need of an on-board computer, and *c*) offload some of the computing power needed to run the necessary tasks. Because of these benefits, a microcontroller is chosen instead of using the USB devices. In order to save cost, maintenance, and replacement cost, an off-the-shelf commercial development board is used. This section describes the chosen low-level microcontroller development board, the programming environment, and the real-time operating system used.

3.3.1 Specifications

Since the low-level controller requires the tasks listed above, the selected microcontroller needs to have some standard peripherals. These include a UART for communication, PWM controller for the servo outputs, analog to digital converter (ADC) inputs for the battery measurements, and digital outputs for warning indicators and the power control. In order to meet these requirements and be expandable in the future, the Olimex LPC-2378-STK development board was chosen [23]. The LPC2378 microcontroller is an ARM 16/32 bit microcontroller with an ARM7TDMI-S^{TM}core. The specifications of this development board and microcontroller are listed in Table 3.1. Figure 3.6 shows the lower level control board and the connections that are used for sensors and the on-board computer.

Specification
72MHz Maximum Clock
512 kBytes Program Memory
16 kBytes RAM Memory
Real Time Clock
4 10-Bit ADCs
4 UARTs with 2 RS-232 Ports
2 CAN Ports
Ethernet
USB
3 I^2C
SPI
6 PWM Outputs
SD Card Interface
128x128 Pixel TFT LCD Display
User Control Buttons and 4-Way Joystick
5V Tolerant Inputs
144 Pin Package with 104 General Purpose IO

Table 3.1: The LPC-2378-STK board specifications.

3.3.2 Development System

Another way to reduce the price of the TXT-1 platform is to use open-source code development tools. The LPC2378 board is programmed over a JTAG interface, so an Olimex ARM-USB-OCD JTAG interface was purchased. This interface allows the microcontroller to be programmed and debugged during development. The OCD (On Chip Debugger) interface can use OpenOCD which is an open-source on-chip debugger and programming solution for ARM7, ARM9, and Cortex-M3 targets. OpenOCD uses GDB, the standard open-source GNU debugger, in order to debug and step through the code. To compile the low-level code, Codesourcery G++ Lite for ARM EABI version 2010q1-188 is used [24]. This is a free ARM GNU compiler which provides commands, *make* and *GDB*, for ARM microcontroller targets. Since GDB is a command line debugger, GNU's DDD (Data Display Debugger) is used

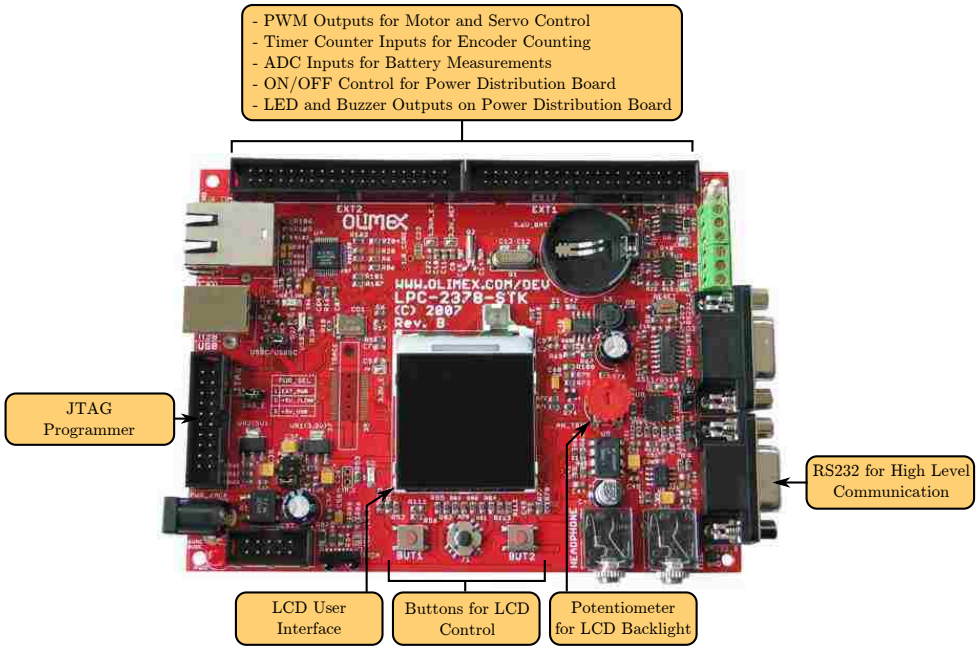


Figure 3.6: The LPC2378 development board and its used connections.

to graphically debug source code. Because OpenOCD and DDD are started over the command line with many additional arguments, scripts are used to make the programming and debugging processes quicker and more simplified. Using all of these tools allowed a free low-level development system to be created.

3.3.3 FreeRTOS

To simplify the design of the lower level controller code and provide robust real-time operation, a real-time operating system, or RTOS, is used. Again free, open-source real-time operating systems were searched for and FreeRTOS [25] was chosen. FreeRTOS is an open-source RTOS that has a small memory footprint and ports to many different architectures of microcontrollers.

The RTOS is beneficial to use for the controller board because hard real-time

requirements can be met. Real-time operating systems are made up of a collection of independent threads. However, the microcontroller can only execute one task, or thread, at a time. Therefore, the RTOS also includes a task scheduler that switches between threads based on priority. When programming with an RTOS, tasks are created in an infinite while loop and are assigned a priority. The RTOS then uses a scheduler with a timer interrupt to switch between tasks based on priority and meeting hard real-time deadlines. Figure 3.7 shows how the scheduler switches between tasks. In the figure, there is the idle task, Task 1 and Task 2. Tasks 1 and 2 have

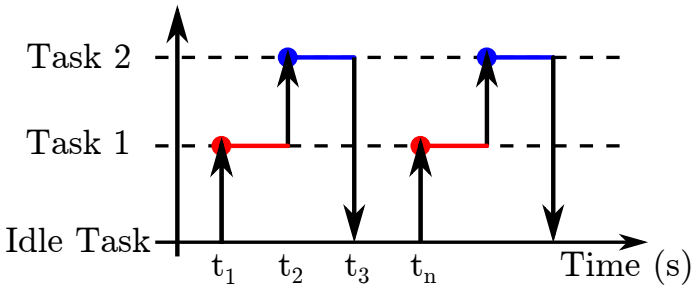


Figure 3.7: An example of RTOS task scheduling.

the same priority and have a higher priority than the idle task. Tasks are usually periodic, so at time t_1, the time has come to service Task 1 so the scheduler switches to that task. Then when Task 1 completes its function, it goes into a blocking mode where it releases its time slot. Then the scheduler switches to Task 2 because it was also supposed to occur at time t_1 also. Then when Task 2 is finished at time t_3 and there are no other tasks to service, the code switches to the idle task until time t_n, when the process repeats itself. This shows an example of RTOS scheduling. The RTOS also provides queues, mutexes, and semaphores to manage the resources of the microcontroller and avoid conflicts with accessing memory locations that have been modified.

The low-level controller code consists of the FreeRTOS, driver, and task files. The

FreeRTOS files include source files for the scheduler, task creation, queue and memory management, and semaphore/mutex creation. On the LPC2378 development board, the UART, ADC, LCD, PWM, and timer counters are used. Driver files were produced for these peripherals. The bulk of the code for the low-level controller is in the tasks. For the task scheduler, 5 task priorities were used, 0 to 4, with 4 being the highest priority. A functional description and the priority and frequency of each task is given below.

Controller Task The controller task calculates the PID control loop motor and steering outputs for the linear and angular velocity controllers, respectively. It has a priority of 4 and hard real-time requirements. It also runs at 50 Hz. The controller design and implementation is discussed in Chapter 4.

Encoder Task The encoder task calculates the speeds and directions of the front right and left wheels using the encoders' quadrature signal counts from the timer counter inputs. The linear and angular velocities are calculated from the wheel speeds and then used in the velocity controllers. This task has a priority of 3 and runs at 50 Hz. The timer counter is used to count both rising and falling edges of a modified quadrature signal, which is described in detail in Section 3.7.1.

Serial Processing Task In order to command the robot to move and perform other functions, an RS-232 communication link is used. With the link, the low-level and high-level controllers can communicate with each other. This task parses the messages sent by the high-level computer and provides functions to send data to the high-level computer. The task runs continuously at the 4th priority level.

Mode Task The mode task is used to switch between a couple of states. With the TXT-1, there are two sources of odometry. In this section, they will be

described as the encoder odometry and the combined odometry. A Kalman filter can be computed on the high-level computer and then sent to the lower level controller over the serial interface to be used in the PID controller calculations. This task determines if an adequate amount of Kalman filter messages have been received in a certain amount of time. If the message count is high enough, then the Kalman filter velocities are used in the velocity controllers. If not, then the wheel encoder velocities are used instead. The task also checks if an adequate amount of commanded velocity messages are received in a certain amount of time. If so, then the velocity controller is used. However, if an adequate amount of the velocity messages are not received, then the controller is turned off and the robot is immobilized until communication is restored. The ROS driver on the computer is responsible for sending the velocity messages periodically. This is a safety feature that stops the robot if communication has been lost between the high and low level controllers. This task has a priority of 2 and runs at 1 Hz.

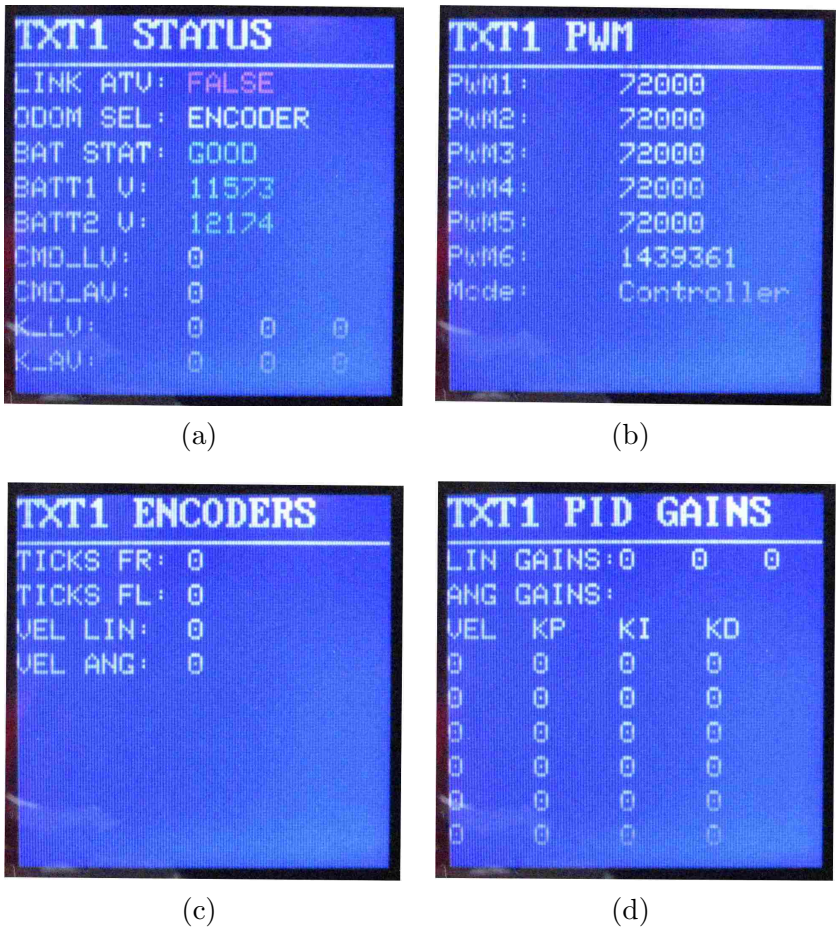
Battery Task Because the robot is powered from batteries, the batteries need to be monitored to detect their state of charge. Also, some applications need to know the remaining battery life for energy critical operations. This task measures the battery voltages, calculates the estimated battery life and sends the information to the high-level computer. This task runs at 1Hz and has a priority of 1.

Display Task The LPC2378 board has a 128x128 pixel LCD display that is used as an operator interface for communicating the robot state and debugging information to the user. This helps the user to debug the lower level controller if an error were to occur. Four LCD display screens are designed to infer information to the user. These screens are shown in Figures 3.8(a), 3.8(b), 3.8(c), and 3.8(d). Figure 3.8(a) shows the *TXT1 Status* display. This display shows

the status of the two modes, the battery voltages and states, the commanded velocities, and the current gains for the linear and angular velocity controllers. Figure 3.8(b) shows the *TXT1 PWM* display. The PWM values are used to control the main motor and servos. This is helpful in debugging the controllers. *PWM6* is used to control the brightness of LCD screen. The on-time of *PWM6* is calculated from the analog voltage read from the board's potentiometer. The PWM mode can be either *controller* or *manual*. *Controller* mode is when the PID controller controls the PWM values. *Manual* mode is when the PID controller is bypassed and the PWM values are controlled with PWM messages over the serial link. The modes are switched by pressing the joystick button. The *manual* mode is useful in determining the PWM limits or when a controller is being developed on the higher level controller. Figure 3.8(c) shows the values of the wheel encoders and calculated velocities. This display is used to test the functionality of the encoders. Figure 3.8(d) shows a list of all the gains that were loaded into the lower level controller. The up/down joystick buttons are used to scroll up and down the screen to see all the values. The display task runs every 100 ms at a priority of 2 because it has soft real-time requirements.

Button Task The button task is used to get the change in state of the user buttons. The state is changed on the button press. The buttons are used to switch between display screens and switch the PWM outputs from being operated by the controller or by manual messages sent from the high-level computer. The manual mode is used to determine the limits of the PWM outputs. The button tasks runs at a priority of 1 at 10 Hz.

All of the tasks described above work together to make a functioning and safe mobile robot. The tasks make the system modular and the RTOS ensures a robust real-time low-level robot controller.



(a) (b)

(c) (d)

Figure 3.8: Images of the designed display screens.

3.4 On-board Computer System

In order for the TXT-1 to be an autonomous robot, it must to be able to process a large amount of data, interpret the data, and make decisions based on the data. These tasks require more processing power and resources than the low-level controller can provide. Therefore, an on-board computer system is chosen to handle these tasks. Additionally, a standard and widely used development environment is needed to make programming robot applications and experiments less time consuming and tedious. Consequently, an open-source robotic operating system is implemented on

the on-board computer system. The on-board computer and the robotic development environment are described in the next sections.

3.4.1 Specifications

Some of the requirements of the on-board computer are as follows: *a*) It needs to have a fast processor and enough memory to handle processing the sensor data and make the appropriate decisions. *b*) The computer must communicate with the low-level controller. *c*) It must be small enough to be mounted on the robot, and light enough to not considerably affect the dynamics of the robot. *d*) In order to communicate with other computers and robots, it should have a standard wireless connection. *e*) Because the robot was designed to use only one main battery, the on-board computer should not have a battery. *f*) The computer should be reasonably priced for its capabilities. These requirements were taken into account when searching for a on-board computer and an Apple Mac Mini was chosen [26]. The specifications of the Mac Mini are shown in Table 3.2. These specifications fulfill the requirements

Specification	Value
Processor	2.4 GHz Intel Core 2 Duo
RAM	2 GB DDR3 SDRAM
Hard Drive	320 GB
Networking	802.11n, Bluetooth 2.1, 1000BASE-T Ethernet
Graphics Card	NVIDIA GeForce 320M
Peripheral Connections	1x Firewire 800, 4x USB 2.0, SD Card, Audio
Size	7.7 x 7.7 x 1.4 inches (w x l x h)
Weight	3 pounds
Voltage	120 VAC
Max Power	85W
Price	$699

Table 3.2: The specifications of the Apple Mac Mini.

stated above. The Mac Mini has a suitable amount of processing power, is very

small and lightweight, has suitable communication capabilities, and is powered with an external power source.

3.4.2 ROS – The Robot Operating System

There are many libraries and operating systems available for robotics research in order to speed up and simplify development. Some of the robotic libraries include Player, Pyro, RoboMind.net, Microsoft's Robotics Developer Studio, and ROS. The requirements for the robotics development environment include that it is free, has continuous updates and improvements, has a community-based code database, standardization of sensor messages, and the ability to run multiple processes on multiple computers. ROS was chosen because it meets all of these requirements.

ROS, the Robot Operating System, is developed by Willow Garage [27]. ROS is an open-source, meta-operating system that provides services to make development of robotic applications easier. It provides device drivers, standard data messages, package management, and message passing between processes. ROS provides the tools and libraries for building, obtaining, writing, and running code across multiple computers. The ROS runtime graph is a peer-to-peer network of processes that are loosely coupled using the ROS communication infrastructure.

ROS provides the ability to send data between processes using three different methods. ROS can be used to send asynchronous streaming messages between processes called topics. With this message type, one node or process publishes data to a topic, and one to many nodes subscribe to the topic. The second message format synchronously passes messages using a service. With this method, a node sends a request message to another node which in turn sends a reply message to the requesting node. The last method uses a parameter server to store data or parameters that all nodes can use. Using message passing between processes allows robotic applications

to be developed that are faster, because they can run using multiple processes and not just one while loop.

3.4.3 TXT-1 ROS Driver

For the TXT-1 robot, a ROS driver was created in order for robotic applications to communicate with the low-level control board. The ROS driver has its own topics, services, and parameters that user code can access and modify in order to control the robot. The driver acts as a ROS interface to the lower level controller which encapsulates the communication protocol described in the next section. The topics are shown in Figure 3.9 as square boxes and the TXT-1 driver node is shown as an oval. The services are shown to the right.

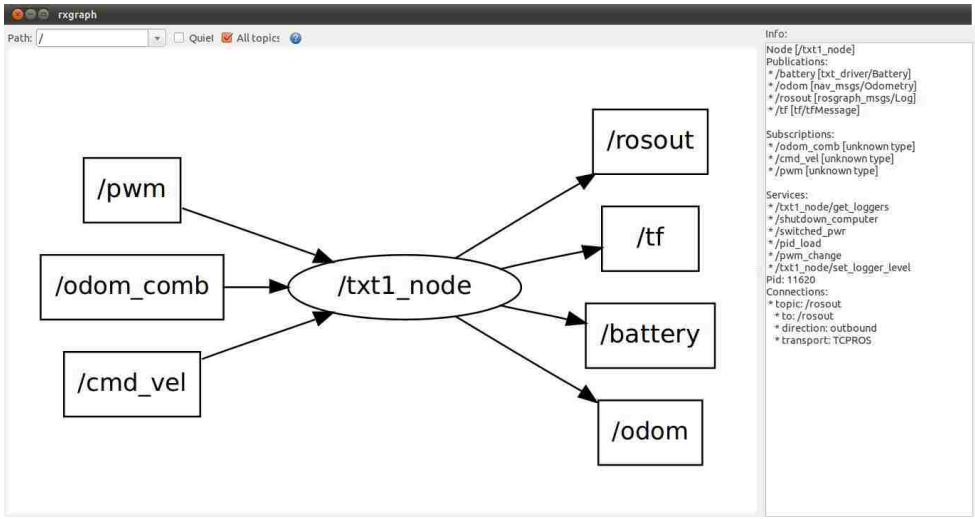


Figure 3.9: The TXT-1 node's topics and services.

The TXT-1 node subscribes to the commanded velocity, combined odometry, and PWM messages. The commanded velocity message is used to send velocities for the lower level controller to follow. A *geometry_msgs/Twist* message is used with the

cmd_vel topic name because this is the standard ROS message used for commanding velocities. The *Twist* message is a part of the *geometry_msgs* package and consists of two 3-D vectors as shown below:

```
Vector3 linear
  float64 x
  float64 y
  float64 z
Vector3 angular
  float64 x
  float64 y
  float64 z.
```

For the *cmd_vel* message, only the `linear.x` and `angular.z` values are used to control the linear and angular velocities, which are in m/s and rad/s.

The combined odometry message is used as an alternative to using the wheel encoders to sense the velocity. The combined odometry measurements fused from sources like the encoders, IMU, GPS, and visual odometry can be used as the measurements for the PID controllers, which allow algorithms to be developed for testing sensor failures. The combined odometry uses the *nav_msgs/Odometry* message with the *odom_comb* topic name. The *Odometry* message is in the *nav_msgs* package and consists of the following:

```
Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
```

```
    geometry_msgs/Pose pose
      geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
      geometry_msgs/Quaternion orientation
        float64 x
        float64 y
        float64 z
        float64 w
    float64[36] covariance
  geometry_msgs/TwistWithCovariance twist
    geometry_msgs/Twist twist
      geometry_msgs/Vector3 linear
        float64 x
        float64 y
        float64 z
      geometry_msgs/Vector3 angular
        float64 x
        float64 y
        float64 z
    float64[36] covariance.
```

The *Odometry* message consists of a timestamp, position and orientation with a covariance and the *Twist* message with a covariance. Only the *Twist* message of the *Odometry* message is used for the combined odometry.

In order to control the two extra PWM signals on the low-level controller, the PWM message is used. This is a custom message within the *txt_driver* package called

Pwm. The message format is as follows

```
int32 pwm4
int32 pwm5
```

where `pwm4` and `pwm5` are the PWM outputs of channels 4 and 5. Their values have a range of $[-24000,\ 24000]$ which represents a 1-2 ms on-time of the PWM signal. This message can be used to control up to 2 extra servos like those on a pan-tilt camera mount.

The TXT-1 ROS driver also publishes the battery status and the wheel encoder odometry from the low-level controller board. The battery status message is a custom message of type *txt_driver/Battery* and has a topic of *battery*. The message format is

```
Header header
  uint32 seq
  time stamp
  string frame_id
float64 batt1
float64 batt2
duration expected_time
```

where `batt1` and `batt2` are the voltages of battery 1 and 2. These batteries are diode OR'ed together on the power distribution board so the individual voltages of the batteries can be measured. The `expected_time` is the time left until the battery runs out, which uses the `ros::duration` data type. The driver uses the battery life plot in Figure 3.13 to determine the remaining operating time left on the battery. The `expected_time` variable calculation does not include the effects of heat, number of discharges/recharges, or other significant variables that have an

effect on the remaining capacity of the battery. The wheel encoder odometry is published using the *nav_msgs/Odometry* message shown above on the *odom* topic. This message provides the calculated odometry data from the lower level controller in the form of a position vector, orientation quaternion, linear velocity vector, and angular velocity vector. The variable, θ, can be extracted from the quaternion using the static function `tf::getYaw(const geometry_msgs::Quaternion & msg_q)`.

The parameters used on the parameter server are the port, and linear and angular velocity PID gains. The *port* parameter specifies the serial port that the TXT-1 driver should communicate to the low-level control board. The linear and angular gains are array parameters corresponding to gains that the TXT-1 should send the controller in the PID load service. The *linear_pid* parameter is an array of 3 floats,

```
[proportional, integral, derivative],
```

which are the PID gains of the linear velocity controller. The *angular_pid* parameter is another array of floats; however, it has the gains for a gain scheduling angular velocity controller. The array is

```
[..., max linear velocity, proportional, integral, derivative, ...]
```

where the maximum linear velocity is the limit for this set of PID gains. There can be up to 14 sets of gain values for this array making the maximum array size 56. The gain sets should be ordered by the maximum linear velocity in increasing order.

The services of the driver include PID load, shutdown, PWM test. and switched power. All of these services use custom messages. The PID load service sends the PID gain table parameters to the low-level controller. The PID gain table needs to be loaded first in order to use this service. On the TXT-1 start up, this service is automatically called in order to load the gains onto the lower level controller. To

use this service, call the *pid_load* service with a *std_msgs/Empty* message. Then the PID gains loaded on the parameter server will be loaded. The service will return a *bool* value indicating a success or failure.

The shutdown service shuts down the Mac Mini, which can be used in normal operation or for certain algorithms. To do this, a *bool* message is sent to the *shutdown_computer* service. The TXT-1 driver then issues the `sudo shutdown -h now` command to shutdown the computer. This command requires superuser privileges, so the `/etc/sudoers` file needs to be modified by adding this line:

```
%admin ALL = NOPASSWD: /sbin/shutdown.
```

The PWM test service allows the user to manually set the PWM values of the lower level controller. This is only used when the lower level controller is configured in manual PWM mode, which is entered when the center joystick button is toggled. Then the *txt_driver/PwmTest* service is called by using the following message,

```
int32 esc
int32 front
int32 rear
int32 pwm4
int32 pwm5
---
bool result,
```

on the *pwm_change* service name where the messages above and below the `---` are the request and response. The request values are in a range of $[-24000, 24000]$ similar to the *txt_driver/Pwm* message. After the request is processed, the result is returned.

The switched power service sets the status of the two switches on the power distribution board. The following message is the *txt_driver/SwitchedPwr* service:

```
uint8 SOURCE_ATX = 0
uint8 SOURCE_ESC = 1

uint8 source
bool on
---
bool result.
```

To command one of the switches, the *switched_pwr* service is called with the source and on state specified. The source can either be the ATX power supply or the ESC as indicated by the service constants. The on/off state is specified by the on variable.

The ROS driver allows the TXT-1 to easily be controlled from user programs. Appendix B.4 shows a C++ source template to use the TXT-1 ROS driver. All of the TXT-1 ROS services and topics are used in this C++ file. This can be used as a starting point for other applications to start programming with the TXT-1 robot. Instructions are also provided to start the example project, compile it, and run the TXT-1 driver and the example in Appendix B.3. Appendix B.1 shows how to get the source code for the lower level controller and the TXT-1 ROS driver. Appendix B.2 gives instructions to compile the low-level source code and program the development board.

3.5 Communication Protocol

A communication protocol between the low and high-level controllers is created in order to control the robot and access its low-level functions. The RS-232 interface is used to communicate with the low-level control board. However, the Mac Mini does not have an RS-232 port, therefore, a USB to RS-232 converter is used. The

communication protocol uses a baud rate of 56700, 8 data bits, 1 start bit, 1 stop bit, and no parity, or 8N1. The communication protocol is a modified version of the MobileRobots ARCOS protocol [28]. The communication protocol used here is shown in Figure 3.10. The two header bytes signal the start of a message so that the high

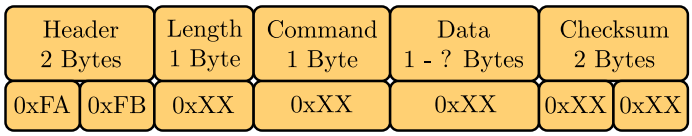


Figure 3.10: The TXT-1 communication protocol.

and low-level controllers can be synchronized when passing messages. The length byte is the variable length of the data portion of the packet. The maximum length of the whole packet is defined to be 255 bytes. Therefore, the maximum number of data bytes is limited at 249 bytes because of the 2 byte header and checksum, length byte, and command byte. The command byte indicates the type of message and the data's purpose. The different types of messages, their command bytes, and data lengths are shown in Table 3.3. The following list describes the different types of messages

Message Type	Data Length	Command Byte
Velocity	4	0x67
Encoder Odometry	20	0x68
Combined Odometry	20	0x69
Battery Levels	4	0x6A
PWM Test	20	0x6C
Power Control	2	0x6E
PWM Command	8	0x6F
PID Set	236	0x70

Table 3.3: The communication protocol message types and their lengths and command bytes.

and their purposes. All of the messages are one-way and do not have a response,

because they do not need to verify if they were received. The only exception is the PID Set command which requires a response to start the low-level controller.

Velocity The velocity message is sent from the ROS driver to the low-level board to set the velocities of the controller. The message consists of 2 16-bit integers, the first for the linear velocity and the second for the angular velocity. The values are in mm/s and mrad/s, respectively. The velocity message is sent continuously by the ROS driver to the microcontroller board. As a safety feature, the controller board will stop controlling the velocities and set its PWM outputs to 0 if it does not receive a velocity message every second. This will stop the vehicle if the ROS connection is lost.

Encoder Odometry The encoder odometry message is sent from the low-level board to the ROS driver. This message consists of 5 32-bit integers representing the x position (mm), y position (mm), orientation θ (mrad), linear velocity v (mm/s), and the angular velocity ω (mrad/s) calculated from the wheel encoders in the local frame. All of the data values are then converted by the ROS driver to m, m/s, rad, and rad/s.

Combined Odometry The combined odometry message is the same format as the encoder odometry message except that it is sent from the ROS driver to the microcontroller board. If this message is received by the low-level controller, the controller uses this message's velocity information to control the vehicle's velocities, v and ω. This is useful in situations when more accurate velocity information from a Kalman filter is available by fusing sensor data. For example, the IMU, GPS, and encoder odometry can be combined through a Kalman filter to obtain more accurate velocities and odometry information. Then the velocity controller would not be directly affected by encoder errors of wheel slip.

Battery Levels The battery levels message consists of 2 16-bit integers, in millivolts, sent from the controller to ROS.

PWM Test The PWM test message is mainly used for ensuring that the servos and the motor controller work correctly. To use this message, the directional button must be pressed in order to change the controller's mode to manual mode, which disables the controller algorithm. Then the message can be sent from ROS to the development board in order to control the 5 PWM outputs. The message consists of 5 32-bit integers from a range of $[-24000,\ 24000]$. This message can also be used to make a velocity controller for the robot in ROS instead of it being on the development board.

Power Control One of the features of the power distribution board which is described in Section 3.6 is its ability to turn off power to certain parts of the system to conserve energy. This message is sent from ROS to the controller board to set the state of these switches: the ATX power supply switch and the ESC switch. The message consists of two bytes: the switch byte and the switch state byte. The switch byte can be 0x00 for the ATX power supply and 0x01 for the ESC. A switch state byte of 0x00 or 0x01 turns off or on the selected power supply. This message can be used in critical, low-power situations.

PWM Command The PWM command controls the 2 extra PWM outputs which can be used to control pan-tilt camera servos or other items needing servo signals. The PWM command message consists of 2 32-bit integers in a range of $[-24000,\ 24000]$. Unlike the PWM test message, this message can be used during normal operation of the controller algorithm because the controller does not use these servo outputs.

PID Set The PID set message sets the gains of the velocity controller. Since the vehicle's velocities are coupled by $v = \omega R$, gain scheduling is used for the

angular velocity controller. The controller algorithms are discussed more in Chapter 4. This message is composed of the linear velocity and angular velocity PID gains as shown in Figure 3.11. The gains are sent from the ROS computer to the development board as 3 32-bit integers with each gain multiplied by 1,000, so that a decimal number can be sent serially. This message is required from ROS before velocities are sent to the controller board, because the PID gains are initially set to 0 when the controller board powers up. Then the controller board sends a response to ROS if it has received the PID gains. On the startup of the ROS driver, ROS sends this message until it receives a response. The three linear PID gains, proportional (K_P), integral (K_I), and derivative (K_D) are sent as the first 12 bytes of the data packet. Next, up to 14 sets of angular velocity gains are sent in the data packet. Each set consists of the maximum linear velocity as a 32-bit integer in mm/s and 3 32-bit PID gains. The sets of gains should be sent in increasing order according to their linear velocity. Also only positive linear velocities are used since the gains are the same for negative velocities.

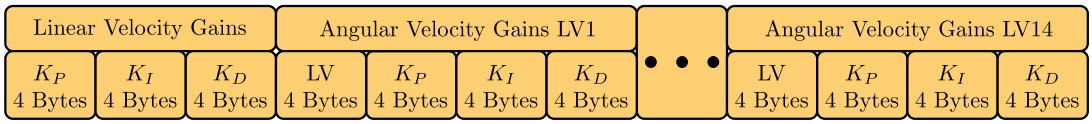


Figure 3.11: PID set message format.

The checksum is used to check for accidental data errors, which could cause the system to behave improperly. For example, if a high order bit is accidentally set, then the value read could be significantly higher than intended. In the velocity message, this could cause the vehicle to follow a velocity higher than commanded, potentially causing the vehicle to crash if it is in a critical area. The checksum adds all of the byte pairs from the command message to the last data byte. If there is an

odd number of bytes the last byte is exclusive OR'ed to the low-order byte of the checksum. Then the two checksum bytes are put into the message in reverse order. An example message is shown in Table 3.4. This is a velocity message with $v = 0.5$ m/s and $\omega = 0.75$ rad/s.

Header		Data Length	Command Byte	Linear Velocity		Angular Velocity		Checksum	
0xFA	0xFB	0x04	0x67	0x01	0xF4	0x02	0xEE	0x5B	0xED

Table 3.4: Example velocity message from the TXT-1 communication protocol.

3.6 Power Distribution System

The power distribution system is an essential component of the TXT-1 platform. Without it, many of the components of the platform can not be powered, resulting in a limited robot, lacking autonomous capabilities. This section describes the requirements and components of the power distribution system. The battery life results of the platform are also discussed.

3.6.1 Requirements

All of the electrical components of the robot platform require power. The microcontroller board, on-board computer, sensors, and the motors and servos all require power. However, most of these components need power in various forms, such as DC and AC current, different voltage and current levels. For example, the Mac Mini requires 120 VAC and consumes a maximum of 85 W. The sensors that are used on the robot typically use 5 VDC or 12 VDC, and use less than 1 A. The ESC has a maximum input of 12 VDC and the steering servos usually run on 4.8 VDC to

6 VDC. These motor elements use a varying amount of current, depending on the load of the motors. The microcontroller board has a voltage regulator and has an acceptable input voltage of up to 20 VDC, depending on the load being used. All of these parts need their respective power supplies to operate.

Another requirement for the robot is to power everything with one battery. This is desired so the remaining run time of the robot will not be limited by a single battery. In the previous TXT-1 design, two batteries were used: an 11.1 V Lithium-Ion battery pack was used to power the PC-104 computer and a 7.2 NiMH battery was used to power the robot motors. This design worked; however, one battery would run out before the other, which made determining the battery health complicated. Therefore, in this design, a single battery type is used for the TXT-1. The chosen hardware is discussed in the following section.

3.6.2 Hardware

This section describes the hardware that was chosen for the power distribution system to meet the power requirements of the robot. Figure 3.12 shows the power distribution board and its connections and features. This design allows the power distribution board to be the center of the delivery of power. The descriptions of the major components of the power system follow.

Battery To provide the robot with enough power to complete longer missions and tasks, an appropriate battery is selected. A nominal voltage of 11.1 V is selected because it is in the voltage range of the ESC. The Lithium-Ion Polymer (LiPo) type of battery is chosen because of its energy density. It is very lightweight while providing the necessary amp-hours required to run the vehicle for long periods of time. The MaxAmps 6500 mAh 11.1 V 3-cell LiPo battery is used to provide power to the vehicle for longer experiments. The LiPo batteries can

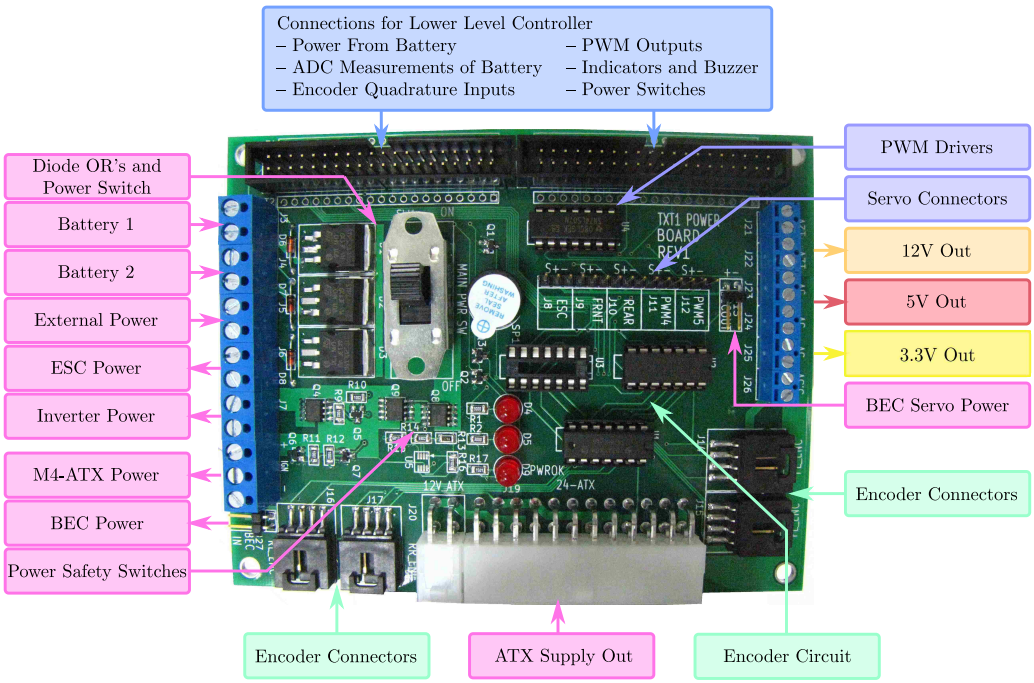


Figure 3.12: The TXT-1 power distribution system.

be dangerous if the voltage levels during charge or discharge exceed 4.2 V/cell or 3.0 V/cell. Therefore, protection circuitry is used on the power distribution board. The power distribution board is designed to handle up to 2 batteries. However, if another battery is necessary, it can be connected to the external power input.

Inverter In order to power the Mac Mini from a battery, an inverter is selected. The Mac Mini has an 85W internal AC-DC power supply, so the AIMS 180W Pure Sine Inverter is chosen. The inverter is lightweight, small, and can power the Mac Mini. It also features a pure sine wave output, which helps protect sensitive electronics like the Mac Mini, and has an efficiency greater than 90%. However, the voltage input of the inverter is limited to $10.5 - 15$V and the lowest voltage of the battery can be 9V. Therefore, an additional power supply

is chosen to utilize the full range of the battery limits.

Power Suppy The power supply selected is the M4-ATX from Mini-box. This is a 250W ATX DC-DC computer power supply with a 6-30 VDC input capable of supplying 15A @ 3.3VDC, 15A @ 5VDC. and 12A @ 12VDC. The power supply is capable of supplying enough power to the inverter at 12V to power the Mac Mini. The power supply is also used to power sensors that require external power.

ESC and BEC The ESC and BEC (Battery Eliminator Circuit) supply the power to the motors. The ESC controls the speed of the main drive motors to the wheels. It has a maximum input voltage of 12 VDC. The BEC eliminates the need for an additional battery to power the steering servos. The Castle Creations BEC, a buck switching regulator with an output of 5V @ 10A, is used. It also has an input voltage range of 5-25 VDC. The ESC also has a 3A BEC incorporated into it. However during testing, the servos would draw too much current, causing the ESC to reset and result in jerky motion of the vehicle. Thus, the separate BEC is used.

Power Distribution Board The power distribution board is the only custom electronic hardware on the TXT-1. It is made in order to distribute power reliably and flexibly, and route connections to the low-level control board. The schematic, PCB layout, and parts list of the distribution board is shown in Appendix A. The board is designed using KiCad, a free, open-source PCB development suite. The board uses inputs for two batteries and an external power supply which are diode OR'ed to provide hot-swappable power to the robot. The battery voltages are also sensed with the ADC of the microcontroller. Then the battery inputs are routed through a shutoff switch and a cutoff circuit. The cutoff circuit switches off power if the input voltages are not in the 9.25 V - 14 V range. The power is routed to the power supply,

low-level control board, ESC, and BEC. The ATX power supply and the ESC can be switched on and off from the microcontroller board if the robot is to be immobile and power is to be conserved. Then the output of the ATX power supply is then input into the power distribution board to provide 3.3V, 5V, and 12V. These voltages are used with terminal blocks to supply power to sensors requiring external power. The 12V from the power supply also powers the inverter, which powers the Mac Mini. The board has four connections for wheel encoders and the circuit in Figure 3.14 for the microcontroller to read the encoder signals. The encoders are powered from the ATX power supply's 5 V output. The board has the 5V output from the BEC and connectors to power, connect and control the servos and drive motors. The power distribution board has a warning buzzer and serial transmit and receive LEDs. The buzzer indicates to the user that either the TXT-1 lost communication with the Mac Mini or the batteries are low. The LEDs indicate whether the TXT-1 low-level board is receiving or transmitting messages.

3.6.3 Battery Life Results

After robot assembly and controller development, the battery is tested for its run time. For this test, the battery is completely charged and the robot is given waypoints in the shape of a rectangle to navigate until the battery runs out of power. All of the sensors are mounted to give the robot a full weight load, and all of the sensors are also powered to give the batteries a full electrical load. The electrical load should be higher if more processing power is used on the Mac Mini; however, this experiment does not test the battery life with different computational loads. The battery voltage is sensed by the ADC of the microcontroller board and is sent to ROS over the communication protocol developed. The voltage over time was then plotted as shown in Figure 3.13. It can be seen that one battery can run the robot

for up to 50 minutes. For safety, 46 minutes is used as the maximum for this data. Since the data is fairly linear from 0-46 minutes, a line is fit to the data. The line is plotted in the figure. This equation can then be used to determine the remaining life of the battery for algorithms with power constraints. This method of determining battery life works; however, future tests should be completed in order to determine if the characteristics of the battery change after several charges and discharges.

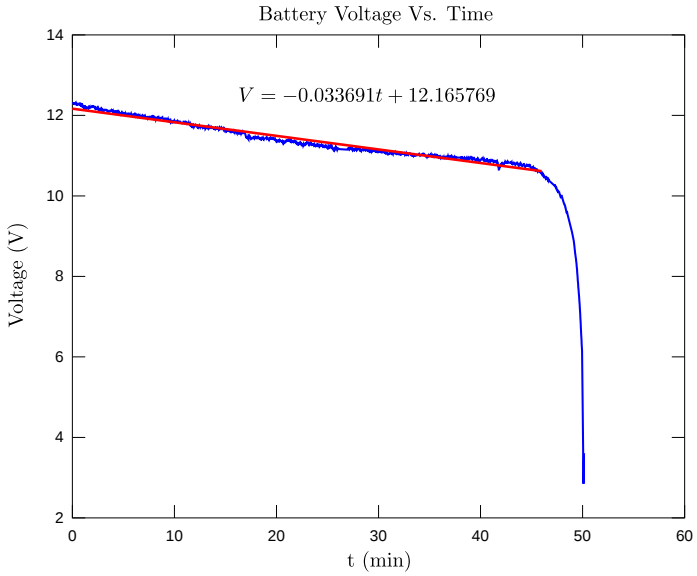


Figure 3.13: A plot of the battery voltage vs. time.

3.7 Sensor Suite

In order for the robot to be autonomous, it needs to be able to sense its surroundings and position in the world. This is done using a wide range of sensors. The sensors used on the TXT-1 platform are described in the following section.

3.7.1 Encoders

The wheel encoders are used to sense the wheel speeds of the vehicle and then calculate the position, orientation, and velocities as shown in Section 2.3. The encoders used are US Digital's E7P-400-236-S-H-D-B. These encoders have a resolution of 400 counts per revolution (CPR) and use a quadrature signal. The quadrature signal comprises of two square waves which are out of phase by $\pm 90°$. The sign of the phase of the signals indicates the direction of the wheel rotation, and the frequency of the square waves indicates the wheel speed. The encoders are connected to a simple circuit on the power distribution board that simplifies the velocity calculations on the controller board as shown in Figure 3.14. The circuit XOR's the two signals

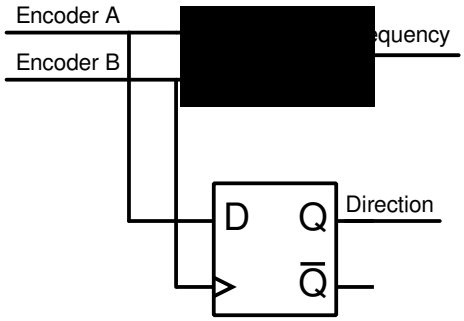


Figure 3.14: The encoder circuit.

of the quadrature signal and outputs a doubled frequency and uses the flip-flop to determine the direction of the wheel rotation. This frees many resources on the controller board so it only has to count one frequency signal per wheel instead of two. Additionally this circuit frees an additional interrupt from occurring when the direction needs to be checked. In order to calculate the velocity of the wheels, the tire circumference is measured. The tire was measured to have a circumference of 512.5 mm. To calculate the velocity of the wheel, we use

$$v_W = \frac{n_T c_W}{CPR\Delta t}, \tag{3.1}$$

where v_W is the velocity of the wheel, n_T is the number of ticks that are counted for the interval, Δt, c_W is the circumference of the wheel, and CPR is the count of ticks from the encoder per revolution of the wheel. Once the velocities of the left and right wheels are calculated, the position, orientation, and velocities of the robot can be updated.

3.7.2 IMU

The IMU (Inertial Measurement Unit) used is the Microstrain 3DM-GX2. This IMU has a tri-axial accelerometer, tri-axial gyro, tri-axial magnetometer, temperature sensors, and a processor that fuses this data for an accurate orientation result. The IMU is connected over USB to the Mac Mini, and a ROS driver organizes the data into a standard message format consisting of an orientation quaternion and raw angular velocity and acceleration measurements. The IMU can be used for gyro-enhanced odometry because the orientation calculation from the encoders can be prone to errors. By using the IMU orientation, the odometry results can be better. Also the IMU can be fused with other sensors in a Kalman filter to calculate accurate estimates of the position and orientation of the robot in the world. Figure 3.15 shows the IMU and GPS mounted on the TXT-1.

3.7.3 GPS

The GPS is a Garmin GPS 18 5Hz which is capable of using 12 parallel channels and WAAS. The GPS is connected to the Mac Mini with an RS232 to USB converter and interfaces with ROS easily. There is a ROS package that uses GPSD to access the GPS information. GPSD is a service daemon that makes data from any NMEA 0183 emitting GPS's available on a TCP/IP port. This allows multiple programs to access the data from one GPS. ROS has a package that listens to the GPSD

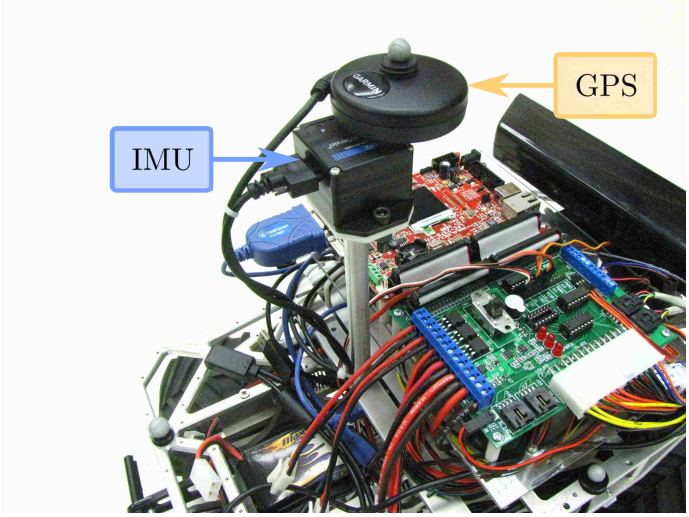


Figure 3.15: The mounting locations of the IMU and GPS.

port and publishes ROS GPS messages. Another package can listen to the ROS GPS messages and publish an odometry message in the UTM (Universal Transverse Mercator) coordinate system. The GPS can be used in a Kalman filter for position and velocity information, and it can be used for waypoint navigation. The GPS is a good absolute sensor.

3.7.4 Kinect

Vision sensors are necessary in order for the robot to accomplish complex tasks in a demanding 3-D environment. Vision sensors are used in robotics, traffic light sensors, military intelligence, industrial inspection, and many other fields. On the TXT-1, different vision sensors can be used, like the Microsoft Kinect, Point Grey Bumblebee2, and Unibrain Fire-i. The first is connected with USB and the last 2 are connected with Firewire to the Mac Mini. ROS has drivers for all of these cameras. The Kinect is an indoor 3-D sensor that utilizes a time of flight method. It provides

depth information so the robot can perceive the 3-D world and interact with it. The Kinect sensor uses the *openni_kinect* ROS driver to obtain both the RGB and depth images. The driver provides this data on the `/camera/rgb/points` topic as a point cloud and `/camera/rgb/image_color` topic as an image. An example visualization of the data is shown in Figure 3.16. This shows RVIZ, ROS's visualization tool, viewing the Kinect's 3-D point cloud along with the depth image on the left and the RGB image on the right. The mounting of the Kinect is shown in Figure 3.17. The

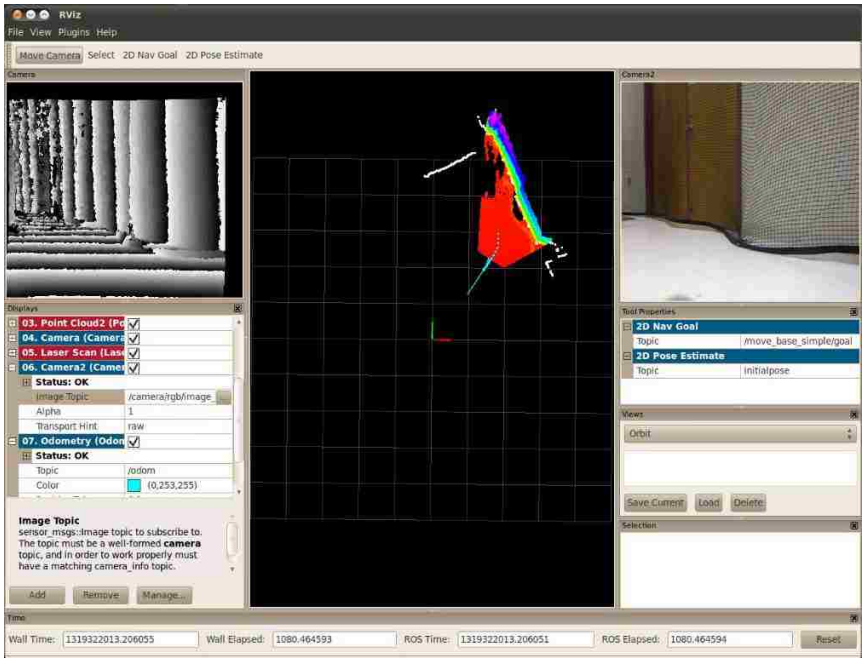


Figure 3.16: The data from the Microsoft Kinect and Hokuyo Laser Scanner. The upper left corner of the RVIZ window shows the depth map from the Kinect. The upper right corner shows the RGB image from the Kinect. The center shows the point cloud from the Kinect in color along with the Hokuyo laser scan data in white.

Bumblebee2 is a stereo camera that needs additional processing to get the depth information. The Fire-i camera is a single camera that can be used with some vision processing algorithms.

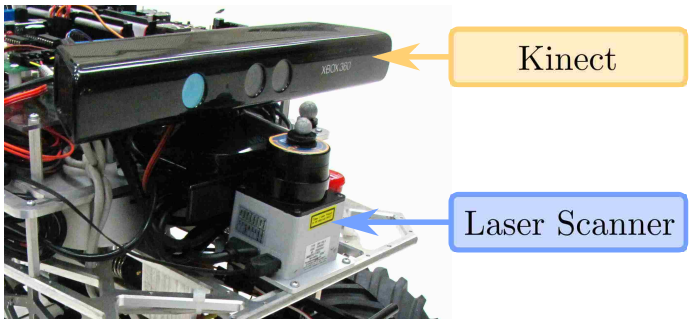


Figure 3.17: The mounting of the Kinect and the laser scanner.

3.7.5 Laser Scanner

The laser scanner used is a Hokuyo URG-04LX and can sense the distances to objects on the plane of the laser scan. The laser scanner is a 2-D sensor that works by spinning a laser and reading the sensed distance to the reflection at each angle increment. ROS has a Hokuyo laser driver and the message is an array of distances at each angle increment. The sensor can sense a distance of 4 m at a field of view of 240 degrees. The mounting location on the TXT-1 is shown in Figure 3.17 and an example of the sensed data is shown in Figure 3.16. The data is shown as white points in the center view of the window.

3.7.6 Vicon Tracker

The Vicon Tracker system [29] is an indoor, multi-camera rigid-body tracking system. Reflective markers are attached in a unique pattern on the rigid body and the cameras calculate the 3-D position and orientation of the object. This system is used mainly as an indoor GPS sensor. In order to use the Vicon system, a ROS driver is created

using the VRPN[1]library [30]. The ROS driver publishes ROS pose messages of the tracked object. The Vicon system can also be used as a velocity sensor for the velocity controller by using the combined odometry message described earlier. To do this, a Kalman filter is used to differentiate the position information into velocity information. The results of using the Vicon velocity are discussed in the next chapter.

[1]The VRPN Library was used for this experiment, which was developed by the CISMM project at the University of North Carolina at Chapel Hill, supported by NIH/NCRR and NIH/NIBIB award #2P41EB002025.

Chapter 4

Robot Controllers

In order to complete navigation tasks, an accurate, stable controller for the robot's linear and angular velocities is necessary. Without the velocity controller, smooth control of the robot would not be possible. The robot would control itself using bit-bang techniques, causing abrupt and rough behaviors. The lower level controller's main task is to control the vehicle's velocities. For the velocity controllers, we implemented Proportional-Integral-Derivative (PID) controllers to close the loop of the system. Since the linear and angular velocity is coupled by $v = \omega R$, the angular velocity controller uses gain scheduling. The next sections discuss the PID controllers used, both the linear and angular velocity controllers, tuning the controllers, and the performance results of each.

4.1 PID Control

The PID controller is a generic feedback control loop algorithm for controlling processes. The controller calculates the error between the process output and the desired setpoint and tries to minimize the error by adjusting the outputs to the process. It

works fairly well when the transfer function of the process is not known. A PID control loop is shown in Figure 4.1. As seen from the figure, the PID controller output

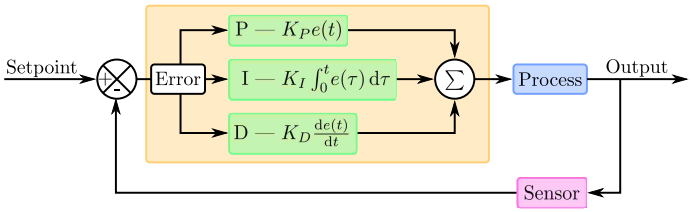


Figure 4.1: PID control loop.

to the process is

$$u(t) = K_P\, e(t) + K_I \int_0^t e(\tau)\,\mathrm{d}\tau + K_D \frac{\mathrm{d}}{\mathrm{d}t} e(t) + u_0, \tag{4.1}$$

where

$$e(t) = SP - PO, \tag{4.2}$$

and K_P is the proportional gain, K_I is the integral gain, K_D is the derivative gain, SP is the setpoint, PO is the process output, and u_0 is the base level control signal.

In order to implement the controller on a digital system, the continuous time domain equation needs to be discretized. By substituting differences and sums for derivatives and integrals, we obtain:

$$u(k) = K_P\, e(k) + K_I \Delta t \sum_{i=0}^{k} e(i) + \frac{K_D}{\Delta t}(e(k) - e(k-1)) + u_0, \tag{4.3}$$

where Δt is the sampling period. Equation 4.3 is the discrete positional form of the PID equation. It can be implemented on a digital system; however, u_0, the base level control output, needs to be known. This is overcome by using the velocity form of the PID equation, which is obtained by calculating

$$\Delta u = u(k) - u(k-1). \tag{4.4}$$

This is solved by substituting Equation 4.3 at time steps k and $k-1$ to get

$$\Delta u = K_P[e(k) - e(k-1)] + K_I \Delta t\, e(k) + \frac{K_D}{\Delta t}[e(k) - 2e(k-1) + e(k-2)]. \quad (4.5)$$

Then $u(k)$ is easily found by

$$u(k) = u(k-1) + \Delta u \quad (4.6)$$

to get a control signal of

$$u(k) = u(k-1) + K_P[e(k) - e(k-1)] + K_I \Delta t\, e(k) + \frac{K_D}{\Delta t}[e(k) - 2e(k-1) + e(k-2)]. \quad (4.7)$$

Equation 4.7 shows the PID update equation without u_0. This equation is used for both the linear and angular velocity controllers. The controllers are discussed in the next section.

4.2 Linear Velocity Controller

In this case, the processes that are controlled are the velocities of the TXT-1 robot. The setpoints of the PID loop are the desired linear and angular velocities and are set via the ROS communication link. The process outputs are the measured linear and angular velocities from the wheel encoders or the combined odometry message. The inputs to the processes are the motor controller and steering servo PWM outputs. Figure 4.2 shows the inputs and outputs of the TXT-1 velocity controller where v_c and θ_c are the velocity and steering angle PWM outputs. The sampling period, Δt, is 20 ms. The PID controller is actually composed of two PID controllers: the linear and angular velocity controllers. The implementation of the linear velocity controller is described in this section.

The linear velocity controller follows the commanded linear velocity from the ROS velocity message. The PWM output that goes to the motor controller has a

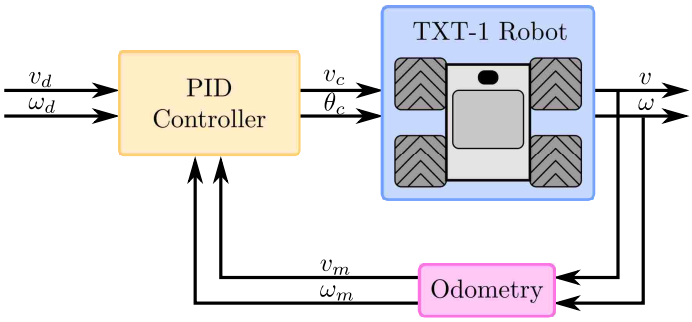


Figure 4.2: TXT-1 velocity control loop.

range of ±24000 with a value of 0 being a 1.5 ms pulse causing the motor to turn off and the vehicle to be stationary. A value of 24000 would spin the wheels forward at the maximum velocity and -24000 would spin the wheels backwards at the maximum velocity.

The linear PID controller is initially tuned by using a step input and having the steering servos positioned at 0 degrees. Then gains are iteratively chosen until the step response is quick with little to no overshoot. The linear PID controller is tuned by iteratively adjusting the PID gains. After the initial tuning, a constant velocity is selected and the steering is changed using a joystick to determine whether the coupling, $v = \omega R$, affected the controller substantially. The gains are then modified slightly and the controller is tested with constant velocity setpoints over the range of the allowable linear velocities. A range of ±1.5 m/s is used and the gains are again adjusted slightly. After tuning, the following gains are selected:

$$K_P = 11.0, \qquad K_I = 18.0, \qquad K_D = 0.250.$$

These gains produce a fast response and are found to overcome the coupling problem the best. However, the coupling of the linear and angular velocities affects the controller minimally. When the robot goes from driving straight to the maximum steering angle, the linear velocity droops by up to 10% of the setpoint. When turning

at the maximum steering angle and switching to driving straight, the linear velocity overshoots by up to 10% of the setpoint. This is acceptable because the linear velocity stabilizes after 1 second and it occurs when there are fast, large steps in the steering angle. These results are discussed more in Section 4.4. After the linear velocity controller is tuned, the angular velocity controller is tuned.

4.3 Angular Velocity Controller

The angular velocity controller follows the commanded angular velocity from the ROS velocity message. It controls the steering angle, ϕ, in order to follow the commanded angular velocity. The steering servos cause the wheels to be centered with a PWM output of 0, fully turned one direction with a PWM value of 24000, and fully turned the other direction with a PWM value of -24000. However, the TXT-1's steering mechanisms do not allow the maximum limits of the PWM outputs because its steering mechanism does not rotate as much as the servo. This causes the servos to draw excess current when the maximum PWM outputs are set. Therefore, the maximum PWM output for the steering servos is set at $\pm$18000.

The angular velocity controller assumes the linear velocity is constant for the sampling period, Δt. With this assumption, the steering angle, ϕ, can be used to control the angular velocity. It was observed through experiments that having one set of PID gains is not acceptable. For low velocities, the angular velocity gains need to be high. When used with higher velocities, the angular velocity step response has a large overshoot and oscillations. Therefore, gain scheduling was used to provide stable, fast response over the allowable range of linear velocities. The gain scheduling algorithm uses a maximum of 14 sets of gains for 14 linear velocity ranges. It uses the current measured linear velocity to select the set of gains to use for the control output calculation. The set of gains includes the maximum linear velocity that the

angular velocity is tuned. For example, the third and fourth gains could have a linear velocity maximum of 0.5 m/s and 0.7 m/s, respectively. The fourth gain would then have a range of $0.5 < v \leq 0.7$ m/s. The gains need to be provided for positive and negative linear velocities.

The angular velocity controller is tuned by commanding a constant linear velocity and then commanding step input angular velocities with a joystick. First, the commanded linear velocity is set small, and then the angular velocity PID gains are adjusted until the step response is acceptable. Once the gains are set for the first linear velocity, the commanded linear velocity is increased until the response starts to become poor. Then the gains are tuned and the process is repeated until the maximum linear velocity is attained. After all the gains are found, they are adjusted until an appropriate response is found. Table 4.1 shows the resulting angular velocity gains.

$v_{\max}$ (m/s)	K_P	K_I	K_D
-1.35	1.0	25.0	0.005
-1.15	1.0	25.0	0.005
-0.95	1.0	25.0	0.005
-0.75	1.0	35.0	0.005
-0.55	1.0	40.0	0.005
-0.35	1.0	50.0	0.005
-0.15	1.0	30.0	0.005
0.15	6.0	200.0	0.005
0.35	6.0	170.0	0.005
0.55	4.0	130.0	0.005
0.75	3.0	80.0	0.005
0.95	4.0	60.0	0.005
1.15	4.0	30.0	0.005
1.35	4.0	20.0	0.005

Table 4.1: Angular velocity gains.

4.4 Results

Multiple tests are performed to verify the controllers' performance. The first test shows the linear velocity controller's response to step inputs of varying degrees. For this test, $\omega = 0$ rad/s and the linear velocity input is given a step input. Then the vehicle is stopped and rests for 3 seconds. Next a negative step of the same magnitude is commanded and the process is repeated. Then the magnitude of the step input is increased and the test is repeated until the maximum tested velocity is reached. The linear and angular velocity plots of the test are shown in Figure 4.3. The tests are done on a tile floor. During the test, the robot loses traction when testing the higher velocities. Also the encoder readings are fairly noisy.

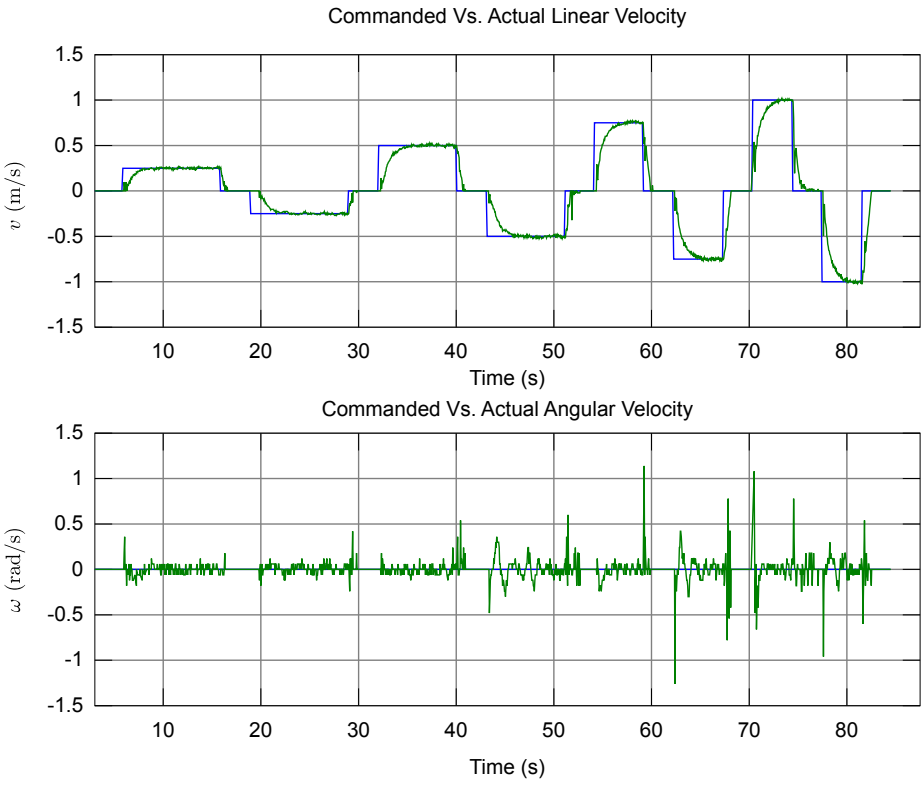


Figure 4.3: Linear velocity controller test.

The second test shows the angular velocity controller's response to step inputs of varying magnitudes. During the test, the linear velocity is held at a constant 0.5 m/s and the angular velocity is continuously stepped to increasing angular velocities. The test is repeated with increasing negative angular velocities. The results in Figure 4.4 show that the robot tracks the angular velocity without much droop in the linear velocity.

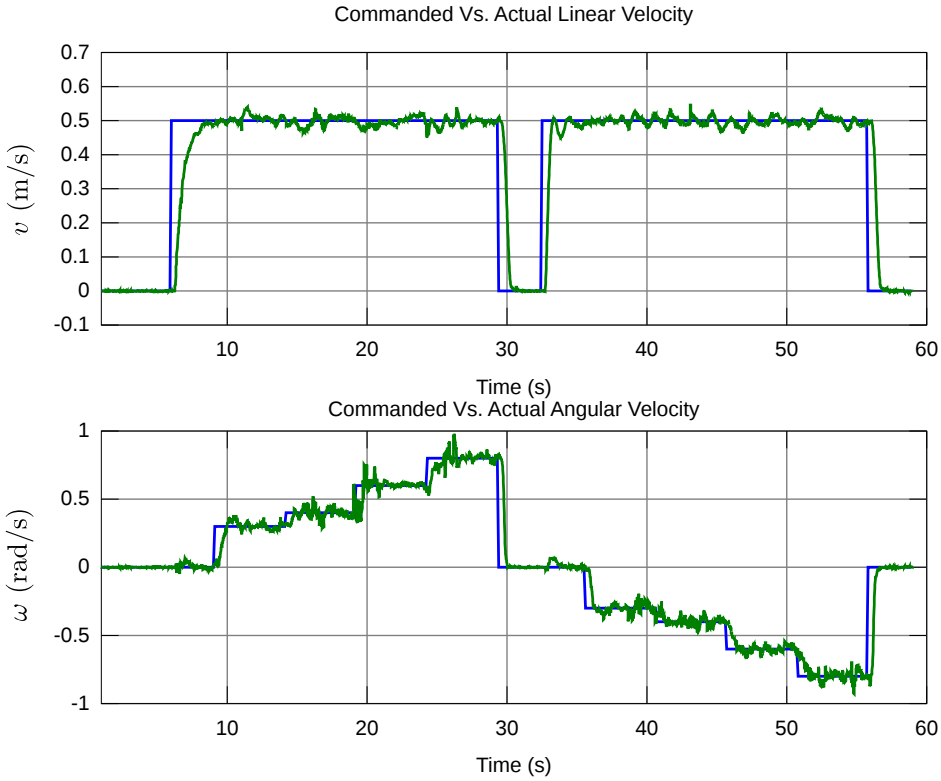


Figure 4.4: Angular velocity controller test.

The third test is to command the robot to follow constant linear and angular velocities to check if the path produced is circular and the circular paths overlap. Two of these tests are completed: one with $v = 0.5$ m/s and $\omega = 0.25$ rad/s and the other with $v = 0.5$ m/s and $\omega = 0.5$ rad/s. These tests are shown in Figures 4.5 and

4.6, respectively. In both tests, the circle is followed accurately. These three tests verify that the controller behaves properly.

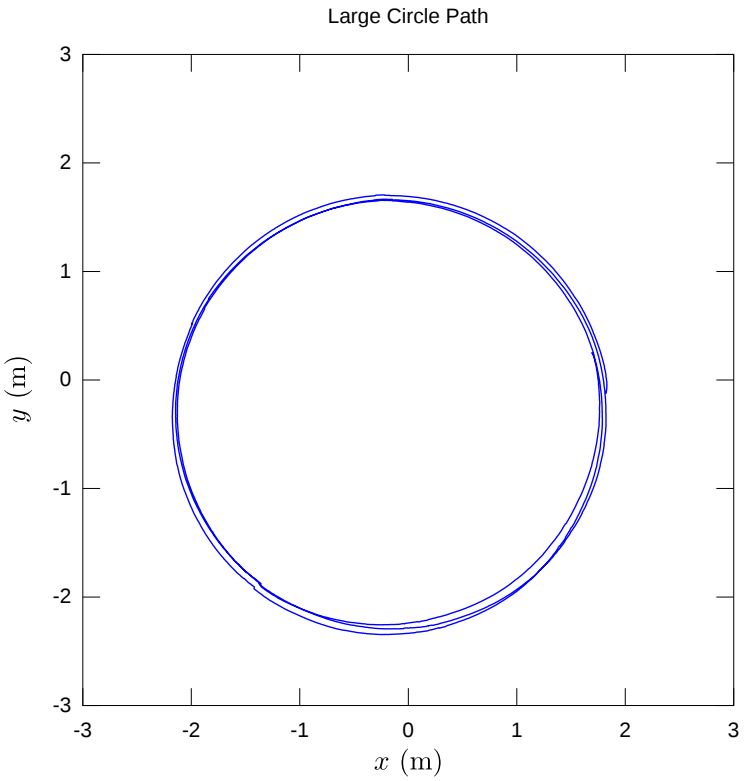


Figure 4.5: Path of robot following $v = 0.5$ and $\omega = 0.25$.

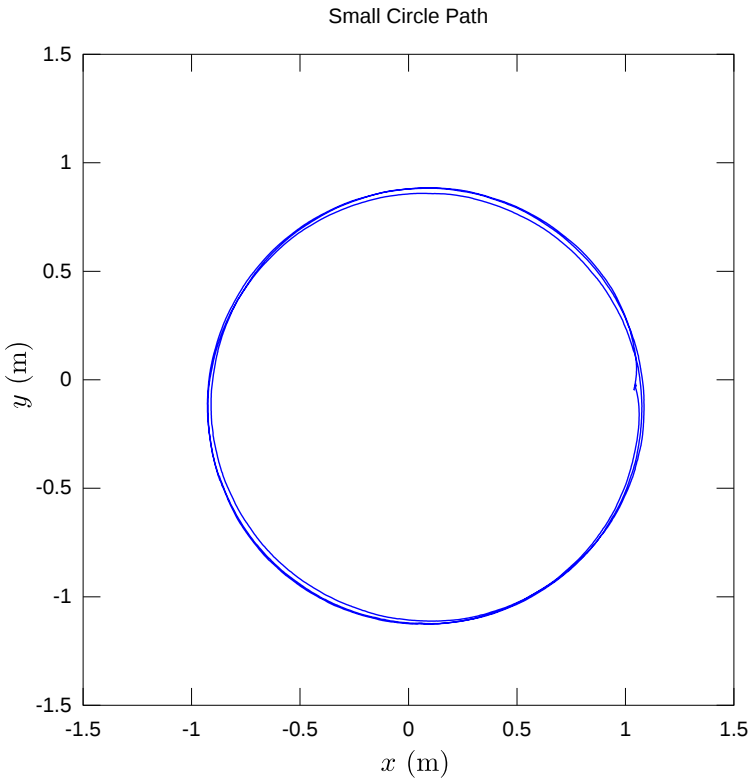


Figure 4.6: Path of robot following $v = 0.5$ and $\omega = 0.5$.

Chapter 5

A learning strategy for source tracking in unstructured environments

5.1 Introduction

Ideally, mobile robots should be able to perform navigation tasks autonomously without human intervention. Because of noise and a dynamically changing environment, this becomes a very difficult task. The navigation tasks are normally implemented with a type of feedback controller that processes the data from the sensors and makes decisions in order to complete the robot's navigation task. These controllers need to be properly tuned, using gains, in order to ensure the controllers are stable and behave optimally with respect to the most direct route to reach a goal. Because the sensors and actuators vary among different robots, these parameters are different. Additionally, over time, the sensors and actuators of robots can change due to wear and tear. Consequently, the navigation controller parameters need to be manually

fine-tuned to ensure optimal performance. For example, robot brush motors may experience wear and their response may change. As a result, the controller may have to be changed in order to accommodate the change in motor response. The manual tuning of parameters on each robot can be very costly and time consuming when the number of robots increases from a couple of robots to many thousands.

Learning, specifically reinforcement learning (RL), can be used to overcome the problem of having to manually fine-tune and update navigation controllers. The robot can learn through reinforcement to perform various navigation problems, thus eliminating the need for the time-consuming task of fine-tuning navigation controllers. It can learn navigation behaviors based on positive and negative rewards from the reinforcement learning algorithm [31]. In this work, Q-Learning (QL), a RL algorithm, is used to have the robot learn to navigate towards a light source. This problem is implemented both in simulation and on real hardware. The organization of this chapter is as follows: Section 5.2 gives a brief introduction to Q-Learning and its implementation, Section 5.3 explains how the problem is organized, Sections 5.4 and 5.5 discuss the results of both the simulation and the implementation in hardware.

5.2 Reinforcement Learning

The field of machine learning is broken up into three categories: supervised, unsupervised, and reinforcement learning. In a few words, reinforcement learning is when an agent learns a behavior by trial-and-error interactions with a dynamic environment and receives a delayed reward. Typically, the reinforcement learning problem is posed as follows: given a discrete set of states, $\{S\}$, a discrete set of actions, $\{A\}$, and reward signal, R, find the optimal policy, π, that maximizes the future reward. The optimal policy is a function that maps states to actions where the agent completes

the task at hand correctly, determined by an expert observer. For example in the light-finding task, optimal means the robot takes the most direct route to navigate to the light source. This is unlike supervised learning where the correct action/output is given based on the states/inputs. In reinforcement learning, the agent only knows the previous and current states and the reward of how good the previous action was. From this information, the agent tries to learn the mapping of states to actions that maximizes the future reward [32, 31]. In this model, the agent takes action, a_t, at time, t, based on its state, s_t. Then the action produces a new state, s_{t+1}, from the environment, and a reward, r_{t+1}, is observed. According to its reinforcement learning algorithm, the agent then updates its policy, π, from s_t, s_{t+1}, *and* r_{t+1}. This is shown in Figure 5.1. In order to converge toward the optimal control policy

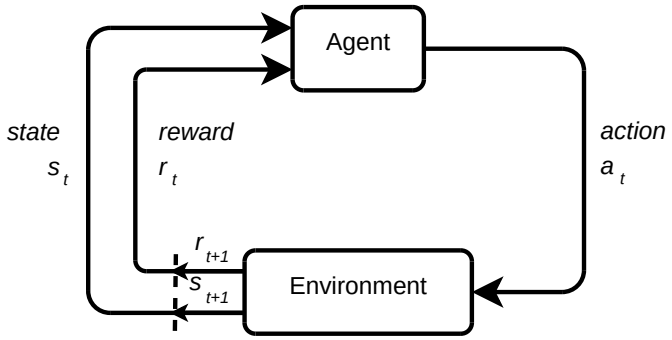


Figure 5.1: Reinforcement learning description.

quickly, the trade-off between exploration and exploitation needs to be understood. Exploration is when the agent selects random actions while learning, regardless of the current state, in order to visit every state-action pair. The agent might select an action that it thinks is suboptimal in order to find out that it might be good. Exploration allows the agent to explore the entire state-action value space to avoid the problem of learning local maxima. For example, in the light-finding case, if the agent does not visit a state-action pair while learning, it may learn to take a suboptimal action for a given state, like turning left when the light is to the right of the

agent. This results in the robot taking more time to navigate to the light goal.

In contrast to exploration, exploitation is the use of the agent's knowledge in selecting actions. The agent selects the action that it thinks will produce the maximum reward given the current state [33]. Care needs to be given when choosing the trade-off between exploration and exploitation. Using mostly exploration or exploitation can cause the agent to take a long time to learn the task or not even learn the task at all. Usually when the agent is first learning, exploration is used, and towards the end of the learning process, exploitation is used. The gradual transition from exploration to exploitation is decided by the algorithm developer. One approach to transition between the two strategies is the Boltzmann distribution, which is described in the next section. For the light source navigation task, reinforcement learning methods are used because they allow the state-action mapping to be learned through reinforcement in real-time.

5.2.1 Q-Learning

Q-Learning is a temporal difference reinforcement learning algorithm developed by Watkins in 1989 [34]. Temporal difference learning methods use changes in predictions over consecutive time steps in order refine the prediction of a quantity. They continuously update their estimates based on previously learned estimates, also known as bootstrapping. This is done until a terminal state is reached. Temporal difference methods use a combination of Monte Carlo methods and dynamic programming methods [31]. Like Monte Carlo methods, temporal difference methods do not need a model of the environment's dynamics. Like dynamic programming, temporal difference methods use bootstrapping.

Q-Learning is an off-policy, temporal difference method; off-policy meaning that it can separate exploration from control. Q-Learning learns a state-action value

function, Q, that directly approximates the optimal state-action value function, Q^*, independent of the policy being followed. Q^* is optimal in the sense that it is the best state-action value function for this problem as deemed by the expert user. This function informs the agent of the action with the greatest reward when the agent is in the current state. The QL states are the inputs to the system and the actions are the outputs from the system. It has been proven that Q converges to Q^* with probability of 1 if all state-action pairs are continuously updated [35]. The QL value function, $Q(s_t, a_t)$, is updated after the agent takes action, a_t, using

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right], \tag{5.1}$$

where s_t is the state at time, t, α is the learning rate, r_{t+1} is the reward after taking the action, a_t, γ is the discount factor, and s_{t+1} is the state after taking the action. The general QL algorithm is shown in Figure 5.2. Q-Learning uses a finite set of

```
Initialize Q(s, a) arbitrarily
while repetitions count < number repetitions do
  state = current state
  repeat
    Action = GetAction(State)
    Take Action, Observe Reward and Resulting
    State
    Update Q-Table with Update Equation
    State = Resulting State
  until State is in Goal State
end while
```

Figure 5.2: Q-Learning algorithm.

discrete states, $s_i \in S$, where $0 \leq i < N_A$, and actions, $a_i \in A, ,$ where $0 \leq i < N_S$. N_A and N_S are the number of discrete actions and states, respectively. The value function, referred to as the Q-Table in the rest of the chapter, is in tabular form. $Q(s_t, a_t)$ is the value of Q at state, s_t, and action, a_t, at time, t [31]. The Q-Table is stored in the agent's memory, which is typically either a computer's hard disk or

RAM. Memory can be a concern if the task to learn has many state and action pairs resulting in the system running out of memory. However, it is hard to learn tasks with many state-action pairs because the time it takes to learn these tasks is very large.

In the QL algorithm, action, a_t, is chosen based on the policy, π, and the current state, s_t. The policy, π, is chosen based on the exploitation/exploration trade-off discussed earlier. The selection of the policy is important in order to make sure that the system converges to the optimal solution. There are many approaches to choose a policy and three are discussed here. The first approach, the greedy policy, is a policy that exploits the value function. Basically, the agent chooses the action based on the maximum Q value over all possible actions for a given state, or $a_t = \max_a Q(s_t, a)$. This policy is usually used toward the end of the training and once the agent is trained. The second approach is a non-greedy policy where the agent chooses random actions at every time step, or $a_t = random(a \in A)$, where the actions have a uniform probability. Using this approach is usually good to apply at the beginning of training. Both of these policies can be used as stated; however, they can theoretically be used for the entire learning process and the Q values should converge to their optimal values regardless of the policy.

One of the more popular policies is the Boltzmann distribution [36] which is used to move from exploration to exploitation. The Boltzmann distribution can be written as

$$\Pr(a_i) = \frac{e^{Q(s,a_i)/T}}{\sum_{b \in A} e^{Q(s,b)/T}},\ a_i \in A, \tag{5.2}$$

$$\lim_{T \to \infty} \Pr(a_i) = \frac{1}{N_A}, \tag{5.3}$$

$$\lim_{T \to\ 0} \Pr(a_i) = \begin{cases} 1 & \text{if } a_i = \max_a Q(s_t, a), \\ 0 & \text{otherwise,} \end{cases} \tag{5.4}$$

where T is the temperature and N_A is the number of actions. Equations (5.3) and (5.4) show the limits of the Boltzmann distribution with respect to T.

The temperature parameter, T, in the Boltzmann distribution controls the randomness of selecting an action. When T is large, the probability of choosing an action is approximately equal to the probability of choosing any other action, favoring exploration. When T is small, the probability of an action increases with its Q value, favoring exploitation. In practice, the temperature parameter is decreased over time as $T = C\alpha^t$ where C is the temperature at time, $t = 0$, and α is the rate of decay. Using the Boltzmann distribution along with the decay function allows the agent to gradually transition from exploration of the state-action space in the beginning of the learning stage to exploitation of its acquired knowledge towards the end, thereby speeding up the learning procedure. Some examples of how the Boltzmann distribution affects the Q values are shown in Table 5.1. As the parameter T increases, the probability to choose any action is approximately equal. Once the temperature decreases, the probability favors the highest Q value. To use the Boltzmann distribution, first a random number is generated. Then the probability of each action corresponding to the current state, s_t, is calculated using Equation (5.2). Next, the cumulative distribution is generated from the calculated probabilities. The generated random number is compared to the cumulative sum and the action whose probability range contains the random number is chosen as the action to use.

Q-Value	0.2	0.5	1.0
T=0.1	3.33e-04	6.69e-03	9.92e-01
T=0.5	0.128	0.234	0.637
T=1	0.218	0.295	0.486
T=5	0.309	0.328	0.362

Table 5.1: The probabilities of taking actions with different temperatures and Q values.

The following description explains the Q-Learning algorithm in detail. Figure 5.3 is used in this explanation. First, the Q-Table is initialized. It can be initialized to a constant value, such as zero, or to some random numbers. Then the current state, s_t is observed and an action, a_t, is chosen using any of the policies discussed above. Figure 5.3 shows the use of the Boltzmann distribution. After the action is selected, it is then executed and the agent is delayed for a period of time, t_d. After the delay, the next state, s_{t+1}, and reward, r_{t+1}, are observed. Then the Q value, $Q(s_t, a_t)$, is updated according to the update equation, Equation (5.1). The update equation uses the next state, s_{t+1}, and finds the maximum Q value over all actions for that state. It then calculates the expected discounted reward, which is the reward added to the maximum Q value multiplied by the discount factor, γ. The expected discounted reward tells the agent how good the action that was taken actually was. If it was good, $Q(s_t, a_t)$, increases, and if it was bad, $Q(s_t, a_t)$ decreases. The discount factor, γ, adjusts how much future rewards affect the update. The expected total discounted reward can be written as:

$$\begin{aligned} R_t &= r_t + \gamma \max_a Q(s_{t+1}, a), \\ &= r_t + \gamma \left[r_{t+1} + \max_a Q(s_{t+2}, a) \right], \\ &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n r_{t+n} + ... \end{aligned} \tag{5.5}$$

The discount factor is on a range of $[0, 1]$. When it is small, the agent only weighs current rewards. When γ is large, future rewards are weighted more heavily. Next the expected discounted reward is multiplied by the learning rate, α. The learning rate is a number from $[0, 1]$ and controls how much of the new information is used to update the old information. When $\alpha = 0$, nothing is learned, and when $\alpha = 1$, full weight is given to the new information. The learning rate does not have to be constant while the agent is learning. The learning rate can start at 1 and decrease to 0 so that the agent uses the new information more heavily in the beginning of training

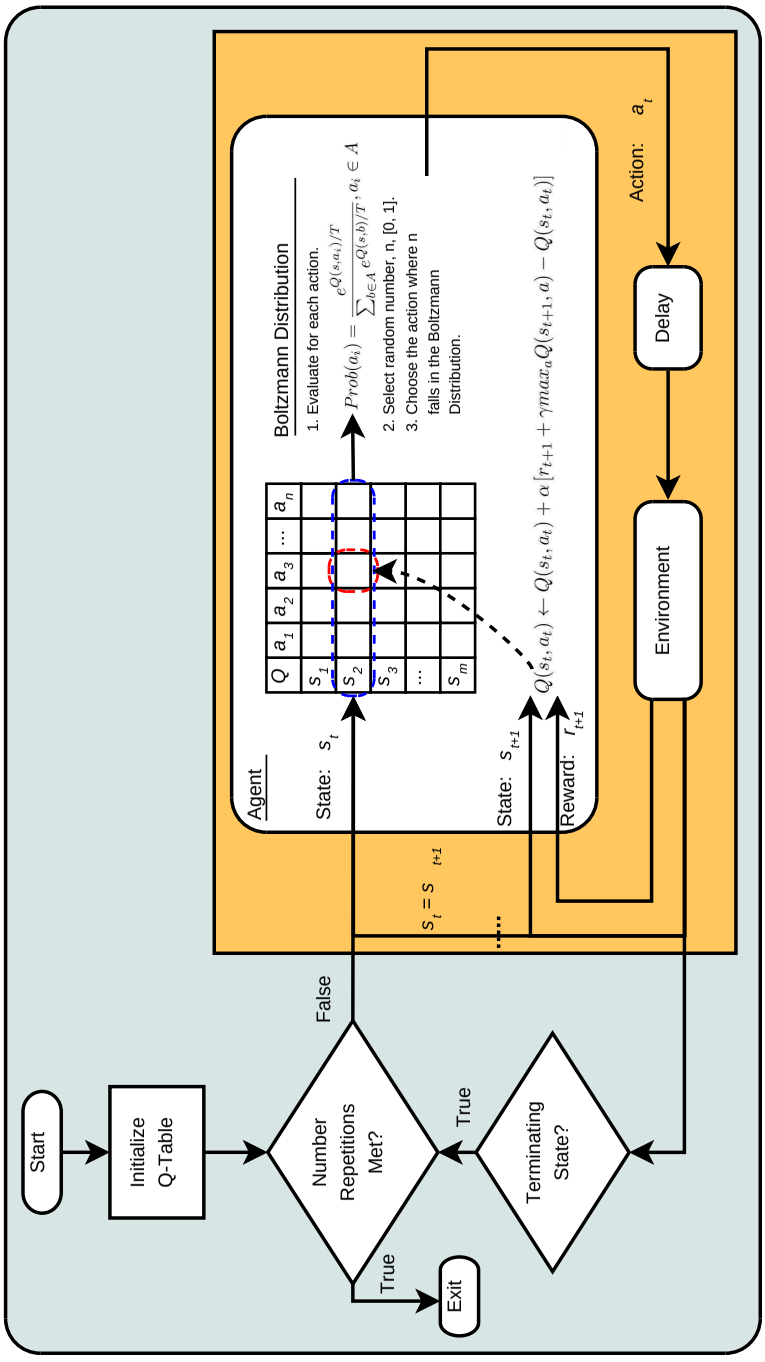


Figure 5.3: Q-Learning flowchart.

and uses less in the end. The learning rate can even be different for each state. If the agent visits a state more frequently, that state's learning rate can decrease faster than others.

After the update step, the next state is set as the current state. Then this algorithm is repeated until the agent enters the terminating state. Then the process is repeated for a number of repetitions or until the agent has learned the task. After the agent learns the task, the task can be accomplished by the agent by using the state-action mapping that was learned. After the task is learned, the agent normally uses a greedy approach by exploiting the learned mapping and does not update the Q-Table. However, the agent can be designed to continuously learn changes in the environment that may occur after the initial training. Q-Learning is a simple, popular reinforcement learning algorithm that trains an agent to learn a task from discrete states and actions and a reward signal.

5.3 Light-Finding Robot

In order for the robot to learn to navigate towards a light source, the problem needs to be formulated with more details. In the following sections, the robot model, defined actions, environmental states, reward function and the policy used are described in more detail.

The Robot Model

The TXT-1 prototype is used in this experiment. 5.4. Figure 5.4 shows the robot model and the position of the 4 light sensors in the four corners of the robot. The nonholonomic unicycle model is used for simulation. Limits on the angular velocities are used to make the unicycle model work for the car-like TXT-1. There are 4

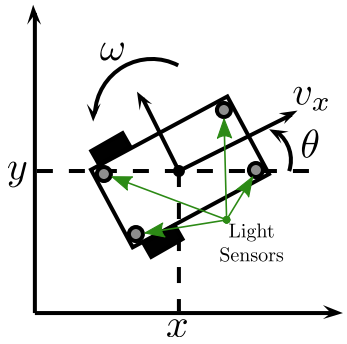


Figure 5.4: The mounting of the light sensors.

light sensors that measure the intensity of the light on the robot. The light sensors measure human perceptible light on the range of 1 lux to 1000 lux. The sensors are positioned in a rectangle pattern on the top of the robot. They are mounted at a distance of ± 0.2286 m in the X direction and ± 0.127 m in the Y direction from the center of the robot. The sensors are labeled front-right (FR), front-left (FL), rear-right (RR), and rear-left (RL). The sensor readings are not corrupted by noise in the simulation but are in the real experiment.

Robot Actions

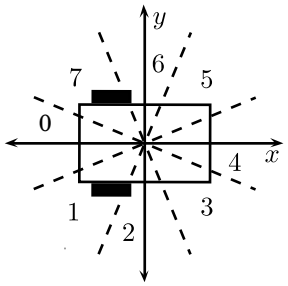
Since Q-Learning requires discrete actions, three actions are chosen: forward left, forward straight, and forward right. Only forward actions are chosen because it limits the number of state-actions pairs and the learning time. These actions with their associated linear and angular velocities are listed in Table 5.2. The forward velocity is maintained constant in order that the navigation algorithm can be applied to both differential drive and car-like robots. The selected angular velocities are the maximum achievable angular velocities corresponding to the selected linear velocity satisfying $v_x = \omega R$, where R is the minimum radius of curvature of the car-like robot used in the experiment.

Action	v_x (m/s)	ω (rad/s)
Forward Left	0.3	0.54
Forward Straight	0.3	0.0
Forward Right	0.3	-0.54

Table 5.2: Chosen robot action velocities.

Light Sensor States

From the mounted light sensors, the direction of the light is calculated. The magnitudes of the light readings are separated into their X and Y components and then added to compute the direction of the strongest intensity of the light source. Then the direction is discretized into 8 states as seen in Figure 5.5. In the figure, if the direction falls within the specified boundaries, then the direction is discretized according to the corresponding area. Therefore, the total number of states is 8 and the number of actions is 3, resulting in 24 state-action pairs. The small number of states permits the robot to learn quickly.



0. Rear Center
1. Rear Right
2. Center Right
3. Front Right
4. Front Center
5. Front Left
6. Center Left
7. Rear Left

Figure 5.5: Discrete light direction states.

Reward Function

The reward function uses the current and next light direction states to calculate the reward. Equations (5.6) - (5.8) show how the reward function is calculated.

$$s'_t = s_t - 4, \tag{5.6}$$

$$s'_{t+1} = s_{t+1} - 4, \tag{5.7}$$

$$r_{t+1} = \begin{cases} 1 & \text{if } s'_{t+1} = s'_t = 0 \text{ or } |s'_t| - |s'_{t+1}| > 0 \\ 0 & \text{if } s'_{t+1} = s'_t \neq 0 \neq -4 \\ -1 & \text{otherwise.} \end{cases} \tag{5.8}$$

First the current and next states are subtracted by 4. This results in the front center state equal to zero and the rear center state equal to -4 allowing for an easier calculation in the next step. Next the reward is given based on the conditions in Equation (5.8). These conditions give a reward that encourages the robot to navigate to the light source. The reward gives positive reinforcement when the heading error to the light source is reduced and when the light is continuously in front of the robot. Negative reinforcement is given when the heading error to the light source is increased or the light source is continuously behind the robot. A reward of 1 is given when the current and next states are the front center state, meaning that the light source is in front of the robot. This reward is also given to the robot when its direction state gets closer to the front center state, resulting in the robot's heading error decreasing. A reward of zero is observed by the robot when the light direction state does not change and the light is not directly in front of or behind the robot. A reward of -1 is given to the robot when the heading error increases or when the light source is behind the robot. This reward function produces a light finding behavior as seen in the following simulation and experiment.

Test Environment

The test environment consisted of a room, safety boundary, starting radius, light source, robot, and light terminating position. Figure 5.6 shows all of these in detail. First of all, the light source is positioned in the center of the test room at $(0, 0)$. At the beginning of each learning repetition, the robot started at a specified radius from the center of the room at a random angle from the positive x-axis of the room. This allows the robot to start in different locations of the room. At the starting location for each repetition, the robot is also set at a random heading towards the light to generalize the learning. During the learning repetitions, it is likely that the robot navigates and collides with one of the walls of the room. Therefore, a safety boundary is created to terminate the learning repetition when a collision is eminent. The learning repetition is also terminated after the robot passed through the terminating radius. The terminating radius is a boundary that represented the robot finishing its navigation task towards the light. After the learning repetition is terminated, the robot is moved to a new starting location and heading towards the light and the next learning repetition is started. Both of the terminating states needed global frame positioning information. This is provided through Stage, a robot simulator. In the hardware experiment, the Vicon motion capture system [29] is used. These are described in the simulation and experiment sections.

5.4 Simulation Results

The formulated Q-Learning navigation problem is then simulated and implemented on a real robot. Both the simulation and hardware experiments are implemented in ROS, which allowed that algorithms to be tested in simulation and then implemented in the experiment with a few code changes.

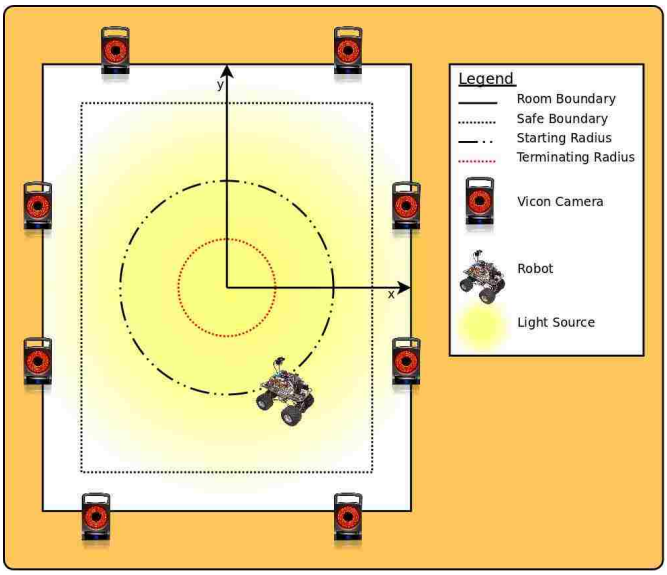


Figure 5.6: Learning environment.

The light-finding learning algorithm is implemented in the Stage simulation environment. Stage provides a 2-D simulated environment with a map, autonomous ground vehicles, and some sensors. In this simulation, the simulated light and the light sensors are artificially simulated because Stage does not have light sensors and light objects. The intensity is modeled as a point light source where the intensity of light decreases at a rate of the reciprocal of the distance squared. The intensity of the light at a position, $(x,\ y)$, assuming the light is at $(0,\ 0)$ is written as

$$I(x, y) = \frac{1}{(x^2 + y^2)^2}. \tag{5.9}$$

The map chosen is a 5 by 5 meter square, which allowed for ample room during the learning phase. During the experiment, a sample time of 2 Hz is used to update the Q-Table. This presented the robot with enough change in heading and position to make it possible to learn. The terminating state of the simulation is that the robot is 0.5 meters away from the center of the light source or if the robot drives out of the safety boundary. Stage provides global frame position information to check

whether one of the terminating states is reached. Once the termination state was reached, the robot is driven to a position at a fixed radius of 1.25 m from the light source at a random angle from the x-axis and a random heading to the light source. For the policy, a Boltzmann distribution is used. The temperature parameter, T, is decreased over time according to $T = C\alpha^t$. In the simulation, C is chosen to be 5 and α is chosen to be 0.65. This allows for the robot to utilize exploration at the beginning and exploitation towards the end of the learning process. A learning rate of $\alpha = 0.3$ and a discount factor of $\gamma = 0.5$ is used in simulation. When starting the learning phase, the Q-Table is initialized with random values from -0.1 to 0.1.

When testing with the simulator, it is noticed that the robot can reach the commanded velocities instantaneously. In order to make the dynamics of the simulated and real robots similar, a velocity ramp is introduced to the robot in the simulator. This causes some problems when learning. Because a significant control delay was introduced, the robot updates its Q-Table with the wrong data. The action is commanded; however, when the resulting state and reward are obtained, the action is not yet reached by the robot, resulting in the Q-Table being updated with incorrect data.

In order to compensate for the control delay, two approaches are used. In these approaches, the Q-Learning algorithm was slightly modified. In the first approach, the Q-Table is not updated until after the commanded action is reached. After the velocity action is selected and sent to the robot, the actual velocities are continuously read until they are within a certain threshold of the commanded velocity. Then the state, s_t, is read and the velocities of the selected action are held until the delay period is finished. After the delay period, the Q-Table is updated and the algorithm repeats as normal. By waiting for the velocities of the action to be reached, the robot can use the correct information to learn the task. This approach results in the robot learning the correct Q-Table; however, the trajectories are suboptimal because the

robot has to wait for the action to be reached which results in oscillations.

The second approach is to use the sensed action after each delay interval to update the Q-Table without waiting for the robot to reach the commanded action. In this approach, the selected action is sent to the robot. Then the delay time is waited and the action is sensed. The sensed action is used to update the Q-Table instead of the commanded action. The action is sensed by quantizing the sensed velocity into an appropriate action. This is accomplished by sensing the velocities with the wheel encoders and dividing the angular velocity space of ± 0.54 rad/s by the number of actions, 3. Then the sensed action is selected by comparing the sensed angular velocity with the angular velocity boundaries. The boundaries are shown in Figure 5.7 where each color represents a different action. Finally, the Q-Table is updated

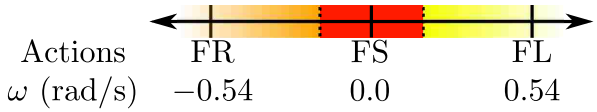


Figure 5.7: The sensed angular velocity boundaries.

with the sensed quantized action, current reward and state. This modification to the Q-Learning algorithm limits the effects of the control delay on learning and provides optimal paths when completing the light finding task. This approach in minimizing the effects of the control delay is used in the simulation and experiment.

In the simulation, the robot performed 31 repetitions of the task starting at a new position and orientation to the light source each time. In the beginning of the training, the robot both succeeded and failed in navigating towards the light source. During the final repetitions of the simulation, the robot succeeded in navigating towards the light source every time. A couple of the paths of the robot are shown in Figures 5.8 and 5.9. The light is shown as the yellow circle in the center of the map and the starting and ending positions are shown as red circles. It can be seen that the robot did not learn the task completely in Figure 5.8. Figure 5.9 shows the

final path of the robot. The path in this figure is almost the direct path to the light source from the starting location. The learning curve in Figure 5.10 shows that

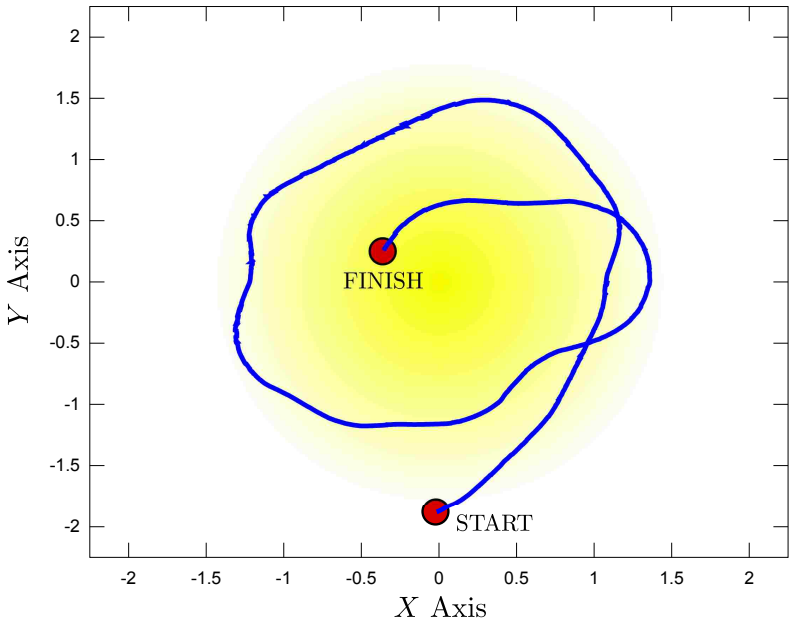


Figure 5.8: Simulated robot's path on the 10th learning trial. The path is not the shortest to the goal, because the agent has not learned all of the state-action pairs.

the time taken to complete the task decreases over the repetitions of the simulation, which implies that the robot learns to navigate towards the light. From Figure 5.10, it is seen that it takes the robot an average of 7.165s to complete the task after the first 7 repetitions. The shortest time that is possible to complete the task with a heading angle of 0 is 4.167s (1.25m at 0.3m/s). The simulation results were double this time, because during the simulation, the heading angle was chosen at random. Also to obtain good results for the learning curve, the room boundary was increased to 8m, because there is no distinction in times when using the small room boundary. The learning curve shows that the robot has learned to navigation towards the light source because the times decrease and converge to value.

At the end of the simulation, the Q-Table is visually checked to see if the values are

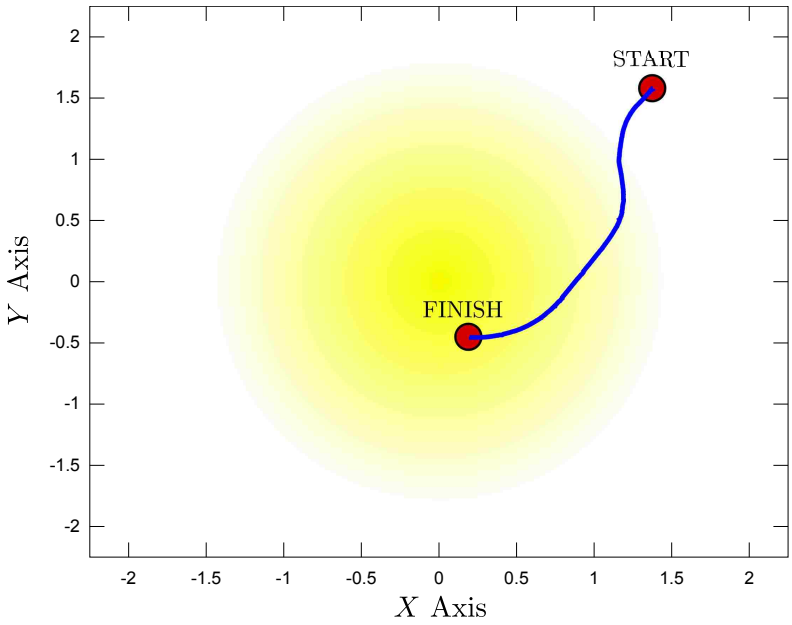


Figure 5.9: Simulated robot's path on the 31st learning trial. The path is shorter than the path in Figure 5.8 because the agent has learned all of the state-action pairs.

intuitively correct. For example, if the light is to the right of the robot, one expects the robot to turn right. Therefore, the action that is expected to be executed for a given state should have the highest Q value among all the action Q values for that state. After the simulation results are obtained, the algorithm is validated on hardware through a series of experiments.

5.5 Experimental Results

Implementing the learning algorithm in hardware produced more problems than in simulation. Noise is encountered in the light sensors and delays are experienced in the robot following the commanded velocities, both of which debilitated the learning process. However, results are obtained by filtering the light sensor signals and

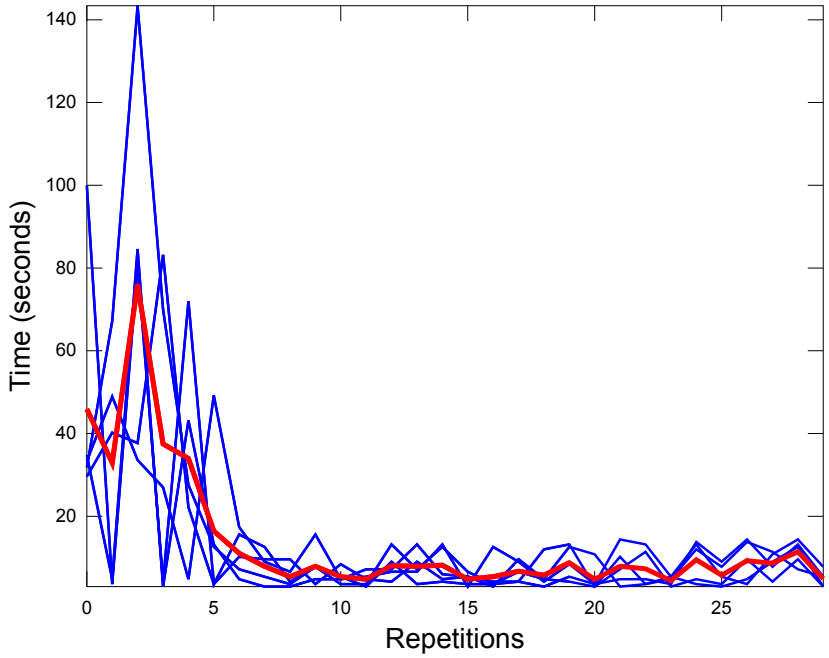


Figure 5.10: Average learning curve after five trials. The blue plots are the five independent trials and the bold red plot is the average of the five trials.

using the learning algorithm from the simulations to avoid the effects of control delays in learning. In this section, the hardware used, problems with the hardware implementation, and results are discussed.

5.5.1 Hardware

Figure 5.11 shows the hardware setup used in this experiment. Firstly, the TXT-1 robot is utilized for the experiment. The light sensors used are a part of the Phidgets Interface Kit [37]. The kit allows for 8 analog inputs, 8 digital inputs, and 8 digital outputs to be connected and read or written to via the USB port. Only 4 of the analog inputs are used with the light sensors. The light sensors measure human perceptible light on the range of 1 lux (moonlight) to 1000 lux (TV Studio Lighting). The Phidgets Interface Kit is connected to the on-board computer of the TXT-1.

Open-source ROS drivers are used to interface with both the TXT-1 robot and the Phidgets interface kit. To get an accurate position of the robot in order to stay

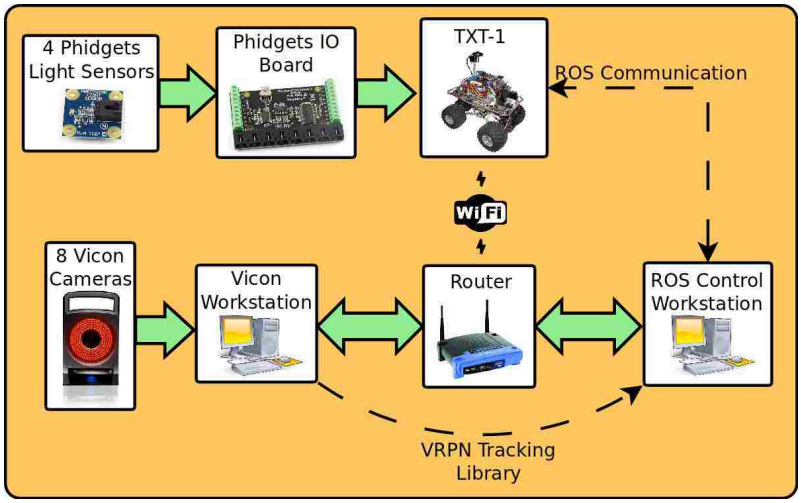


Figure 5.11: The experimental hardware setup diagram.

within the safety boundary of the map, the TXT-1's odometry is not used due to wheel slippage and drift. Instead the Vicon motion capture system is used. The Vicon motion capture system uses multiple cameras with reflective spheres to track rigid bodies. To use the system, the ROS driver developed previously was used. The light sensors and interface kit are also mounted on the robot with the spheres for the Vicon cameras. The mounting is shown in Figure 5.12. Finally, a 100 W light is mounted in the center of the room on the ceiling. Because of ROS's ability to abstract hardware drivers, most of the same code used for simulation was used in the experiment. The only change was the switching of the simulated light sensor nodes to the hardware light sensors and addition of a Vicon positioning node.

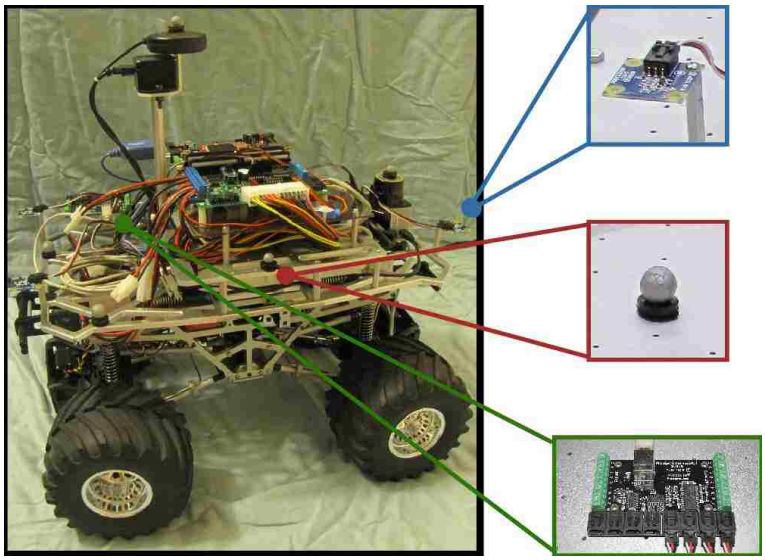


Figure 5.12: The robot and sensors.

5.5.2 Problems in Hardware Implementation

Noise in the light sensor readings affected the learning in the experiment greatly. For example, the calculated direction of the light source oscillates around the actual direction by about $\pm 45°$. When learning, the learner receives the wrong reward a majority of the time. Then the learner never learns anything because the noise causes the learner to unlearn the task once it began to learn it. Therefore each light sensor reading is filtered with the exponential moving average filter,

$$I_t = \alpha I_{t-1} + (1 - \alpha) I'_t, \tag{5.10}$$

where I'_t is the measured light intensity and I_t is the filtered light intensity at the current time. Because the light sensors are sampled every 16 ms, a value of $\alpha = 0.9$ is used to produce an accurate calculated direction without delay from the filter.

Another problem is the delay in the robot following the velocities of the commanded actions. This causes problems with learning; however, the effects of the

control delay are minimized by using the Q-Learning modification used in simulation of using the sensed actions for updating the Q-Table.

5.5.3 Results

Despite the problems encountered during the hardware implementation of the Q-Learning light-finding algorithm, results are still obtained. Table 5.3 shows the resulting Q-Table after the agent learned for 30 repetitions in hardware. Once learned, the highest value in the row corresponding to the current state is the action that the robot thinks it should take for the state. All the actions with the highest Q value for each state logically correspond to the correct action that the robot should take. Therefore, the table shows that the robot has learned to navigate to the light in hardware. The results of this Q-Table are shown in Figure 5.13. This figure shows

State / Action	Front Right	Front Center	Front Left
Rear Center	0.9494	-0.6495	-0.5848
Rear Right	0.7950	-0.0449	-0.2863
Center Right	0.7120	0.1365	-0.3091
Front Right	0.4466	0.1255	-0.0286
Front Center	0.8803	1.4476	1.1834
Front Left	0.3014	0.3008	0.8993
Center Left	-0.5585	-0.0546	1.1205
Rear Left	-0.3775	0.0077	0.8443

Table 5.3: Q-Table after a learning on hardware.

the path of the robot navigating towards the light. The resulting path is optimal in the sense that it moves directly towards the light. During learning, it is noticed that the quantization error of the sensed actions results in some of the actions having close Q values. For example, this occurs mostly with the Front Center light state. Table 5.3 shows that the Q values for this state are close compared to the rest of the table. In this state with the quantization error, it ss easy for the right and left actions

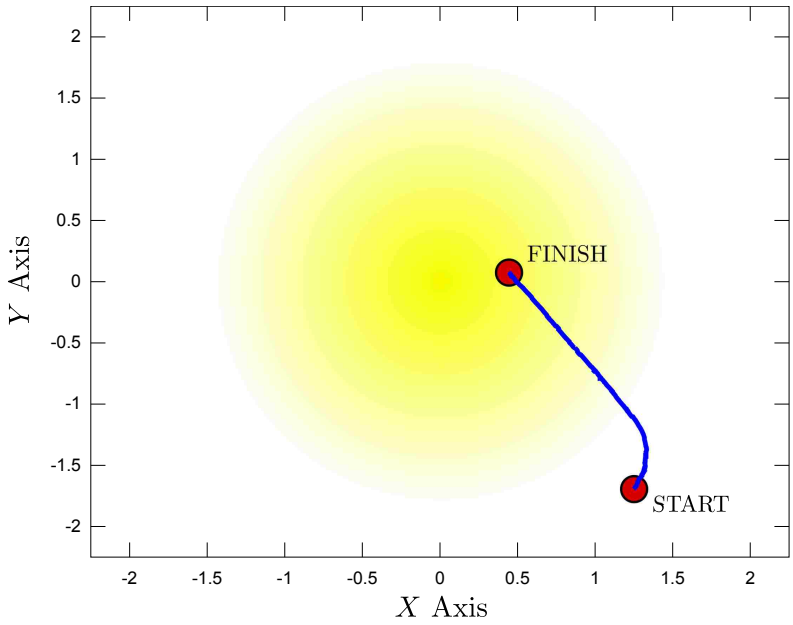


Figure 5.13: The resulting path of the robot using hardware.

to produce good rewards also. Therefore, care needs to be taken when quantizing the sensed actions. Despite the noise and delays that were encountered in hardware experiments, the algorithm performs reasonably well.

Chapter 6

Conclusions and Future Work

In conclusion, the TXT-1 prototype is finished, providing the MARHES lab with a robot capable of autonomous behavior. The TXT-1 is designed to be a robust, safe, and flexible vehicle. The base electromechanical platform is made to be easily replaced and maintained during failures. Many electrical components in the system can even be updated without much redesign of the vehicle. The vehicle allows for several different sensor configurations to be swapped for different experiments. The robot is developed for use with ROS, which will be used for many years to come.

Some future topics of exploration include using the TXT-1 platform to verify algorithms other than learning algorithms, fusing multiple sensors to calculate a highly accurate odometry estimate, and creating more TXT-1's to test and verify algorithms for cooperative control and sensor networks. Another research endeavor could incorporate the IMU or another sensor into the control system to limit speeds to prevent rollover when traveling over non-smooth terrain. Additionally, energy conscious mobile robot algorithms can be developed with the prototype. This testbed vehicle will help in the research of many mobile robotic algorithms.

The Q-Learning algorithm produced encouraging results in simulation and hardware. Using this approach, the robot navigates towards the light, following close to the most direct path. The hardware results are affected by noise in the sensors and delays in the robot response. More work can be done to learn with system delays. Another approach is to learn from the raw sensor inputs and use a neural network to learn the actions [38, 39]. A further possibility for improving performance in the noisy system described here is to use a biologically-inspired reinforcement learner [40]. Biological systems deal with inherently noisy sensors and environments as a matter of course and have adapted sophisticated mechanisms for handling the uncertainty. Lastly, work in multiple robot coordination and learning can be pursued.

Appendix A

Power Distribution Board

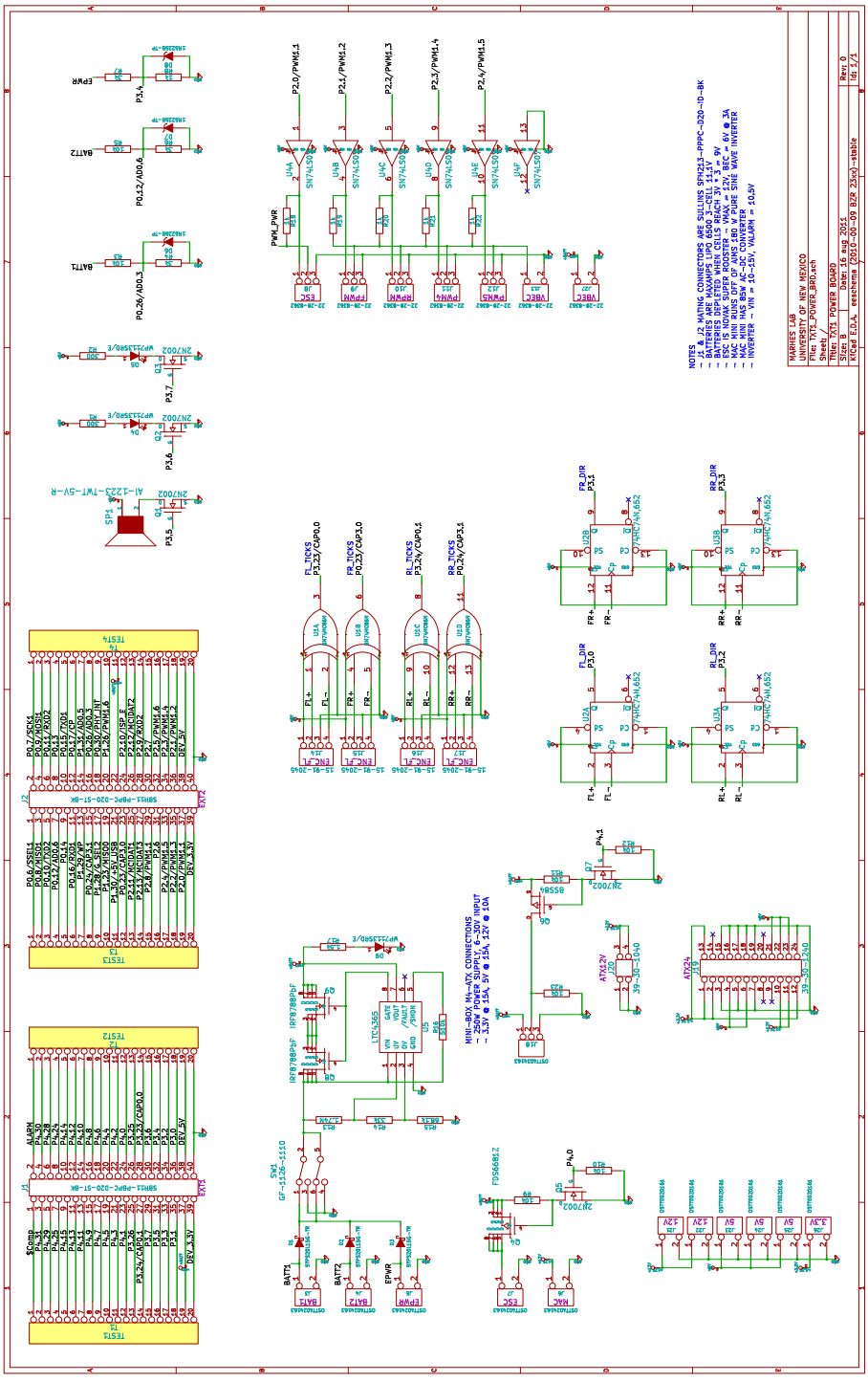


Figure A.1: Power distribution board schematic.

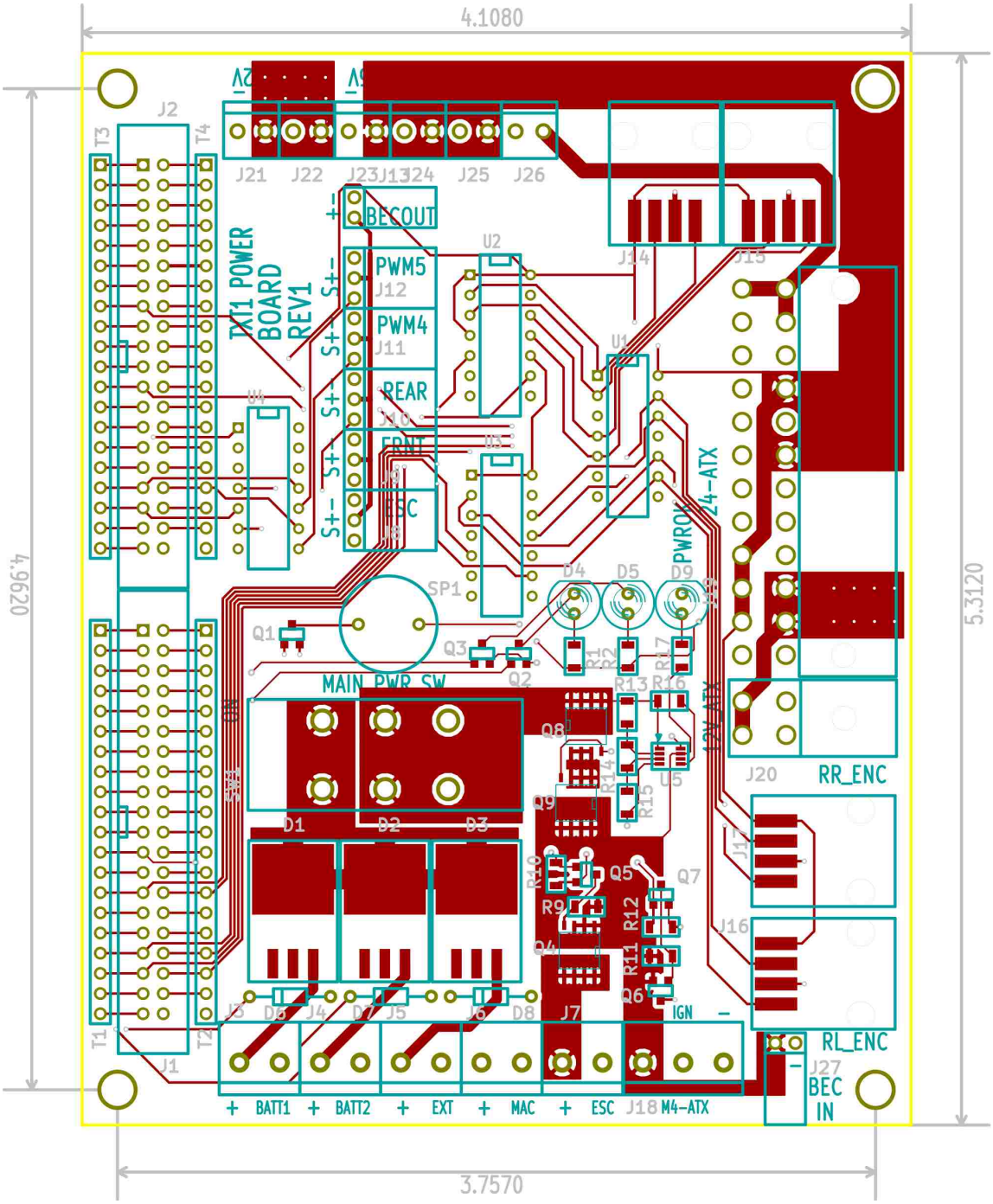


Figure A.2: Power distribution board front layout.

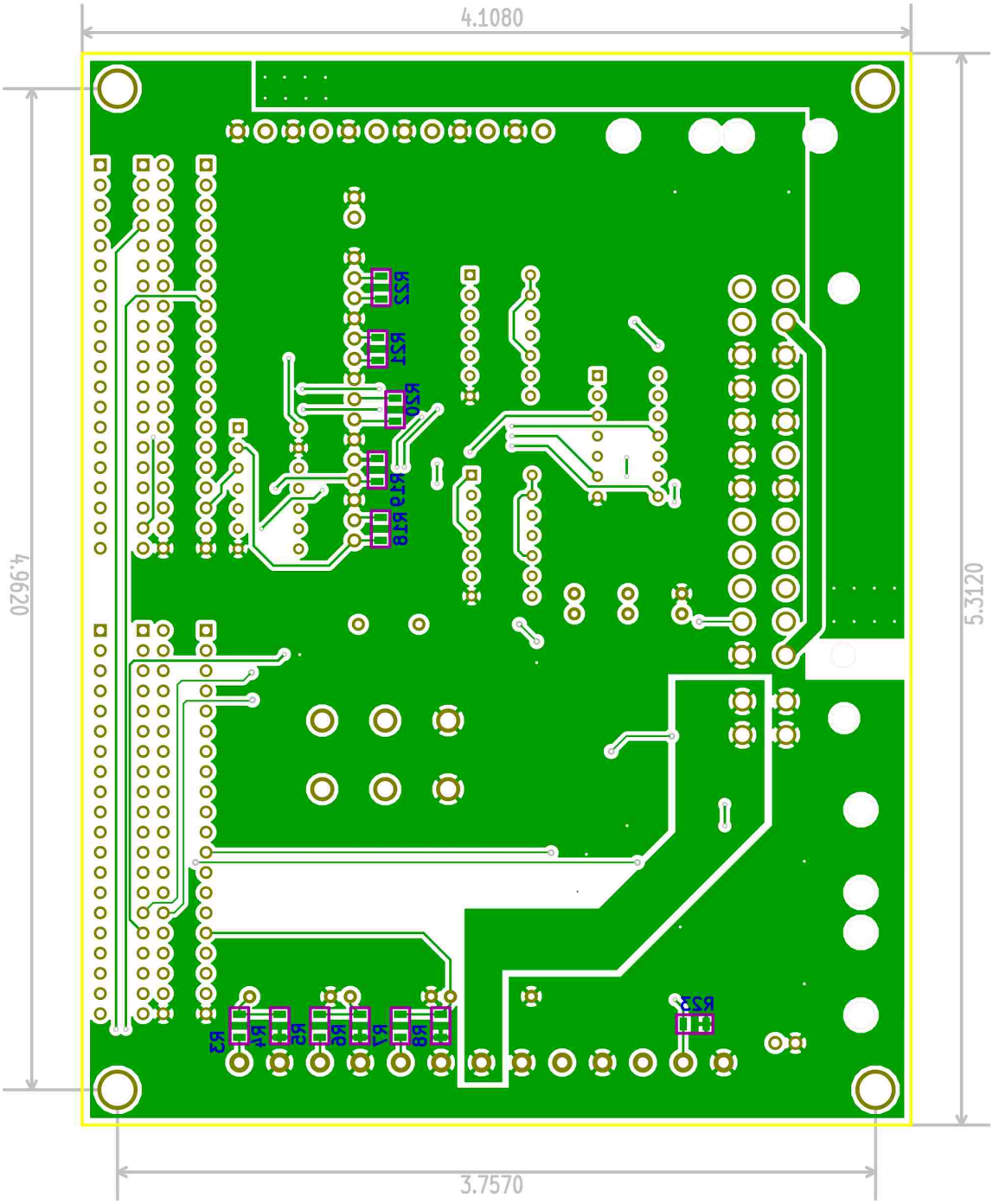


Figure A.3: Power distribution board back layout.

Quantity	Designator	P/N	Designation	Package
1	Q4	FDS6681ZCT-ND	FDS6681Z	SO8N
7	R23,R5,R3,R11,R12,R10,R9	P10.0KFCT-ND	10k	sm_1206
2	J27,J13	WM6136-ND	22-28-8362	th_conn_100_2_ra
6	R22,R21,R20,R19,R18,R8	P1.00KFCT-ND	1k	sm_1206
1	U4	296-14878-5-ND	SN74LS07	DIP-14__300
2	Q8,Q9	IRF8788TRPBFCT-ND	IRF8788PbF	SO8N
1	R13	RHM1.74MFCT-ND	1.74M	sm_1206
1	R14	P33.0KFCT-ND	33k	sm_1206
1	R15	P68.1KFCT-ND	68.1k	sm_1206
1	R16	P510KFCT-ND	510k	sm_1206
1	R17	P1.50KFCT-ND	1.5k	sm_1206
3	D9,D4,D5	754-1276-ND	WP7113SRD/E	LED-5MM
6	J26,J25,J24,J23,J22,J21	ED2635-ND	OSTTE020161	th_conn_138_2
3	R7,R6,R4	P3.00KFCT-ND	3k	sm_1206
2	R2,R1	P300FCT-ND	300	sm_1206
1	SW1	SW105-ND	GF-1126-1110	th_sw_gw_slide
1	SP1	668-1019-ND	AI-1223-TWT-5V-R	th_spkr_pui_ai
5	Q5,Q7,Q1,Q2,Q3	568-5987-1-ND	2N7002	sm_sot_23_3
1	Q6	BSS84W-FDICT-ND	BSS84	sm_sot_23_3
3	D6,D7,D8	1N5225B-TPCT-ND	1N5225B-TP	th_do_35
1	J20	WM1352-ND	39-30-1040	th_conn_atx_4
1	J19	WM1362-ND	39-30-1240	th_conn_atx_24
1	J18	ED2581-ND	OSTTA034163	th_conn_200_3
5	J3,J4,J5,J6,J7	ED2580-ND	OSTTA024163	th_conn_200_2
5	J8,J9,J10,J11,J12	WM6136-ND	22-28-8362	th_conn_100_3_ra
2	J1,J2 S9175-ND	SBH11-PB	PC-D20-ST-BK	th_conn_100_100_20_2
3	D1,D2,D3	497-6578-1-ND	STPS20L15G-TR	sm_to263_3
4	J14,J15,J16,J17	WM1342-ND	15-91-2045	sm_conn_molex_15-91-2045
1	U1	296-8375-5-ND	SN74HC86N	DIP-14__300
2	U2,U3	568-1491-5-ND	74HC74N,652	DIP-14__300
1	U5	LTC4365CTS8#TRMPBFCT-ND	LTC4365	SM_TSOT_23_8

Table A.1: Power distribution board parts list.

Appendix B

TXT-1 Source Code

B.1 MARHES Repositories

With over 4,000 lines of code for the lower level controller, 4,000 lines of code for the txt_driver and other test programs and scripts, and 4,000 lines of code for the light finding application, it is necessary to keep the code managed and safe. Therefore, Google code repositories were made for the lower level controller and any MARHES ROS code. Both of the repositories use mercurial to make any code changes or updates. The lower level controller code is kept at:

`http://code.google.com/p/marhes-txt/.`

The ROS code including the TXT-1 driver and the light finding program is kept at:

`http://code.google.com/p/marhes-ros-pkg/.`

B.2 Development Board Programming Instructions

Things Needed

- Olimex LPC2378 Development Board (ordered from Sparkfun)
- Olimex ARM-USB-OCD JTAG Programmer/Debugger (ordered from Sparkfun)
- Computer with Ubuntu installed (these instructions for Ubuntu 10.04)
  - Internet Connection
  - USB Port

Setup Steps

1. Install OpenOCD
   (a) Open a terminal
   (b) Type `sudo apt-get install openocd`
2. Install CodeSourcery Lite for ARM EABI
   (a) Go to `http://www.codesourcery.com/sgpp/lite/arm/portal/subscription3053`
   (b) Download this version: Lite 2010q1-188
   (c) Download the IA32 GNU/Linux Installer
   (d) In the terminal, type `sudo dpkg-reconfigure -plow dash`
   (e) Select No

(f) Type `/bin/sh ./"path to package"/arm-"version"-arm-none-eabi.bin` where "path to package" is the path to the downloaded installer and "version" is the version of the installer.

(g) Go through the installation steps

i. Typical

ii. Directory: `/home/marhes/CodeSourcery/Sourcery_G++_Lite`

iii. Modify PATH

(h) Add `export PATH=$PATH:/home/marhes/CodeSourcery/Sourcery_G++_Lite/bin` to the end of `~/.bashrc`

3. Install DDD

(a) Open a terminal

(b) Type `sudo apt-get install ddd`

Compiling and Programming

1. Compile the source code

(a) Open a terminal

(b) `cd "source folder with makefile"`

(c) `make`

2. Start OpenOCD

(a) Connect JTAG programmer

(b) Go to source directory with *StartOpenOCD.sh*

(c) In a terminal, `./StartOpenOCD.sh`

3. Program and Debug the development board

(a) Make sure OpenOCD is running

(b) Go to source directory with *ddd.sh*

(c) In a terminal, `./ddd.sh`

(d) **Note:** To debug without programming, use *dddRun.sh*

B.3 Instructions for Using the TXT-1 Source Example

To use the TXT-1 source example, this command should be run while in a directory that is in the `$ROS_PACKAGE_PATH`:

```
roscreate-pkg name roscpp txt_driver tf nav_msgs geometry_msgs,
```

where `name` is the name of the package to create. Then, the example source code can be copied to the `src` directory. Next, the CMakeLists.txt file needs to be modified so the template gets compiled into an executable. Add the following to the end of the file,

```
rosbuild_add_executable(template src/txt_driver_template.cpp)
```

where `template` is the executable name and `txt_driver_template.cpp` is the copied example source file. Then use `rosmake name` to compile the newly created package.

Next start the TXT-1 ROS driver launch file with

```
roslaunch txt_launch txt_node_net.launch (continued on next line)
  host:=hostname client:=clientname
```

where `hostname` is the name of the host computer and `clientname` is the name of the computer on the TXT-1 robot. Next, run the example code executable with `rosrun name template`. The example source code should now be running. To terminate, use `CTRL+C`.

B.4 The TXT-1 ROS Driver Example

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "nav_msgs/Odometry.h"
#include "txt_driver/Battery.h"
#include "txt_driver/Pwm.h"
#include "txt_driver/PidLoad.h"
#include "txt_driver/PwmTest.h"
#include "txt_driver/Shutdown.h"
#include "txt_driver/SwitchedPwr.h"
#include "tf/tf.h"

// The class to handle all the txt_driver topics and services
class TxtTemplate
{
public:
  // Function prototypes
  TxtTemplate(ros::NodeHandle nh);
private:
  // ROS publishers, subscribers, service clients and timers
  ros::NodeHandle n_;
  ros::Publisher vel_pub_, pwm_pub_, comb_odom_pub_;
```

```
  ros::Subscriber odom_sub_, batt_sub_;
  ros::ServiceClient pid_client_, shutdown_client_, pwm_client_
      , pwr_client_;
  ros::Timer pub_tmr_;

  // Storage variables
  nav_msgs::Odometry odom_msg_;

  // Function prototypes
  void odomCB(nav_msgs::Odometry msg);
  void battCB(txt_driver::Battery msg);
  void tmrCB(const ros::TimerEvent& e);
};

// Class constructor
TxtTemplate::TxtTemplate(ros::NodeHandle nh)
{
  n_ = nh;

  // Setup publishers
  vel_pub_ = n_.advertise<geometry_msgs::Twist>("/cmd_vel", 1);
  pwm_pub_ = n_.advertise<txt_driver::Pwm>("/pwm", 1);
  comb_odom_pub_ = n_.advertise<nav_msgs::Odometry>("/odom_comb
      ", 1);

  // Setup subscribers
  odom_sub_ = n_.subscribe("/odom", 1, &TxtTemplate::odomCB, 
      this);
  batt_sub_ = n_.subscribe("/battery", 1, &TxtTemplate::battCB,
       this);
```

```
  // Setup service clients
  pid_client_ = n_.serviceClient<txt_driver::PidLoad>("/↩
     pid_load");
51  shutdown_client_ = n_.serviceClient<txt_driver::Shutdown>("/↩
     shutdown_computer");
  pwm_client_ = n_.serviceClient<txt_driver::PwmTest>("/↩
     pwm_change");
  pwr_client_ = n_.serviceClient<txt_driver::SwitchedPwr>("/↩
     switched_pwr");

  // Setup a 2s timer callback to publish messages and service ↩
     calls
  pub_tmr_ = n_.createTimer(ros::Duration(2.0), &TxtTemplate::↩
     tmrCB, this);
}

// The odom callback
void TxtTemplate::odomCB(nav_msgs::Odometry msg)
61 {
  odom_msg_ = msg;                  // Store the odometry message
  ROS_INFO("RXed odom: X: %f, Y: %f, YAW: %f, V: %f, W: %f",
    msg.pose.pose.position.x,
    msg.pose.pose.position.y,
    tf::getYaw(msg.pose.pose.orientation),
    msg.twist.twist.linear.x,
    msg.twist.twist.angular.z);
}

71 // The battery callback
```

```
void TxtTemplate::battCB(txt_driver::Battery msg)
{
  ROS_INFO("RXed battery: V1: %f, V2: %f, Expected Time: %f",
    msg.batt1,
    msg.batt2,
    msg.expected_time.toSec());
}

// The timer callback
void TxtTemplate::tmrCB(const ros::TimerEvent& e)
{
  // publish velocity message
  geometry_msgs::Twist vel_msg;
  vel_msg.linear.x = 0.5;
  vel_msg.angular.z = 0.3;
  vel_pub_.publish(vel_msg);

  // publish pwm message
  txt_driver::Pwm pwm_msg;
  pwm_msg.pwm4 = 12000;
  pwm_msg.pwm5 = -24000;
  pwm_msg.publish(pwm_msg);

  // publish the stored odom_message as an example
  comb_odom_pub_.publish(odom_msg_);

  // Load the PID values
  txt_driver::PidLoad pid_srv;
  if(pid_client_.call(pid_srv))
    ROS_INFO("PID Load Response: %d", pid_srv.response.result);
```

```
    // Shutdown the computer
    txt_driver::Shutdown sd_srv;
    sd_srv.request.request = true;
    if(shutdown_client_.call(sd_srv))
      ROS_INFO("Shutdown called: %d", sd_srv.response.result);

    // Send pwm values for manual mode
    txt_driver::PwmTest pwm_srv;
111 pwm_srv.request.esc = 24000;
    pwm_srv.request.front = 12000;
    pwm_srv.request.rear = 0;
    pwm_srv.request.pwm4 = -12000;
    pwm_srv.request.pwm5 = -24000;
    if(pwm_client_.call(pwm_srv))
      ROS_INFO("PWM Response: %d", pwm_srv.response.result);

    // Set the switch state of the ATX switch to on
    txt_driver::SwitchedPwr pwr_srv;
121 pwr_srv.request.source = pwr_srv.request.SOURCE_ATX;
    pwr_srv.request.on = true;
    if(pwr_client_.call(pwr_srv))
      ROS_INFO("PWR Response: %d", pwr_srv.response.result);
  }

  // The main function initializes ROS and starts the TxtTemplate↩
      node and then
  // spins, servicing callbacks.
  int main(int argc, char **argv)
  {
```

```
131     ros::init(argc, argv, "template");
        ros::NodeHandle n;

        TxtTemplate * t = new TxtTemplate(n);

        ros::spin();
        return 0;
    }
```

References

[1] DARPA, "Grand Challenge '05," Dec. 2007. [Online]. Available: http://archive.darpa.mil/grandchallenge05/index.html

[2] ——, "Urban Challenge," Apr. 2008. [Online]. Available: http://archive.darpa.mil/grandchallenge/index.asp

[3] J. Markoff, "Google cars drive themselves, in traffic," Oct. 2010, The New York Times. [Online]. Available: http://www.nytimes.com/2010/10/10/science/10google.html?

[4] SARTRE-Consortium, "The SARTRE Project," May 2011. [Online]. Available: http://www.sartre-project.eu

[5] "MobileRobots Pioneer3-AT," 2011. [Online]. Available: http://www.mobilerobots.com/ResearchRobots/ResearchRobots/P3AT.aspx

[6] "Jaguar platform specification," 2010. [Online]. Available: http://jaguar.drrobot.com/specification.asp

[7] "Automated material handling order fulfillment system," 2011. [Online]. Available: http://www.kivasystems.com/

[8] "GRASP Lab Robot Platforms," 2001. [Online]. Available: http://www.cis.upenn.edu/mars/site/platforms.htm

[9] E. Edwan, "Design of a modular autonomous robot vehicle," Master's thesis, Oklahoma State University, Aug. 2003.

[10] D. E. Cruz, "An experimental testbed for swarming and cooperative robotic networks," Master's thesis, Oklahoma State University, Jul. 2006.

[11] B. K. Wilburn, "Hardware, software, and low-level control scheme development for a real-time autonomous rover," Master's thesis, West Virginia University, 2010.

[12] M. Stanley, "Implementation of kalman filter to tracking custom four-wheel drive four-wheel-steering robotic platform," Master's thesis, University of Maryland, 2010.

[13] J. Marshall, "Coordinated autonomy: Pursuit formations of multivehicle systems," Ph.D. dissertation, University of Toronto, 2005.

[14] S. Falamaki, "Simultaneous localisation and mapping on a model off-road vehicle," Master's thesis, The University of New South Wales, Jun. 2005.

[15] K.-H. Park, Y.-J. Kim, and J.-H. Kim, "Modular Q-learning based multi-agent cooperation for robot soccer," *Robotics and Autonomous Systems*, vol. 35, no. 2, pp. 109–122, 2001.

[16] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," *Auton. Robots*, vol. 27, no. 1, pp. 55–73, 2009.

[17] C. Kroustis and M. Casey, "Combining heuristics and Q-Learning in an adaptive light seeking robot," University of Surrey, Department of Computing, Tech. Rep. CS-08-01, 2008.

[18] M. Riedmiller, M. Montemerlo, and H. Dahlkamp, "Learning to drive a real car in 20 minutes," in *FBIT*, 2007, pp. 645–650.

[19] C. Gaskett, "Q-Learning for robot control," Ph.D. dissertation, The Australian National University, 2002.

[20] C. Chen, H.-X. Li, and D. Dong, "Hybrid control for robot navigation - a hierarchical Q-Learning algorithm," *Robotics Automation Magazine IEEE*, vol. 15, no. 2, pp. 37–47, 2008.

[21] "4x4 Monster Pick-up TXT-1." [Online]. Available: http://www.tamiya.com/english/products/58280txt_1/txt_1.htm

[22] "Super Rooster." [Online]. Available: http://www.teamnovak.com/products/esc/super_rooster/index.html

[23] "LPC2378 Development Board." [Online]. Available: http://www.olimex.com/dev/lpc-2378stk.html

[24] "Sourcery CodeBench Lite Edition." [Online]. Available: http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/lite-edition

[25] R. Barry, "The FreeRTOS Project," Jul. 2011. [Online]. Available: http://www.freertos.org/

[26] "Apple - Mac Mini." [Online]. Available: http://www.apple.com/macmini/

[27] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and N. Andrew, "ROS: an open-source Robot Operating System," in *International Conference on Robotics and Automation*, 2009.

[28] *Pioneer3 Operations Manual*, 3rd ed., MobileRobots Inc., Jan. 2006.

[29] "Vicon Tracker." 2011. [Online]. Available: http://www.vicon.com/products/vicontracker.html

[30] R. Taylor II, T. Hudson, A. Seeger, H. Weber, J. Juliano, and A. Helser, "VRPN: A device-independent, network-transparent VR peripheral system," in *Proceedings of the ACM Symposium on Virtual Reality Software & Technology 2001, VRST 2001*, Nov. 15-17, 2001.

[31] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998. [Online]. Available: http://www.cs.ualberta.ca/%7Esutton/book/ebook/the-book.html

[32] L. P. Kaelbling, M. Littman, and A. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[33] P. Dayan and C. Watkins, "Reinforcement learning," in *Encyclopedia of Cognitive Science*. MacMillan Press, London, England, 2001.

[34] C. J. C. H. Watkins, "Learning from delayed rewards," in *International Symposium on Physical Design*, 1989.

[35] P. Dayan, "Technical note: Q-Learning," *Machine Learning*, vol. 292, no. 3, pp. 279–292, 1992. [Online]. Available: http://www.springerlink.com/index/T774414027552160.pdf

[36] R. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *In Proceedings of the Seventh International Conference on Machine Learning*. Morgan Kaufmann, 1990, pp. 216–224.

[37] "Phidgets Inc. - Unique and Easy to use USB Interfaces." 2010. [Online]. Available: http://www.phidgets.com/index.php

[38] V. Ganapathy, C. Y. Soh, and W. L. D. Lui, *Utilization of Webots and the Khepera II as a Platform for Neural Q-Learning Controllers.* IEEE, 2009, pp. 783–788.

[39] B.-Q. Huang, G.-Y. Cao, and M. Guo, "Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance," in *Proceedings of 2005 International Conference on Machine Learning and Cybernetics*, 2005, pp. 85–89.

[40] B. Rohrer, "A developmental agent for learning features, environment models, and general robotics tasks," in *Joint IEEE International Conference on Development and Learning and on Epigenetic Robotics, Frankfurt, Germany*, Aug. 24-27, 2011.