ARTIFICIAL NEURAL NETWORK-BASED ROBOTIC CONTROL

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Justin Ng

June 2018

COMMITTEE MEMBERSHIP

TITLE:	Artificial Neural Network-Based Robotic Control
AUTHOR:	Justin Ng
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Andrew Danowitz, Ph.D. Assistant Professor of Electrical Engineering
COMMITTEE MEMBER:	Xiao-Hua Yu, Ph.D. Professor of Electrical Engineering
COMMITTEE MEMBER:	Fred W. DePiero, Ph.D. Professor of Electrical Engineering

ABSTRACT

Artificial Neural Network-Based Robotic Control

Justin Ng

Artificial neural networks (ANNs) are highly-capable alternatives to traditional problem solving schemes due to their ability to solve non-linear systems with a non-algorithmic approach. The applications of ANNs range from process control to pattern recognition and, with increasing importance, robotics. This paper demonstrates continuous control of a robot using the deep deterministic policy gradients (DDPG) algorithm, an actor-critic reinforcement learning strategy, originally conceived by Google DeepMind. After training, the robot performs controlled locomotion within an enclosed area. The paper also details the robot design process and explores the challenges of implementation in a real-time system.

ACKNOWLEDGMENTS

Thanks:

- To Andrew Danowitz for his guidance, support, and time throughout the project. Many thanks!
- To my best friend in the world and unwavering fountain of support, Earth Wimolnit. You kept me alive and sane, and I couldn't have done it without you! Thank you and I love you!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

Chapter 1

INTRODUCTION

The game of Go has challenged players around the world for centuries, and since the 1970's, international competitions have sought the world's best Go minds [5]. The relatively simple rules of the game betray its true complexity and myriad moves; for example, there are 361 possible opening moves while chess has a mere 20 [9]. In fact, for the 19x19 Go board, about 2×10^{170} possible board states exist which is about 10^{90} times more than the number of atoms in the known universe [2]. To produce a Go-playing artificial intelligence (AI) at the level of a human expert has been somewhat of a holy grail for machine learning researchers [11]. Until 2014, the very best Go programs were only capable of occasionally beating top amateur players. However, in 2015, AlphaGo grabbed the world's attention with a landmark win, beating European champion Go player Fan Hui in a 5–0 upset [10]. In March 2016, AlphaGo defeated 18-time world champion Lee Sedol 4–1. The success of AlphaGo's algorithms along with other research from DeepMind paved the way for improvements in reinforcement learning, especially in the previously intractable area of continuous control [37][52].

Reinforcement learning (RL), the branch of machine learning concerned with how an actor should take actions in an environment to receive the most reward, has previously struggled with environments containing large state and/or action spaces. The recent development of deep Q-network (DQN), deep policy gradients (DPG), and deep deterministic policy gradients (DDPG) within the past four years has opened the door to RL application in continuously controlled robotics [37][51][30]. This thesis covers the implementation of the off-policy, model-free DDPG algorithm in a purpose-built robot entered in the 2018 Cal Poly Roborodentia [12]. Without knowledge of sensors or environment, the robot learns to control four independent mecanum wheels

to move the robot to a desired position and orientation.

The paper is broken down into four chapters covering the various areas of development: mechanical, electrical, firmware, and reinforcement learning. Additionally, all project files can be found at `https://www.github.com/okayjustin/roborodentia2017`.

1.1 Cal Poly Roborodentia

The robot is designed to compete in the 2018 Cal Poly Roborodentia, the university's annual intramural robotics competition, and thus conforms to its particular specifications and requirements [12]. Briefly, competitors must produce autonomous robots to collect and fire Nerf Rival Balls [22], shown in Figure 1.2, into nets to win points. A drawing of the field is shown in Figure 1.1. Two robots compete separately in each half so the effective field is a 4' wide by 8' long area enclosed by 4" walls. 1 inch PVC tubes along the 4' walls hold the balls which the robots fire into rectangular nets located along the 8' walls. The rules provide additional restrictions on robot dimensions, capabilities, and other aspects, to be covered in the following chapters.

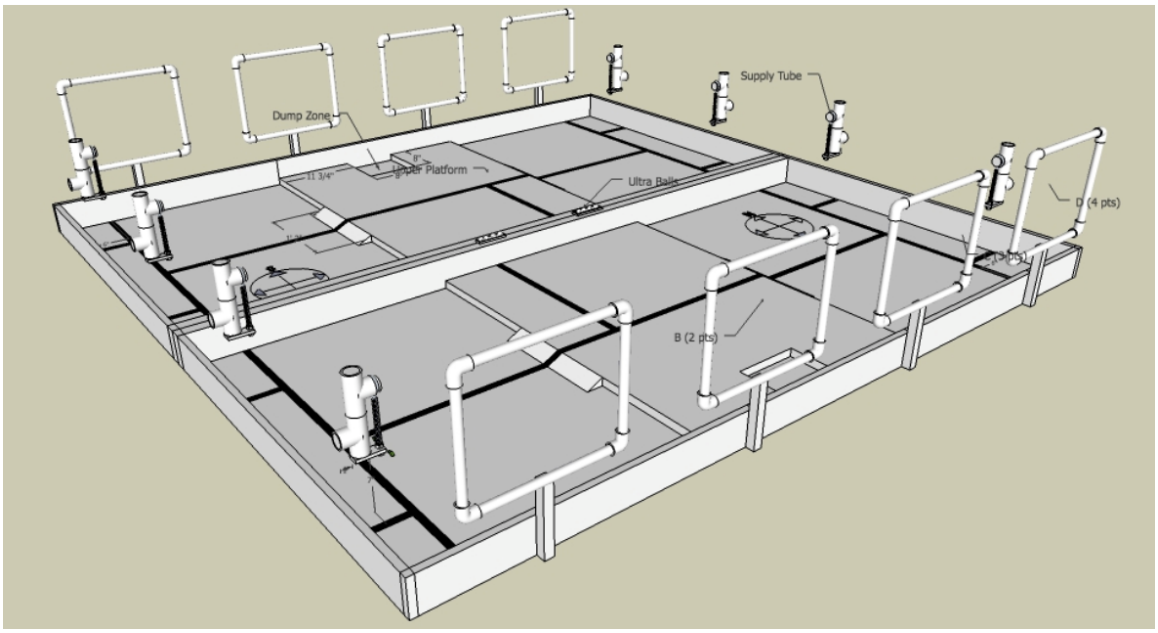


Figure 1.1: Roborodentia Field [12]

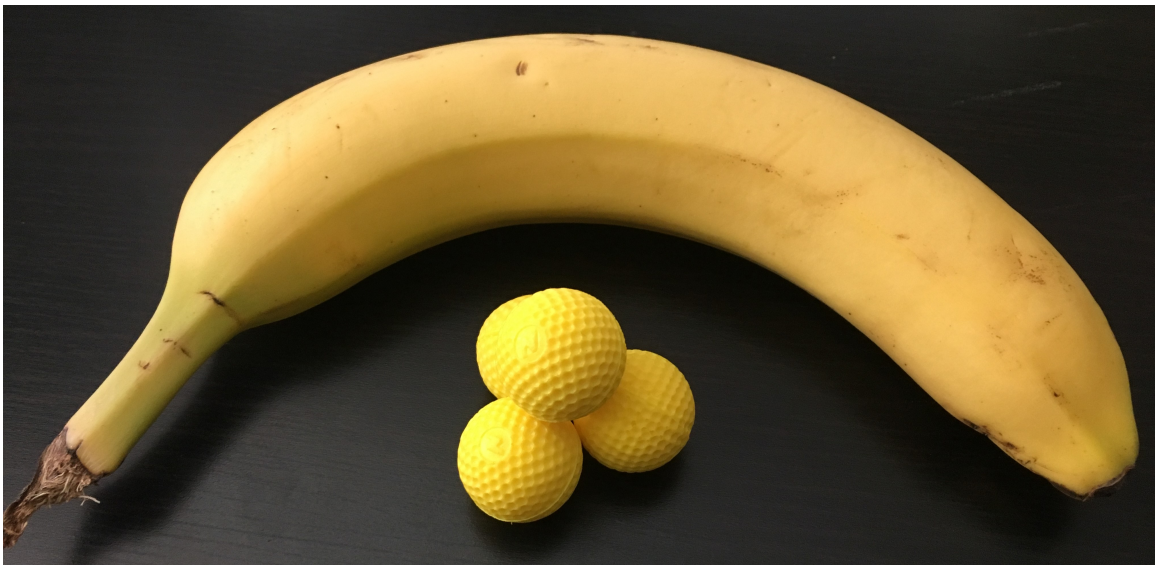


Figure 1.2: Nerf Rival Balls with Banana for Scale

Chapter 2

MECHANICAL DESIGN

The robot meets the Roborodentia design requirements shown in Table 2.1. It consists of four subassemblies: the base platform, shooting mechanism, ball hopper, and control unit. Each section was modeled in SolidWorks, an industry-standard solid modeling CAD program [14][13]. The parts were fabricated using a laser cutter or 3D printer and assembled with metric hardware. Figure 2.1 displays a photograph of the robot while Figures 2.2 through 2.5 show standard view renders of the SolidWorks model. Note that the robot uses mecanum wheels [24] (a type of omni-directional wheel) which are modeled as plain wheels for simplicity. Figure 2.3 also indicates the front, left, back/rear, and right sides of the robot as referenced in the rest of the paper. The robot contains 53 different mechanical components (including fasteners and off-the-shelf parts), of which 24 are custom designed, and 394 parts in total. The bill of materials can be found in Appendix A.

Table 2.1: Roborodentia 2018 Mechanical Requirements

	Requirement
1.	Maximum footprint of 12" x 14" or smaller at start of match but may expand up to 14" x 17" during match.
2.	Maximum height of 14" at start of match but no restriction during match.
3.	Robot may not disassemble into multiple parts.
4.	Robot may not be airborne.
5.	Shooting mechanisms may not accelerate balls past 50 feet per second.

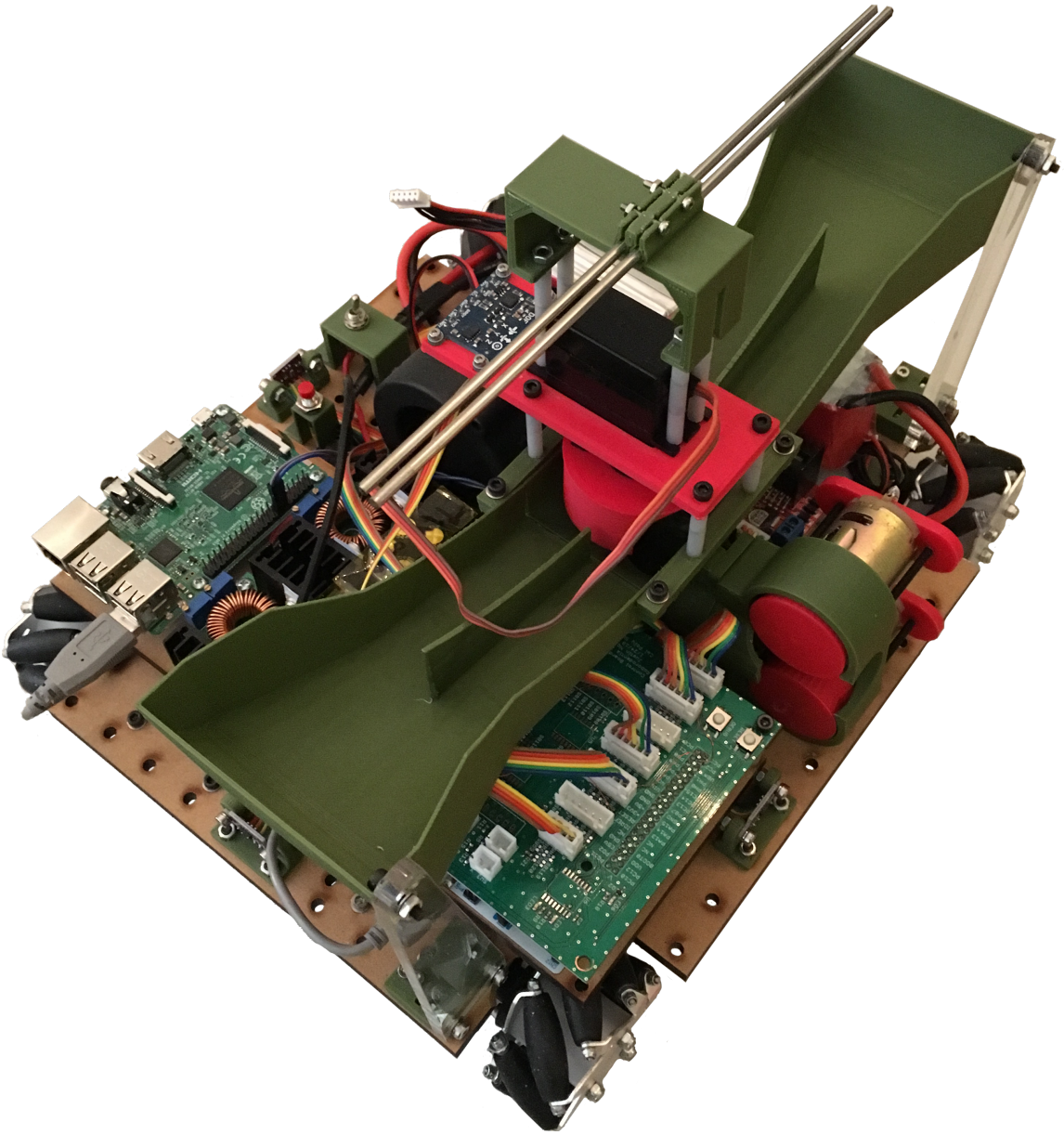


Figure 2.1: Photograph of Robot

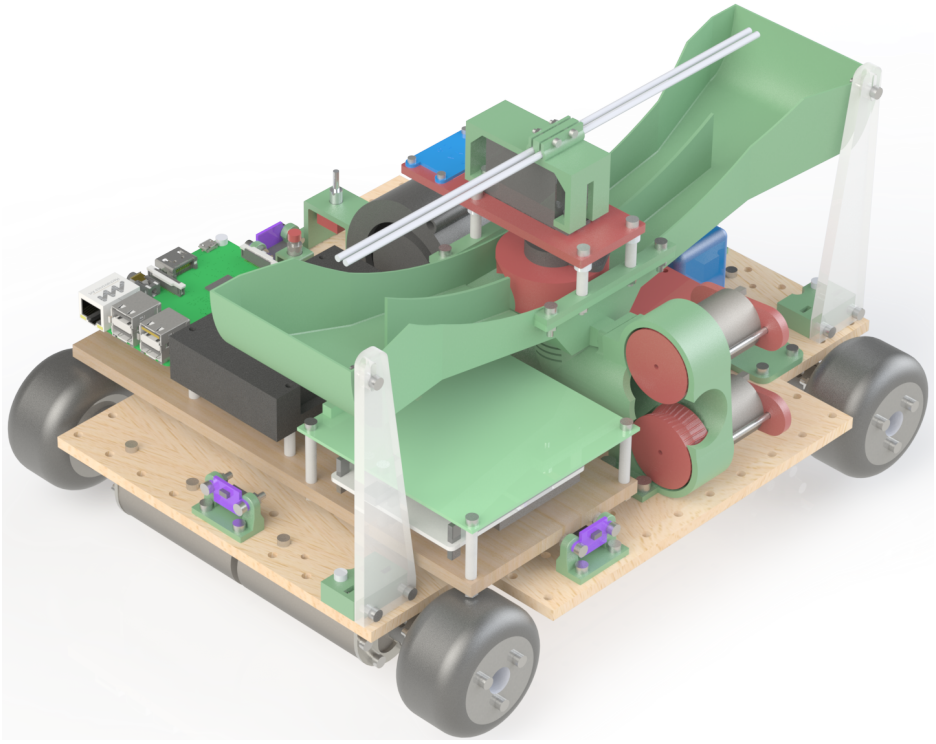


Figure 2.2: Full Robot Render – Isometric View

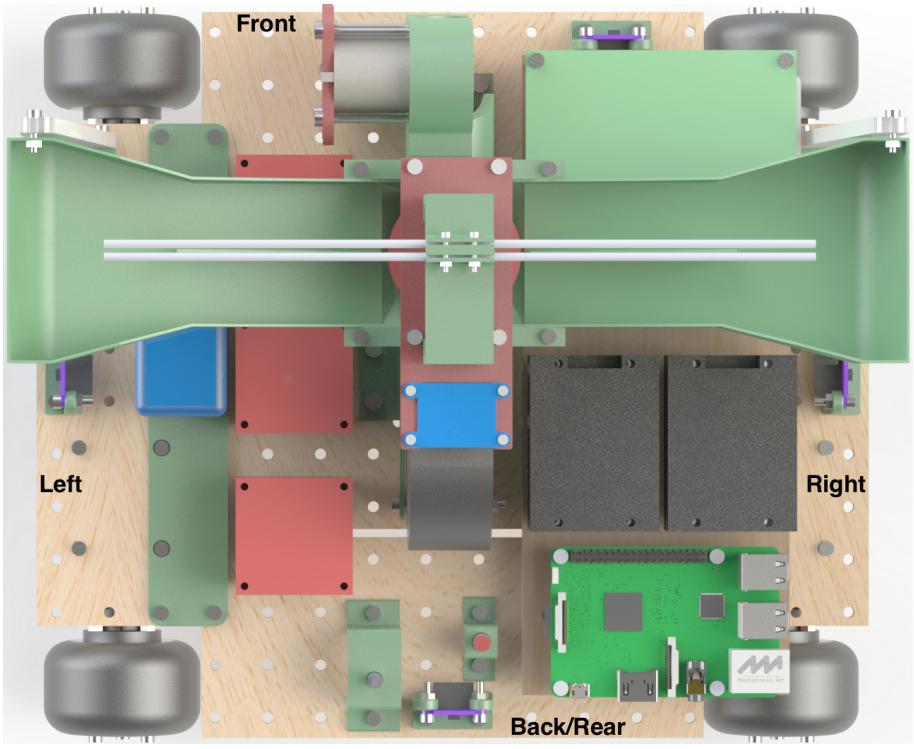


Figure 2.3: Full Robot Render – Top View

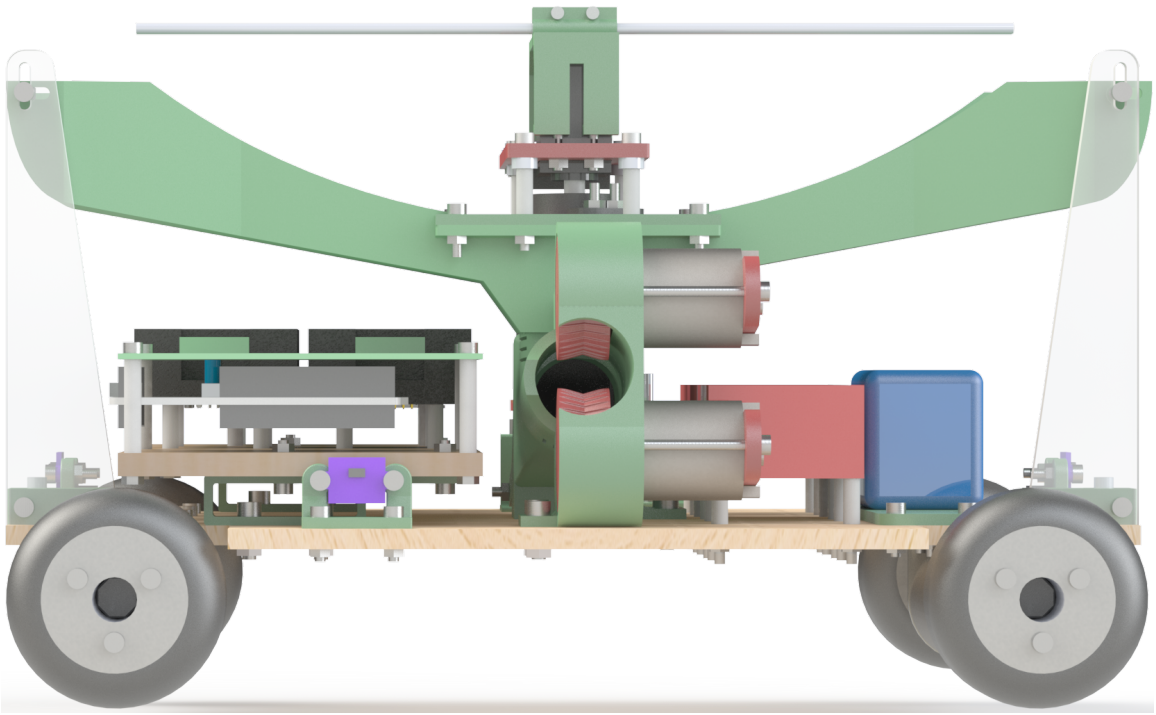


Figure 2.4: Full Robot Render – Front View

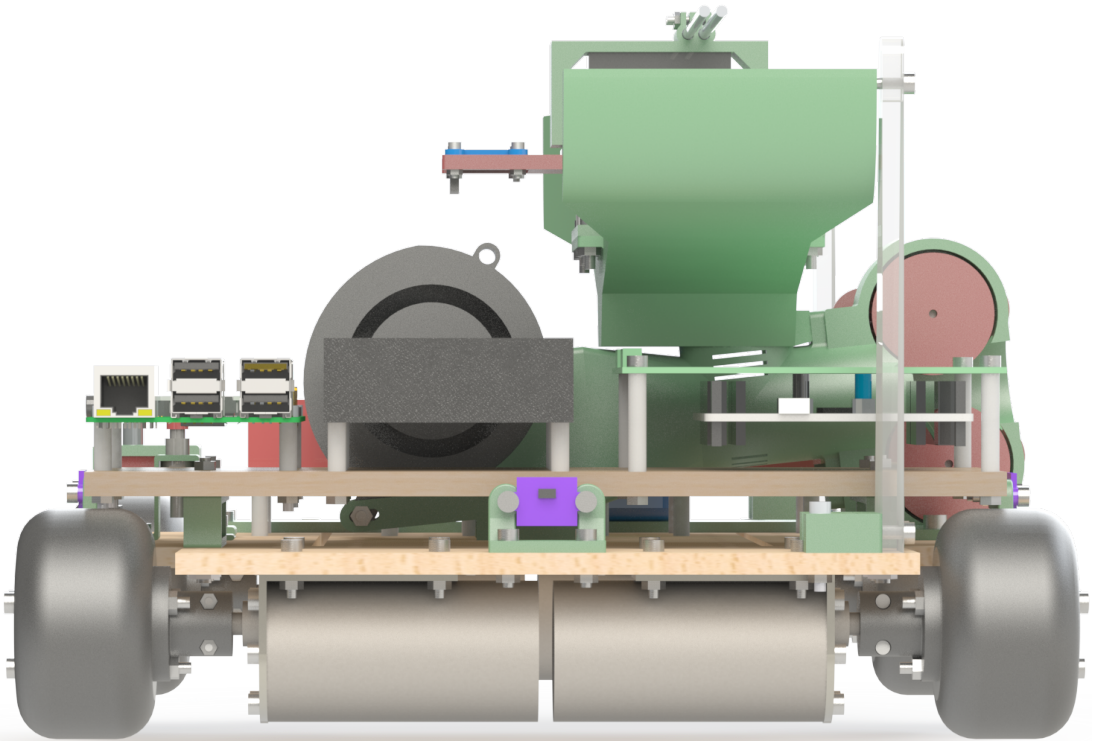


Figure 2.5: Full Robot Render – Right View

The design emphasizes the use of 3D-printed parts to take advantage of the benefits of the technology including rapid part production, high part complexity, and low fabrication cost (excluding the cost of the printer, relative to other methods such as machining, casting, and injection molding), ideal for one-off part production. Parts were printed on a MakerGear M2 fused deposition modeling (FDM) 3D printer equipped with a 0.35 mm diameter nozzle using MakerGeeks 1.75 mm ABS thermoplastic filament [32][33]. Print speeds of 20 to 80 mm/s with 0.20 mm layer height, depending on part dimensions and minimum feature size, lead to part print times of 2–5 hours for smaller components and up to 26 hours for the ball hopper. Due to the non-isotropic strength characteristic of 3D-printed components, the designs must specifically take into account printing direction and orientation. Parts tend to possess greater tensile strength in the X and Y axes but significantly less in the Z direction [50]. Additionally, the print volume of 200 mm x 250 mm x 200 mm limits the maximum part dimensions, requiring the ball hopper to printed as three separate pieces [32].

2.1 Base Platform

The 315 mm x 275 mm base platform of the robot, made from 1/4" thick medium density fiberboard (MDF), serves as the primary structural component and mounting point for the motors, electronics, shooting mechanism, and hopper. The wood is laser cut with a 20 mm grid of 4 mm diameter holes to allow modular placement of components. The corner cutouts allow clearance for the wheels. Long slots permit wire routing from the motors to the motor drivers above. Figure 2.6 shows the subassembly with labels.

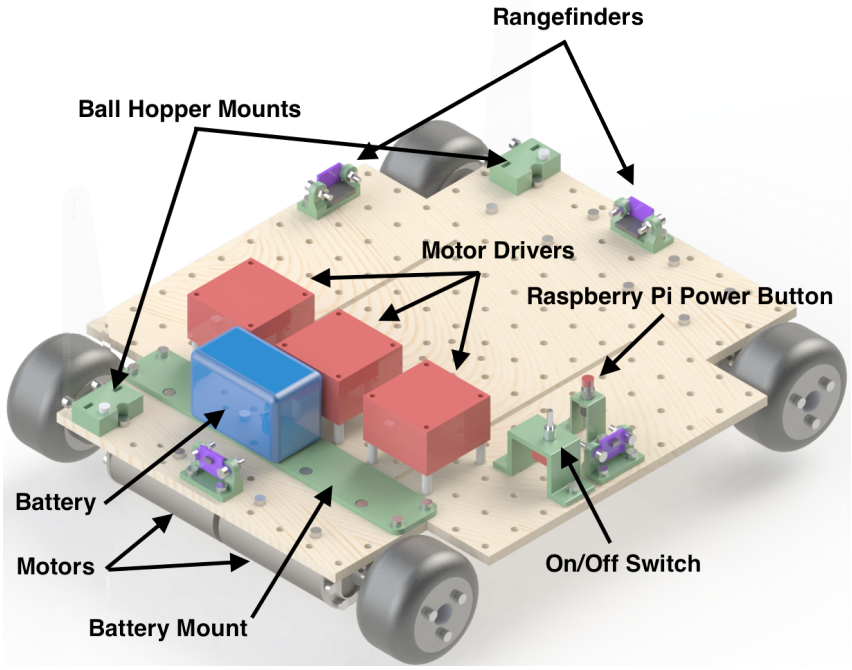


Figure 2.6: Base Platform

Four 12V Pololu 37D motors geared at a 70:1 ratio drive each of the 60 mm diameter mecanum wheels [46]. Figure 2.7 displays an exploded view of the motor assembly. 3D-printed couplings, detailed in Figure 2.8, connect the wheels to the 6 mm diameter, D-shaped motor shafts. The couplers use M2 nuts and bolts to clamp onto the motor shaft and an octagonal stub that press fits into the center of the wheels. Three long M3 bolts and nylon lock nuts clamp the mecanum wheels together and to the coupler. Each wheel contains eight angled rollers mounted with two ball bearings each to smooth operation under load. Unlike regular wheels which only produce a force vector perpendicular to the axis, mecanum wheels also produce a vector parallel to the axis. With the appropriate combination of speed and direction of each wheel, the robot can achieve simultaneous translation and rotation in any direction.

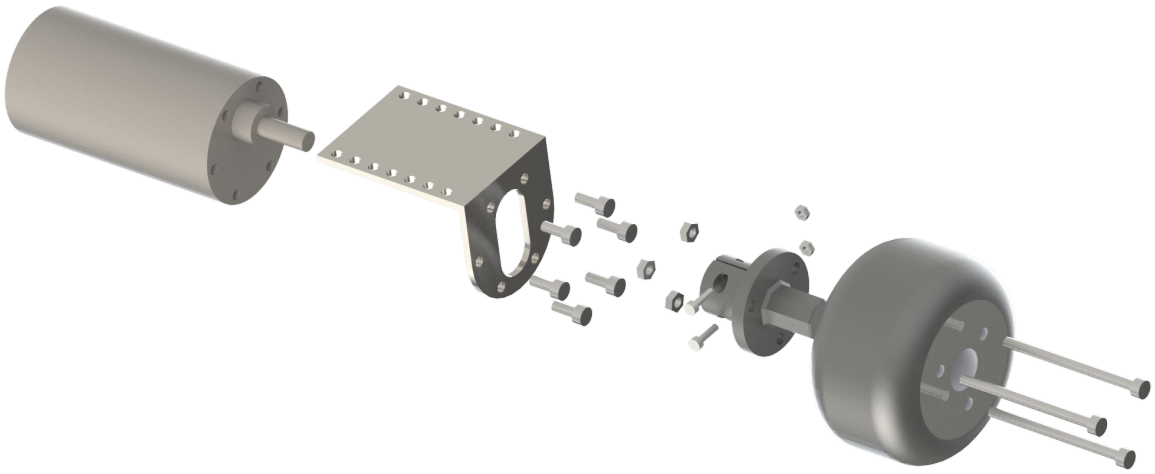


Figure 2.7: Motor Assembly Exploded View

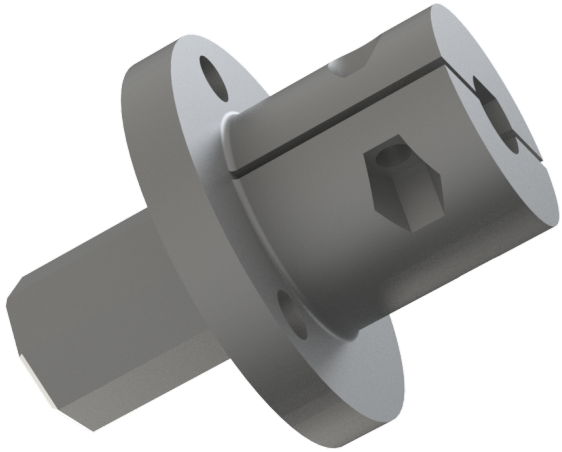


Figure 2.8: Wheel Coupler

Four STMicroelectronics VL53L0X laser rangefinders on the robot periphery sense distances between 30 mm and 2000 mm at a rate of 30 Hz and provide less than 10% error in most test conditions [58]. A 4S, 1800 mAH LiPo battery mounted with an industrial strength hook-and-loop fastener powers the system through a on/off toggle switch seated in a 3D-printed bracket [1]. Three dual H-bridge motor drivers are attached with nylon standoffs and M3 nuts and bolts. Finally, a momentary push button in a 3D-printed bracket toggles power to the Raspberry Pi microcomputer.

2.2 Shooting Mechanism

The competition calls for robots to fire Nerf Rival Balls into large nets positioned several feet away. The shooting mechanism takes inspiration from the official Nerf Rival Blaster toys since they're specifically optimized to fire Nerf Rival balls; the system works similarly to a baseball pitching machine. Figure 2.9 shows an exploded view of the robot's shooter subassembly while Figure 2.10 displays the top view. The mechanism consists of two sections: the **barrel** and the **wheel housing**. Both parts were 3D-printed as the geometries are highly complex. Therefore, the shooting mechanism consists of two separate components versus a unibody design to allow each half to be fabricated with optimal print direction, strength, and finish quality. The barrel is angled 6° above horizontal, targeting the vertical center of the nets 6 feet away.

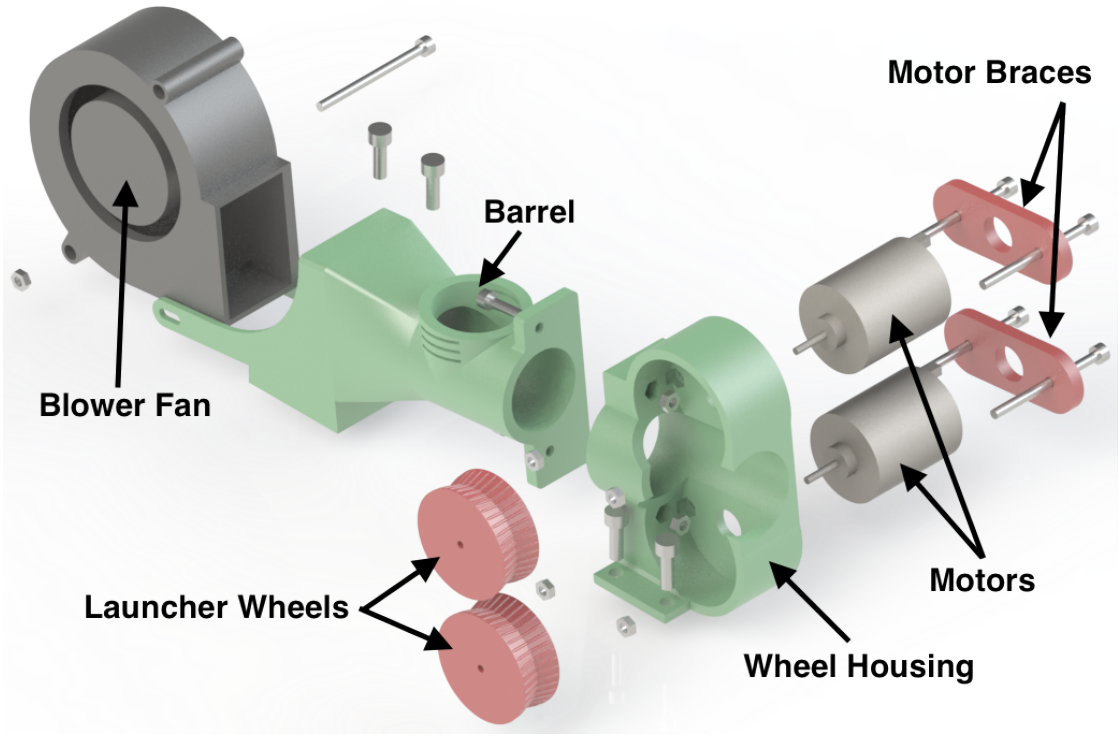


Figure 2.9: Shooting Mechanism – Exploded View

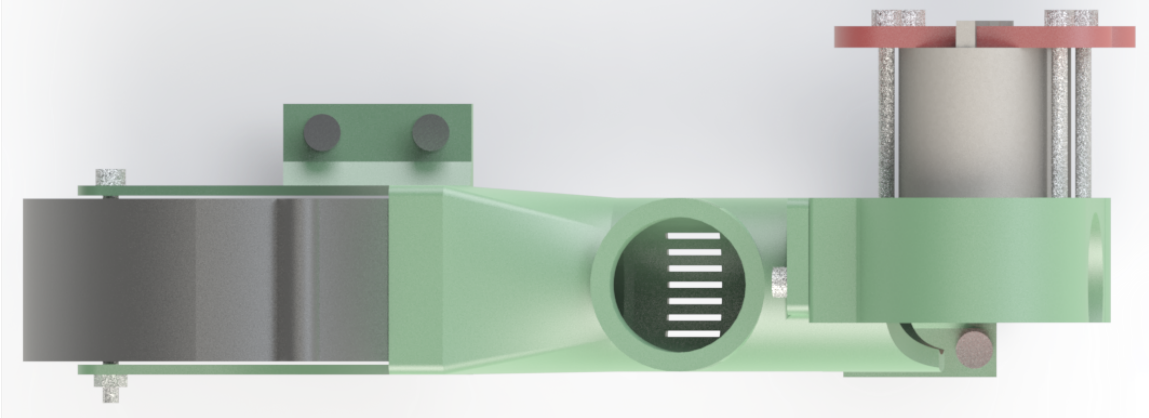


Figure 2.10: Shooting Mechanism – Top View

The **barrel** directs balls from the **ball hopper** to the **wheel housing**. First, the ball enters the barrel through a vertical chute by force of gravity. As the ball falls into the barrel, a high-pressure centrifugal (or blower) fan attached at the back of the barrel pushes it into the wheel housing inlet. As seen in Figure 2.11, the barrel slightly narrows in the area behind the top chute to prevent the ball from rolling backwards towards the blower fan. A 3D-modeling feature called a boss/base loft creates a smooth transition between the rectangular fan inlet and the circular barrel [69]. The foam balls, nominally 23 mm in diameter, would occasionally jam in a 24 mm inner diameter barrel due to ball surface imperfections, so the barrel was increased to 25 mm inner diameter. In the initial design, the pressure created by the blower fan was so high that it prevented the ball from falling down the vertical chute. The revised barrel uses strategically placed vents to reduce pressure before the ball enters the chute. As the ball travels down the chute into the barrel, it blocks the vents, increasing the pressure and forcing itself into the wheel housing.

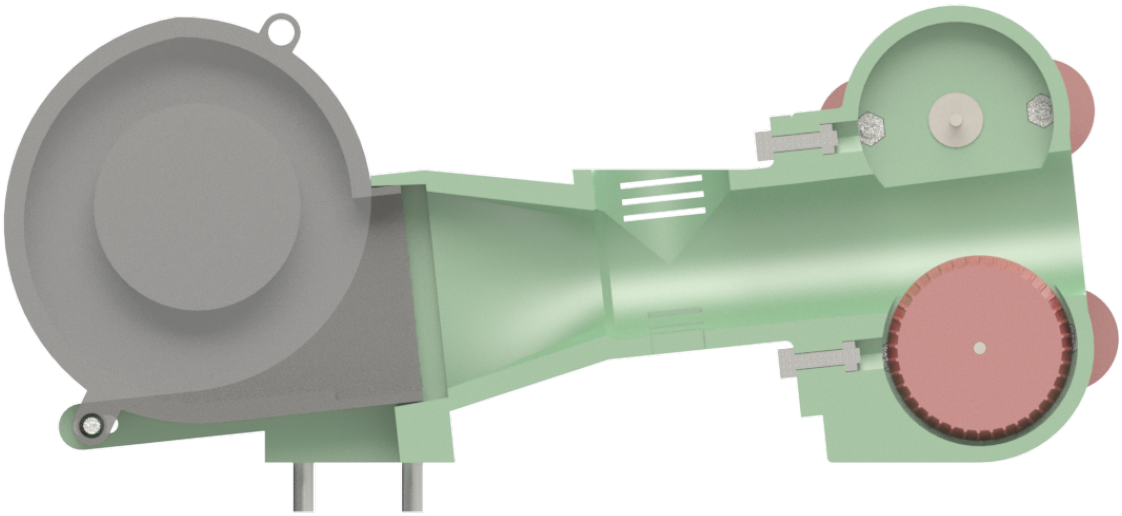


Figure 2.11: Shooting Mechanism – Cross Section View

Inside the **wheel housing**, two counter-rotating 34mm wheels press fitted to two high-speed 12 V motors rapidly accelerate the foam ball up to 50 feet per second. The 14 mm gap between wheels compresses the ball to increase grip, thereby improving energy transfer. The motors lightly press fit into the wheel housing and are secured with 3D-printed braces. The perimeter of each 3D-printed wheel, detailed in Figure 2.12, consists of a ribbed V-groove to increase the contact patch and grip with the compressed foam ball. Two "feet" with bolt holes at the bottom of the barrel and wheel housing secure the shooting mechanism to the base platform.

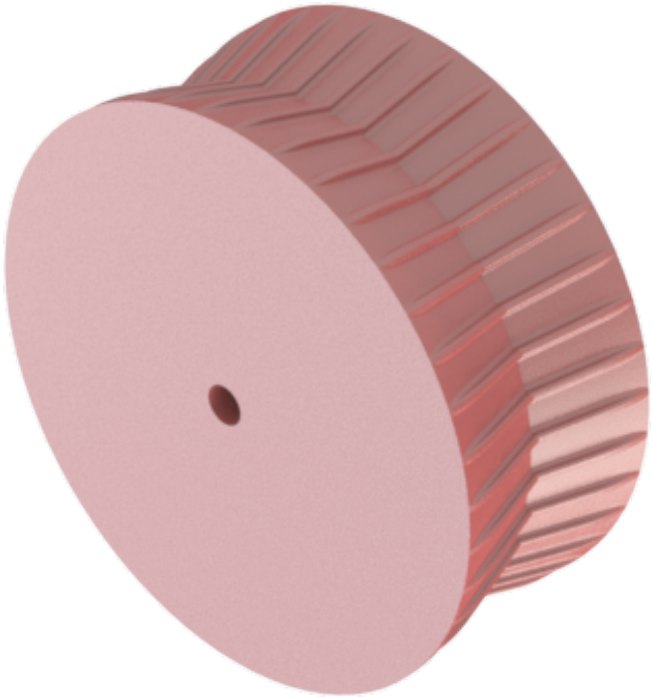


Figure 2.12: Shooting Mechanism – Shooter Wheel

The shooting mechanism performed consistently: in a trial of 100 launches, 100% of balls passed through a 6 inch diameter ring placed 6 feet away where the center of the competition net would be. Since the actual nets are 24 inches in diameter, no further testing was required. The projectile speed averaged 48.1 ft/s with 1.2 ft/s standard deviation as determined with a light gate speed measurement tool.

2.3 Ball Hopper

During the competition, the robot must obtain the foam Nerf balls from supply tubes mounted on either side of the play field. The supply tubes consist of two 1" inner diameter PVC tee joints and an eyebolt. The bottoms of the supply tubes are positioned seven inches above the floor and a swinging flap holds the balls in as shown in Figure 2.13. The ball hopper, shown in Figure 2.14 is a large 3D-printed

component designed to push the swinging flap away, collect the balls, store them, and dispense them into the shooting mechanism. Figure 2.15 shows an exploded view of the subassembly.

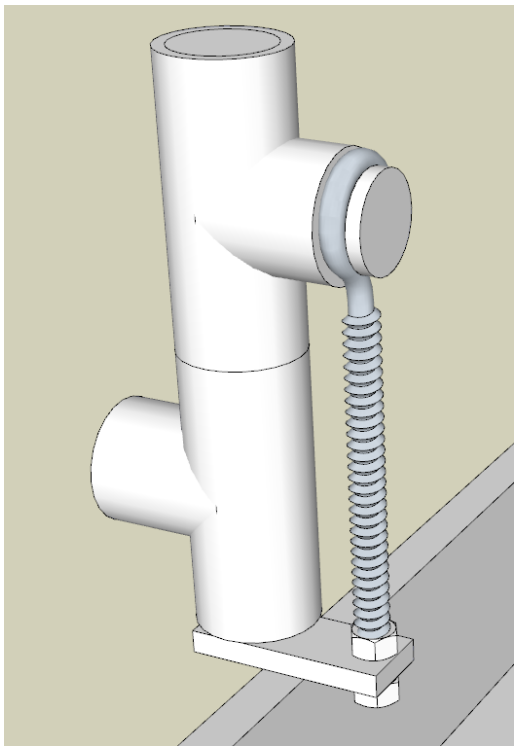


Figure 2.13: Supply Tube

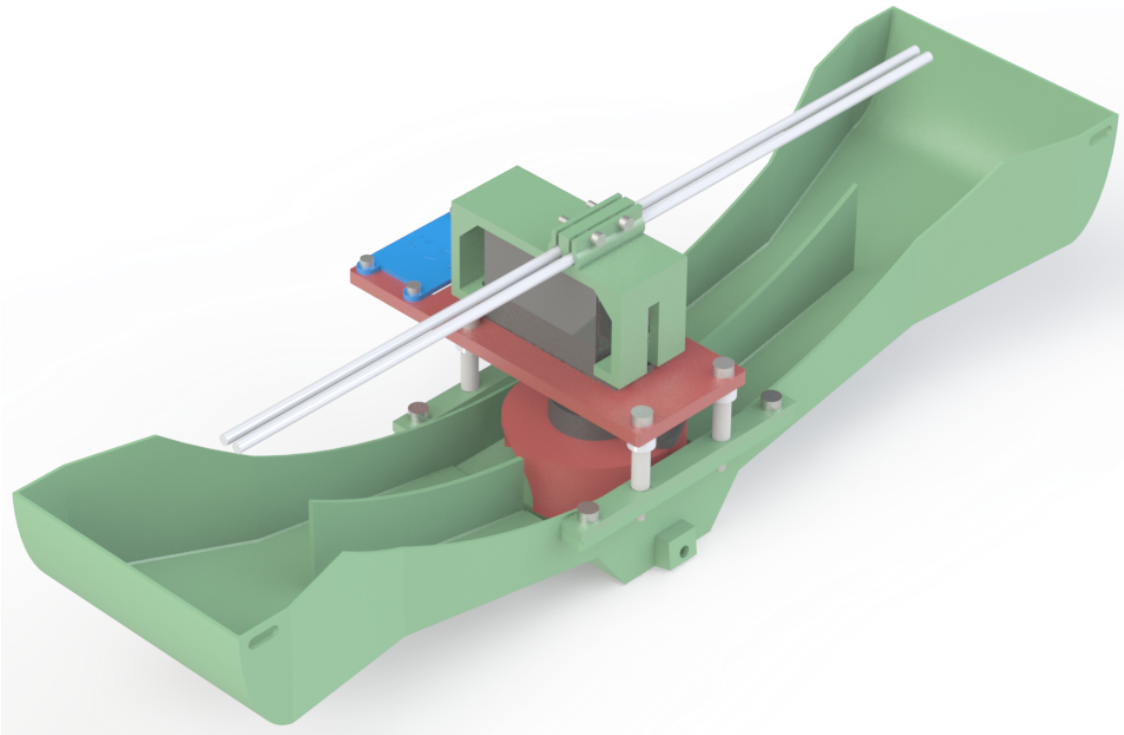


Figure 2.14: Ball Hopper

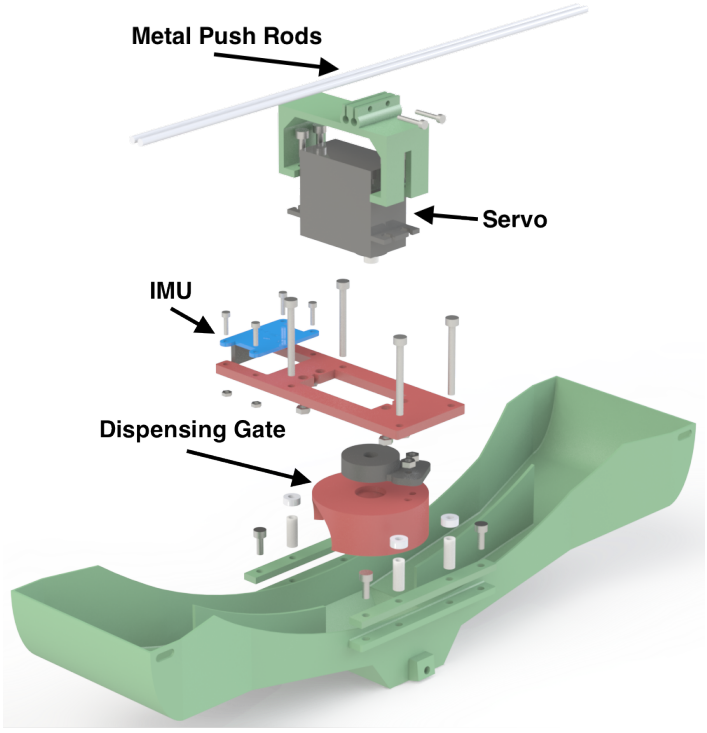


Figure 2.15: Ball Hopper – Exploded View

To obtain balls, the robot first positions itself such that the wide portion of hopper resides next to the supply tube. The robot then moves such that the metal rods at the top of the ball hopper push the eyebolt aside. As the flap swings open, the balls roll out of the supply tube and down the steep sloped portion of the hopper. Visible in the cross section view of Figure 2.16, the slope rapidly becomes shallower in order to convert the balls' downward momentum into sideways momentum which keeps balls from jamming against each other. The balls then roll into one of two channels before stopping at the dispensing gate.

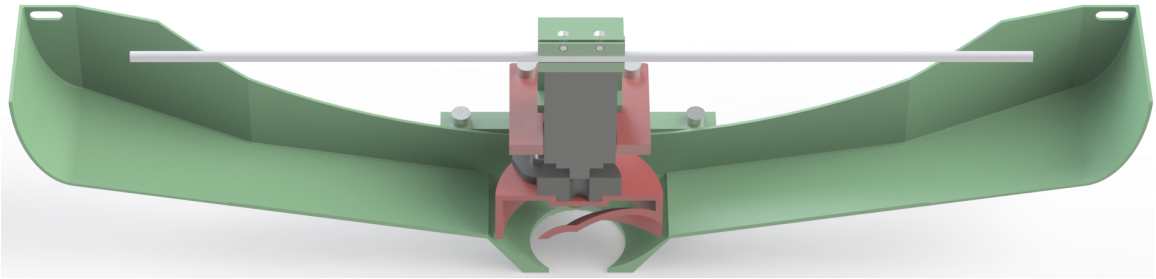


Figure 2.16: Ball Hopper – Cross Section View

The dispensing gate, shown in Figure 2.17, controls the movement of balls between hopper channels and the shooting mechanism entrance. Its complex shape directs balls into the hole at the bottom of the ball hopper from one channel at a time to prevent jamming. A standard 180° rotation servo, mounted in a 3D-printed bracket above the center of the hopper, controls the dispensing gate. Fastened to the same bracket, an inertial measurement unit (IMU) measures magnetic compass heading and acceleration in three dimensions for robot navigation purposes. The IMU is positioned at the rotational center of the robot to prevent the accelerometer from measuring robot rotation as linear movement. The ball hopper is mounted at three points: the top of the shooting mechanism and the left and right edges of the robot using 3D-printed and acrylic braces shown in Figure 2.18.

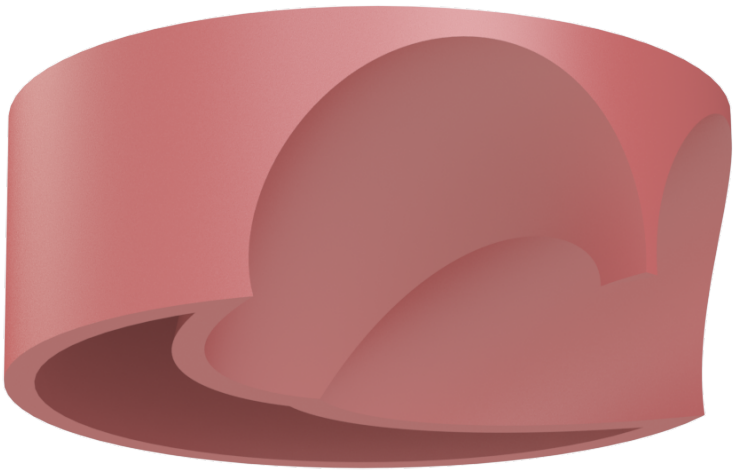


Figure 2.17: Ball Hopper – Dispensing Gate

Figure 2.18: Ball Hopper – Braces

2.4 Control Unit

The control unit, shown in Figure 2.19, consists of a 1/4" MDF board with various electronic components mounted: two off-the-shelf DC-DC switching converters, a custom interconnect printed circuit board (PCB), an off-the-shelf STM32 Nucleo-64 development board, and the Raspberry Pi microcomputer. Two 3D-printed standoffs, shown in Figure 2.20, connect the control unit to the platform and raise it slightly to avoid colliding with the robot's wheels.

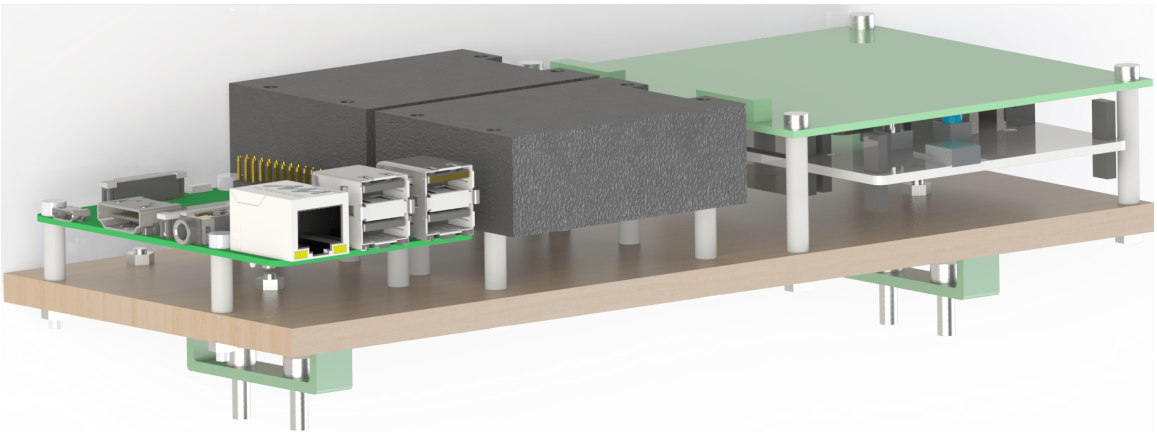


Figure 2.19: Control Unit

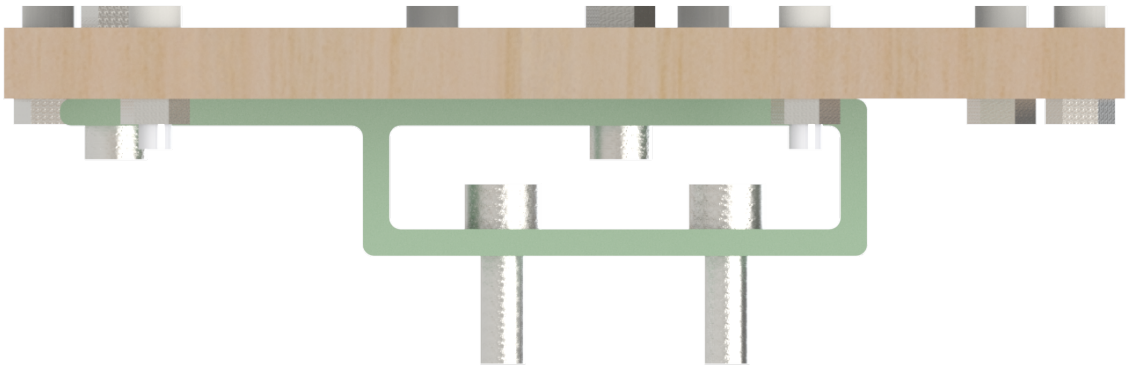


Figure 2.20: Control Unit – Standoffs

2.5 Results and Future Improvements

The fabrication and assembly of the robot matched the 3D model and parts fit together as expected. Overall, the design can be improved through shrinking components by removing excess material. The close proximity of parts to each other made debugging and troubleshooting difficult.

Due to incomplete modeling of fasteners at the wheels, the base platform interfered slightly with the protruding bolt heads. Sanding the wheel cutouts corrected the issue. Otherwise, the base platform was largely successful. The grid of holes enabled easy relocation of components. Future designs may consider adding wheel suspension to improve traction.

The shooting mechanism underwent three revisions. The first used a belt and pulley system to allow a single slow motor to drive both high-speed launcher wheels, but the large footprint prevented compatibility with other robot components. The second revision was identical to the final iteration except for narrower barrel diameter and a pivot system to allow for aim angle adjustment. After discovering the ball jamming issue, the final iteration used a widened barrel diameter and a fixed aim angle. As described above, the final shooting mechanism was 100% accurate in 100 trials and satisfied the competition speed limit. The design can be improved by modifying the barrel geometry to shrink the mechanism as it is still quite large.

The ball hopper used two revisions. The first revision used unnecessarily thick walls and extended the channel divider all the way to the edges of the hopper. It also used a shallower initial slope and a ball exit hole positioned in the center instead of offset. Balls would jam in the channel and refuse to fall down the center hole. The final design uses wider channels and a steeper slope to counter jamming problems. Future designs should consider using a single large "bucket" with an indexing agitator

similar to the Geneva drive [8].

Chapter 3

ELECTRICAL DESIGN

The electronics of the robot use a combination of off-the-shelf parts and custom designed circuits. An STM32F446RE microcontroller handles low-level hardware control and interfacing such as reading sensors and supplying motor driver control signals while a Raspberry Pi 3 Model B microcomputer processes the data within the artificial neural network and determines motor speeds and directions [54][19]. The two processors communicate through a common UART (universal asynchronous receiver-transmitter) link. The other electronic components include a battery, voltage regulation, sensors, motors, and motor drivers.

3.1 Power

A four cell, 1,800 mAH (milliamp-hour) lithium polymer (LiPo) battery powers the entire system. Polarized XT60 connectors provide mechanical reverse polarity protection. The battery voltage varies between 16.8 V when fully charged and 14.8 V when depleted so two off-the-shelf DC-DC switching regulators, shown in Figure 3.1, buck battery voltage down to produce 12 V and 7 V supplies. The high-efficiency switching regulators accept a 7 – 40 V supply and can output 1.2 – 35 V at 8 A. The 12 V bus powers the three motor drivers boards while the 7 V bus powers the STM32 Nucleo-64 development board and two AZ1085CD low-dropout linear regulators (LDO) located on the interconnect PCB. One LDO produces a 5 V bus and the other 3.3 V, each at 3 A. The LDOs possess 72 dB power supply rejection ratio, reducing the switching noise.

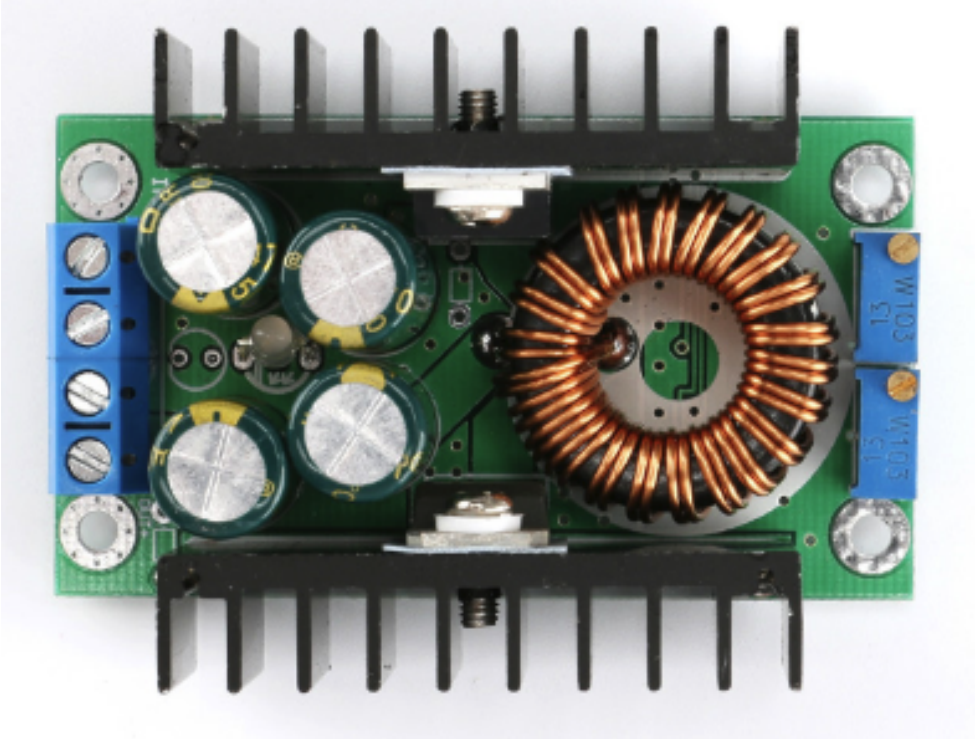


Figure 3.1: DC-DC Buck Regulator [4]

3.2 Sensors

The robot uses five off-the-shelf sensors for determining its position: four VL53L0X laser rangefinders and one Adafruit 9-DOF inertial measurement unit (IMU).

3.2.1 Adafruit 9-DOF IMU

The Adafruit 9-DOF IMU incorporates the L3DG20H gyroscope and LSM303DLHC accelerometer/compass combo on a single carrier board to allow full inertial measurement in a convenient form factor [3]. Figure 3.2 shows the IMU mounted in the 3D printed bracket. The robot uses the accelerometer and magnetic compass to realize a tilt-compensated compass but makes no use of the gyroscope. The LSM303DLHC can measure both acceleration and magnetic fields in three dimensions with configurable

bandwidth and full-scale ranges. It uses 400 kHz I^2C for control and data transfer and draws power from the 3.3 V supply.

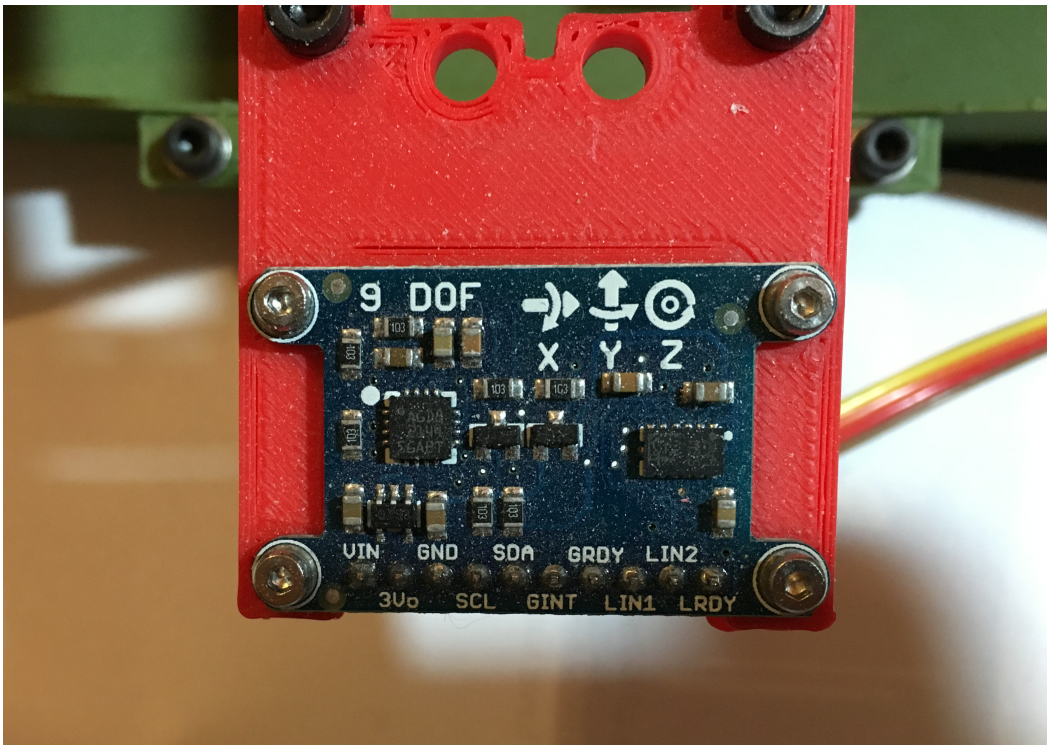


Figure 3.2: Adafruit 9-DOF IMU Mounted

Raw measurements from the IMU possess significant offset and scaling error so a calibration routine is required. The accelerometer and magnetometer calibration process uses the FreeIMU program written by Fabio Varesano [66]. The program features a graphical user interface (GUI), shown in Figure 3.3, to display accelerometer and magnetometer measurements in real time and plots them in 3D space. The calibration program was originally designed to calibrate the open-source FreeIMU IMU when connected to an Arduino with the FreeIMU calibration firmware; the program's device communication back-end was specifically modified for this application to accept the LSM303DLHC IMU connected to an STM32 microcontroller with custom firmware. The calibration algorithm assumes that the sensor's measurements are linearly distorted and therefore produces a linear scaling factor and offset for each

of six measurements (accelerometer X, Y, Z and magnetometer X, Y, Z). The correction algorithm is shown in Equation 3.1 where *val* can represent any of the six measurements.

$$val_{calibrated} = m_{scale} val_{measurement} + b_{offset} \tag{3.1}$$

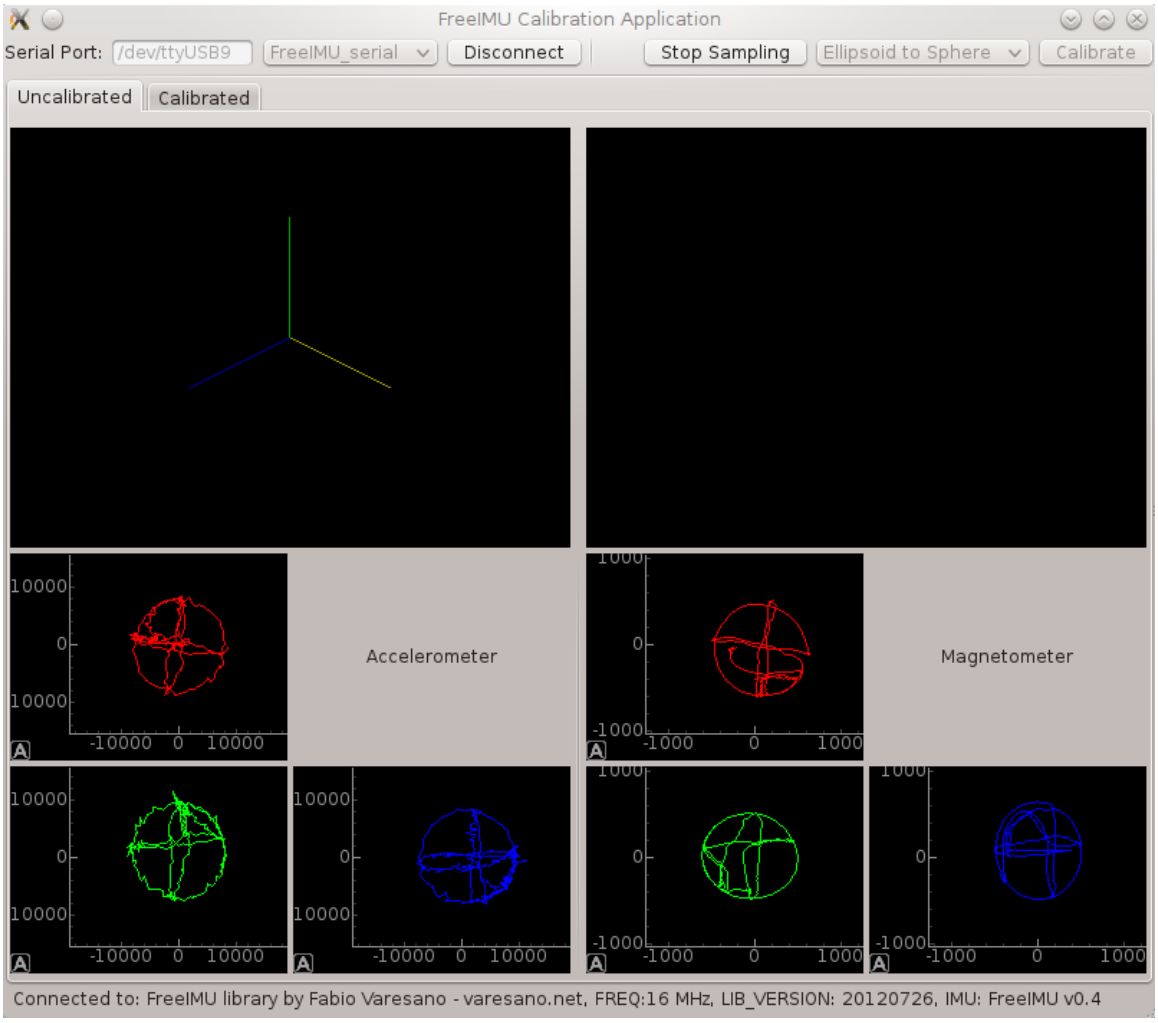


Figure 3.3: FreeIMU GUI [66]

The calibration procedure is as follows:

1. Mount IMU on fully assembled robot and connect to the microcontroller.
2. Connect microcontroller serial port to computer through Serial-to-USB converter.

3. Click "Begin Sampling" to start recording magnetometer and accelerometer measurements.
4. Point the IMU x-axis at the ground and rotate 360° around that axis. Repeat for y-axis and z-axis.
5. Point the IMU x-axis at the sky and rotate 360° around that axis. Repeat for y-axis and z-axis.
6. Repeat Steps 3 and 4 at least twice to increase data size.
7. Click "Stop Sampling".
8. Click "Calibrate" to calculate scaling and offset constants.

Since the IMU is rotated slowly, it experiences a maximum acceleration of 1 G towards the ground as well as a maximum magnetic vector whose specific magnitude and direction depends on the IMU's location on Earth. The procedure ensures that ends of each IMU axis (+X, -X, +Y, -Y, +Z, -Z) eventually receive the maximum acceleration and magnetic field [61]. To ensure hard-iron errors (such as from motors and permanent magnets) as well as soft iron errors (from local ferromagnetic materials like steel) are compensated for in the calibration, the process should be performed with the fully assembled robot and redone each time the robot is modified [39]. Since the expected fields are known (acceleration due to gravity, strength of Earth's magnetic field), the required linear transformation can be calculated [66]. Details of the exact algorithm can be found at the FreeIMU website.

3.2.2 VL53L0X Rangefinders

The rangefinders, marketed by STMicroelectronics as the "world's smallest Time-of-Flight ranging sensor", are capable of measuring between 30 and 2000 mm with a 30 Hz sample rate [57]. They operate by firing pulsed light from a vertical cavity surface emitting laser (VCSEL), measuring time taken for the laser pulse to reflect back, and calculating the distance based on the known speed of light. The robot uses cheap

$8 VL53L0X breakout boards in lieu of designing and assembling custom carriers; the sensor itself comes in a 4.4 x 2.4 mm lead-less package making hand soldering prohibitively difficult. Figure 3.4 shows the sensor board mounted in the 3D printed bracket. The sensor breakout boards include the VL53L0X module, decoupling capacitors, an LDO for the sensor's 2.8 V power supply, and level shifters for the I^2C lines. Control and data transfer both occur over 400 kHz I^2C so the breakout only requires four wires: 3.3 V, ground, and the I^2C data and clock lines. To characterize the accuracy and precision of the sensor, distance measurements were taken by targeting the sensor at a sheet of standard white printer paper. The data is compared with measurements from a tape measure; results are shown in Figure 3.5.

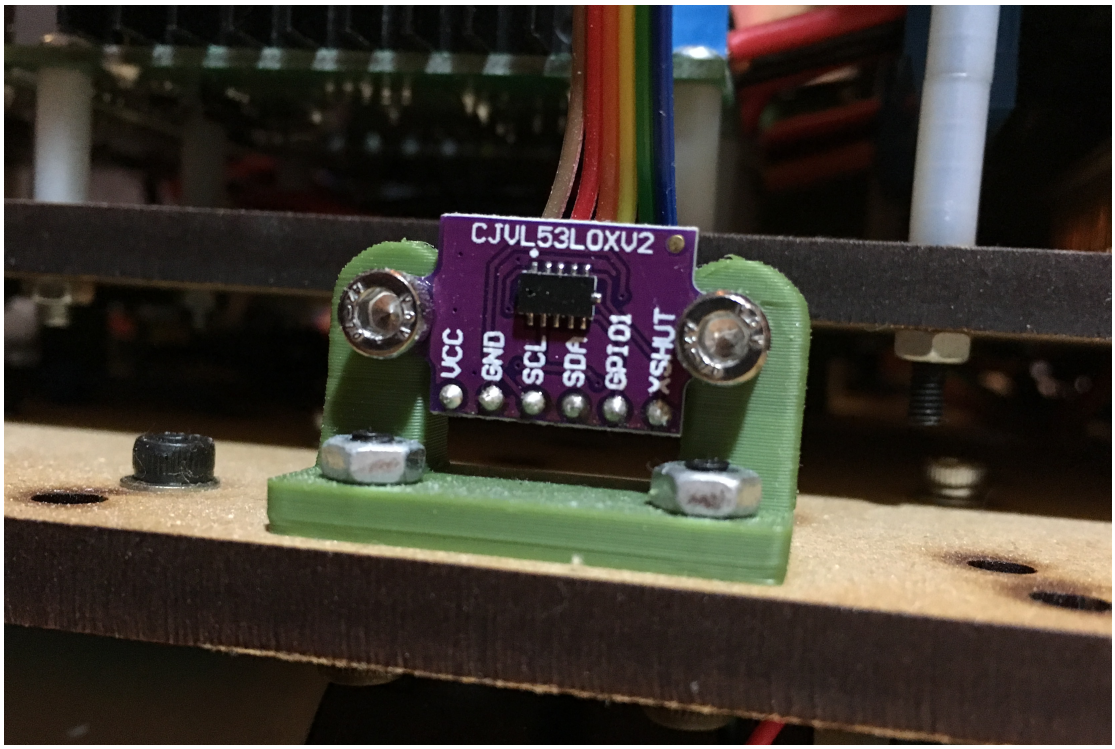


Figure 3.4: VL53L0X Rangefinder Mounted

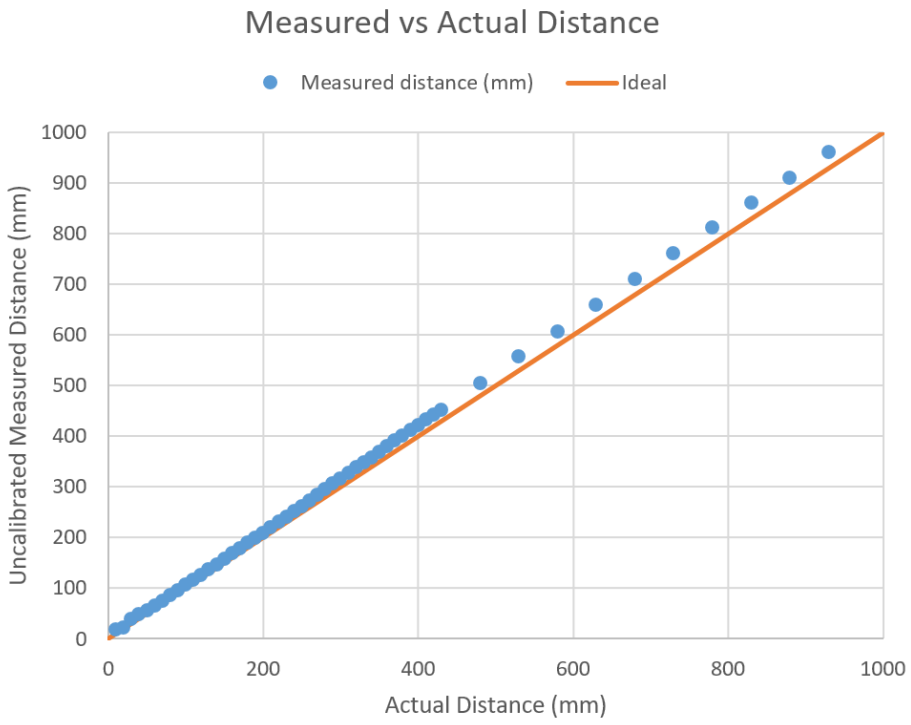


Figure 3.5: VL53L0X Measured vs. Actual Distance

Since the measurements primarily appear to possess linear error, only a linear correction is required as in Equation 3.2:

$$d_{calibrated} = m_{scale} d_{measurement} + b_{offset} \tag{3.2}$$

$d_{measurement}$ is distance as returned by the sensor, $m_{scale} = 0.96502507$, $b_{offset} = -3.8534743$, and $d_{calibrated}$ is the calibrated distance. The squared error for each data point before and after calibration is shown in Figure 3.6. The total squared error for the uncalibrated and calibrated data points are 342.3 mm and 15.0 mm, respectively, indicating the successful use of an appropriate model and constants.

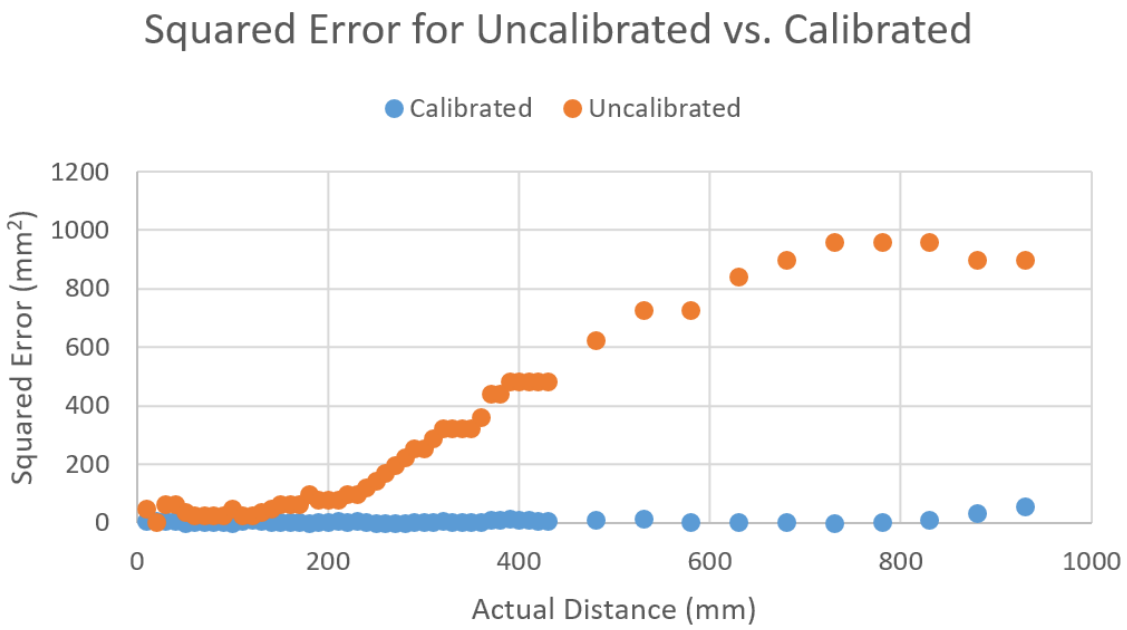


Figure 3.6: VL53L0X Squared Error for Uncalibrated vs. Calibrated

Additionally, 100 measurements were taken at each distance and the variance computed to characterize the measurement spread. The results are shown in Figure 3.7. The rangefinders are very accurate; at 900 mm with calibration, the error is only 7 mm or 0.8% error. Of course, this error is with respect to the average measurement at 900 mm. At 900 mm, the standard deviation is 14 mm so individual measurements can vary, especially with non-ideal surfaces. The closer the target, the more accurate and less varied the measurement. No experiments were carried out to determine the measurement characteristics on various colored, textured, transparent, or angled surfaces as the expected target surface is white painted wood.

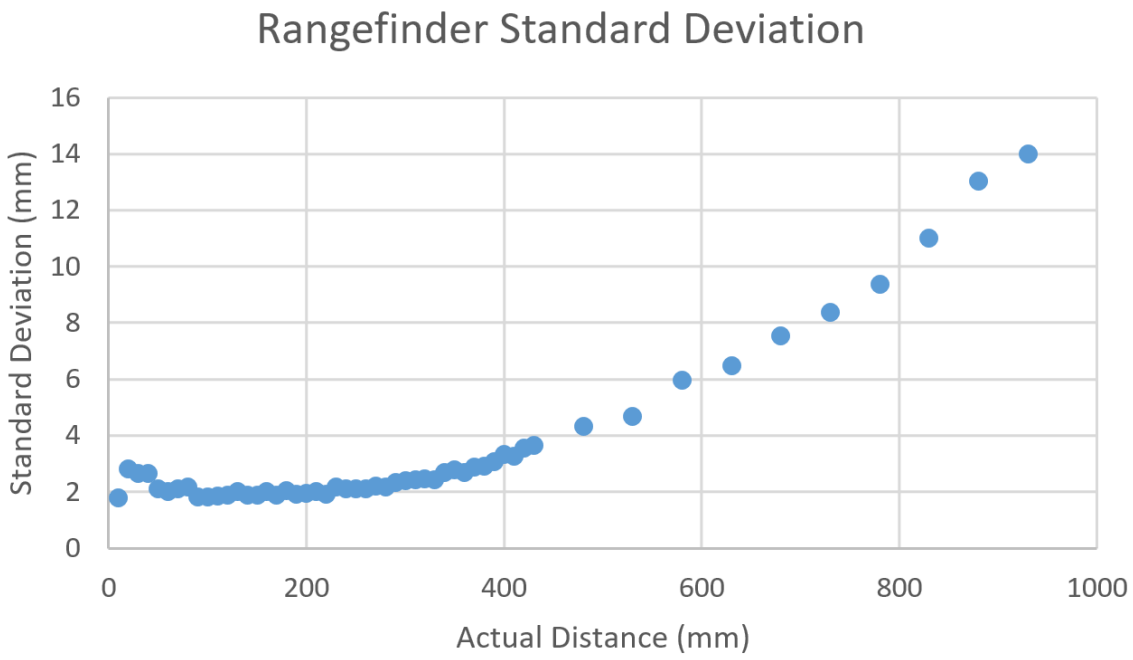


Figure 3.7: VL53L0X Standard Deviation

3.3 Motors

The robot uses four 12V Pololu 37D 70:1 geared motors to drive the wheels. The motors draw about 630 mA each at full speed steady state of 234 rpm (24.46 rad/s) and have a maximum acceleration of 4,100 rpm/s (430 rad/s^2). These values vary slightly from motor to motor due to slight differences in manufacturing and load. Each wheel experiences slightly different static normal force since the robot's weight is not evenly distributed across the four wheels. Additionally, the dynamic load varies because the ground and base platform are not perfectly flat as well as the minor effects from the robot's weight shifting due to acceleration. The motors come with quadrature rotary encoders which were used in motor characterization but not in the robot itself.

3.4 Motor Drivers

The system uses three off-the-shelf L298N motor driver boards since they are easily obtainable for less than $6 each and incorporate features such as heat-sinking, flyback voltage protection, supply filtering, and screw terminal connections [15]. Implementing comparable motor drivers with a similar feature set would undoubtedly cost more both monetarily and in design effort. The L298N dual H-bridge driver can supply 2 A maximum output per bridge using a 5 – 35 V supply. Two motor drivers handle the four robot drive motors while the third powers the blower fan and shooting mechanism motors.

Figure 3.8 shows a wiring diagram for each motor driver. The board uses four digital control inputs, each controlling the state of one half-bridge. Each motor uses a pair of inputs: IN1 and IN2 control one motor while IN3 and IN4 control the other. To achieve direction and speed control, IN1 and IN3 are pulse width modulated (PWM) while IN2 and IN4 are digitally set. Table 3.1 is a truth table of the motor state versus inputs.

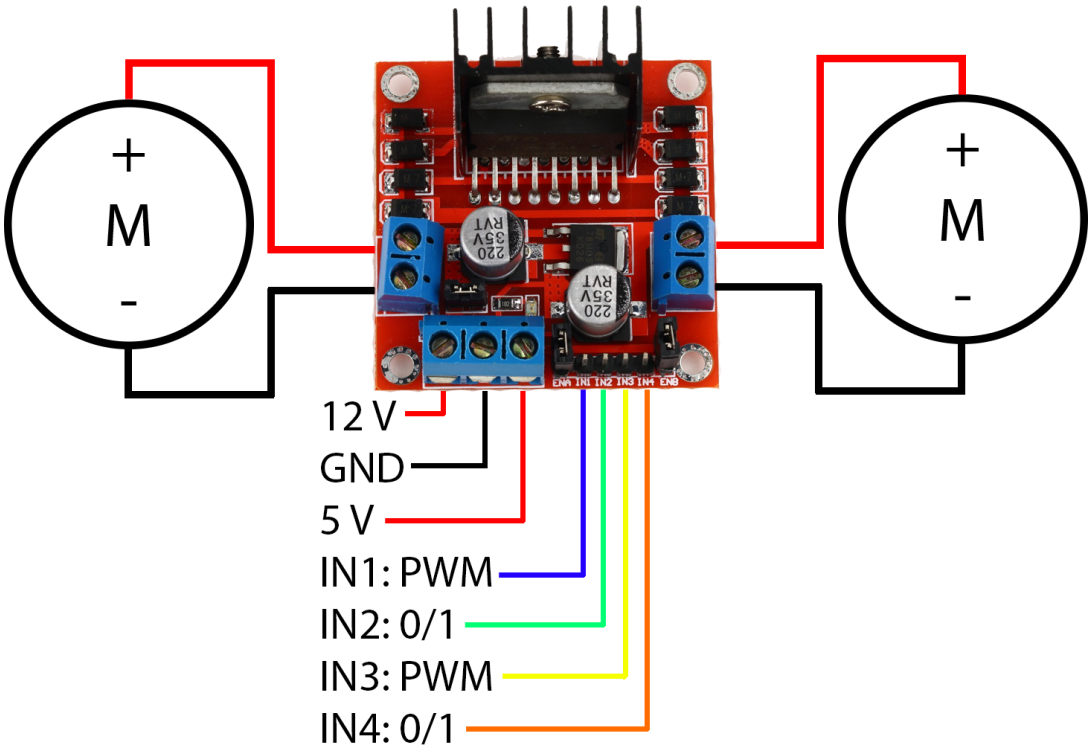


Figure 3.8: L298N Motor Driver Wiring Diagram [36]

Table 3.1: Motor Control Truth Table

IN1/IN3 Duty Cycle	IN2/IN4 State	Motor State
0%	0	Stopped
>0%	0	Forward, speed increases with duty cycle
<100%	1	Reverse, speed decreases with duty cycle
100%	1	Stopped

3.5 Servo

A GWS S03N standard servo powered from the 5 V bus actuates the gating mechanism in the ball hopper. Most servos are controlled by driving the control wire with a pulse width modulated (PWM) signal with a period of 15 – 25 ms and a pulse width between

0.5 ms and 2.5 ms where the pulse width determines the position of the servo as shown in Figure 3.9.

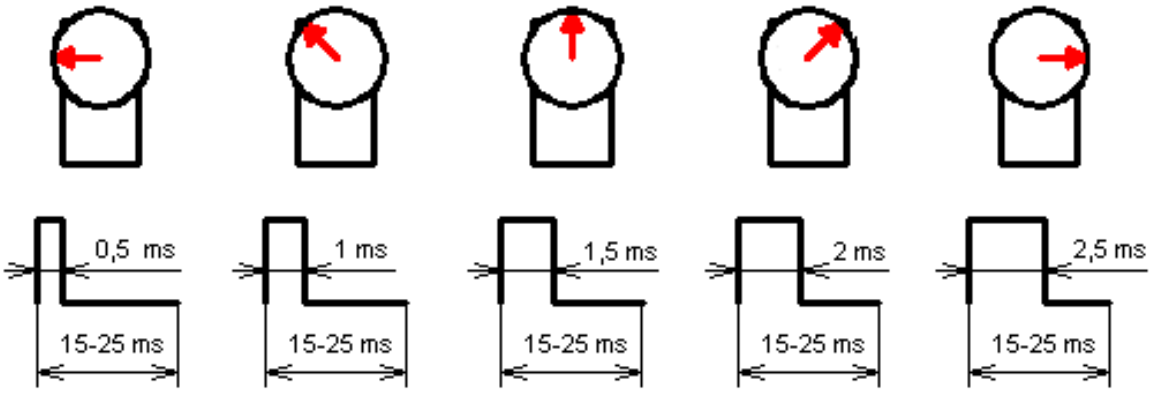


Figure 3.9: Servo PWM Control Scheme [6]

Due to loose tolerances, the actual required pulse widths can vary from servo to servo, requiring calibration to obtain accurate positional control. The process for calibration is simple: apply various pulse widths and record the resulting servo positions. The calibrated pulse widths and servo angles are recorded in Table 3.2. Clearly, the actual required timings for specific angles possess significant deviations from nominal.

Table 3.2: Servo Required Pulse Widths

Servo Position	Pulse Width (ms)
0°	0.674
45°	1.082
90°	1.490
135°	1.898
180°	2.306

3.6 Microcontroller

An STMicroelectronics STM32F446RE microcontroller (MCU) serves as the bridge between the robot's low-level electronics and the high-level control system running on a desktop computer. Specifically, the MCU collects data from sensors over I^2C and general purpose input/output (GPIO), generates control signals for the motor drivers and servo, and services commands from UART. The MCU resides on an STMicroelectronics Nucleo-64 development board, shown in Figure 3.10, which conveniently integrates an ST-LINK V2 debugger, programmer, and USB-to-UART interface. The MCU uses an ARM Cortex-M4 RISC (reduced instruction set computer) core running at 180 MHz with hardware FPU (floating point unit) and adaptive real-time accelerator. It provides numerous communication interfaces, timers, and GPIO as well as DMA (direct memory access) capability. The development board itself plugs into a board-to-board receptacle located at the bottom of the interconnect PCB.

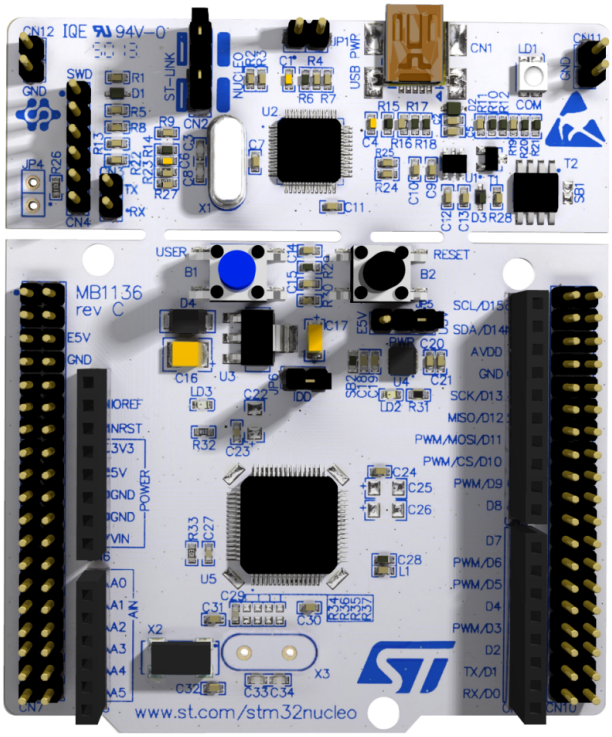


Figure 3.10: STM32 Nucleo-64 Development Board [56]

3.7 Interconnect PCB

The interconnect PCB is a custom designed board with three goals: regulate voltage, route power, and connect peripherals to the development board. The PCB routes input battery power to two external buck regulators, producing 12 V and 7 V. The 7 V bus is further regulated to 5 V and 3.3 V logic rails using two AZ1085CD LDOs and then routed to the development board, sensors, etc. Most importantly, the interconnect board provides dedicated 2.0 mm JST connectors for all sensors, motor drivers, and the servo to simplify and robustify wiring across the robot. In addition, hand-crimped cables made with JST connectors and 28 gauge ribbon cable prevent disorganized point-to-point wiring. A block diagram is shown in Figure 3.11. The board incorporates support for additional sensors to address uncertainty in the robot's final design. Table 3.3 lists the supported features as well as those used in the final

design.

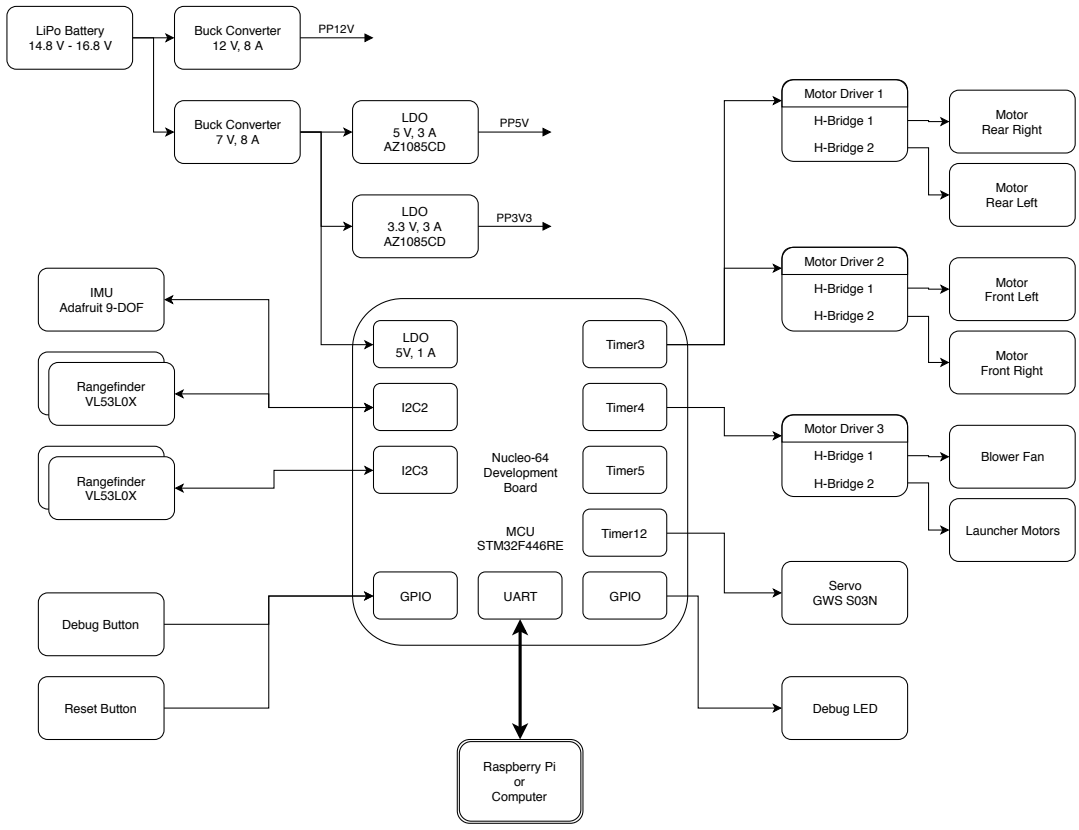


Figure 3.11: Electronics Block Diagram

Due to limited MCU GPIO availability, the board incorporates three TCA9554ADWR I^2C 8-bit GPIO expanders to handle low-speed signals [16]. However, either through an indeterminate design error, defective ICs, or poor soldering, none of the ICs would respond to I^2C traffic so they were not used in the final design. The GPIOs meant for the microswitches were repurposed to drive some signals instead.

The system uses two 400 kHz I^2C buses to improve communication bandwidth with the numerous I^2C devices. Table 3.4 lists I^2C device addresses and bus allocation. The two buses are named I2C2 and I2C3 to maintain parity with the microcontroller's naming convention. The schematic capture and board layout were created in CadSoft

Table 3.3: Interconnect PCB – Supported Features

Qty Available	Qty Used	Feature
6	4	VL53L0X laser rangefinder
1	1	Adafruit 9-DOF IMU
3	3	Dual H-bridge motor driver
2	1	5 V Standard servo
2	0	Infrared proximity array
4	0	Microswitch
4	2	Debug LED
1	1	Debug button
1	1	Reset button

EAGLE 7.4 due to its simplicity of use, popularity, and community support [7]. The schematic can be found in Appendix B, layout in Appendix C, and bill of materials in Appendix D. The board uses two 1-ounce copper layers for cost effectiveness. Two large headers underneath the board connect to the headers on the top of the microcontroller development board while the JST connectors are placed across the top of the PCB. 18-gauge wires soldered to large plated through-holes at the right edge of the board connect the external buck regulators. All components are placed on the top side of the board (except for the development board connectors) to make hand-soldering easier. The bare PCBs, fabricated by PCBWay.com, cost $23 for 10 copies [45]. The completed electronics assembly is shown in Figure 3.12.

Table 3.4: Interconnect PCB – I²C Devices

I²C Bus	Device	Address (7-bit)
I2C2	Rangefinder 4	0x52
I2C2	Rangefinder 5	0x53
I2C2	Rangefinder 6	0x54
I2C2	Adafruit 9-DOF IMU Gyroscope	0x69
I2C2	Adafruit 9-DOF IMU Accelerometer	0x19
I2C2	Adafruit 9-DOF IMU Magnetometer	0x1E
I2C2	GPIO Expander 3	0x3A
I2C3	Rangefinder 4	0x52
I2C3	Rangefinder 5	0x53
I2C3	Rangefinder 6	0x54
I2C3	GPIO Expander 1	0x38
I2C3	GPIO Expander 2	0x39

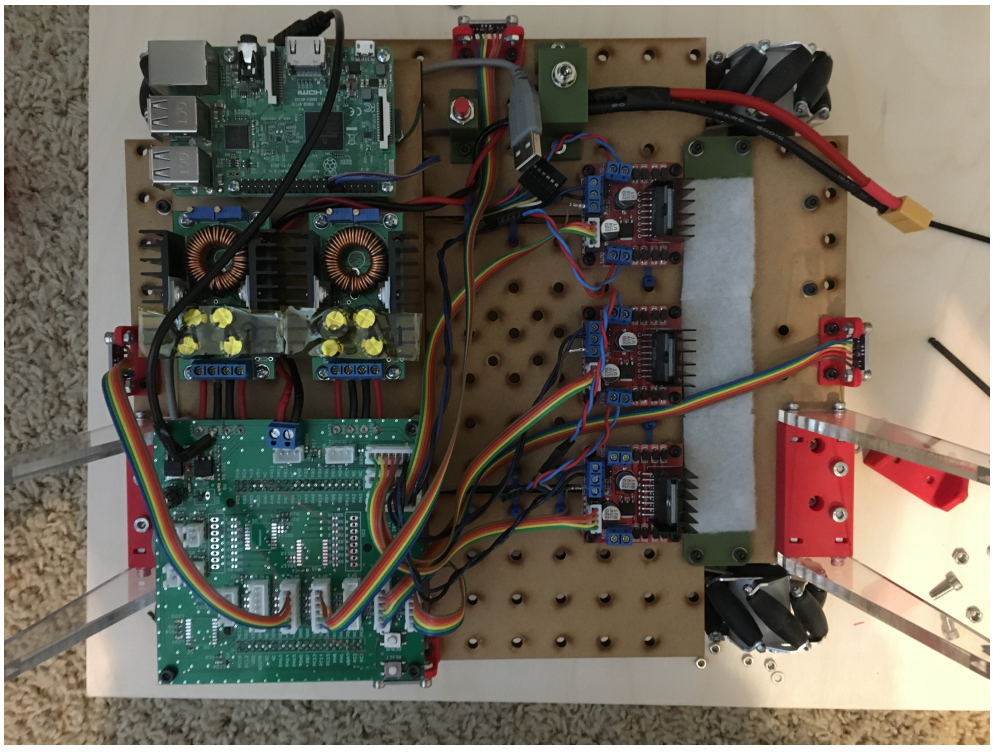


Figure 3.12: Fully Assembled Electronics

Chapter 4

FIRMWARE DESIGN

4.1 STM32CubeMX

The robot uses an STMicroelectronics STM32F446RE microcontroller (MCU) for low level sensor interfacing and motor control. Before starting the electrical design, the various hardware signals must be assigned to the MCU's GPIO with consideration for the device's peripherals such as timers and I^2C buses, a process aided by STMicroelectronics' configuration program, STM32CubeMX, shown in Figure 4.1 [53]. Within the program, the user selects which MCU to configure and a graphical representation of the chip is shown in the GUI. The MCU's GPIO pins are arranged into banks of up to 16 pins each called ports. For example, PC2 is the second pin in Port C while PB11 is Pin 11 in Port B. The left column of the GUI lists the device peripherals including analog-to-digital converters (ADCs); various buses such as SPI, USART, and USB; and timers. Peripheral settings can be chosen here such as USART mode (asynchronous, single-wire, LIN, etc) and flow control as well as timer clock sourcing and channel modes.

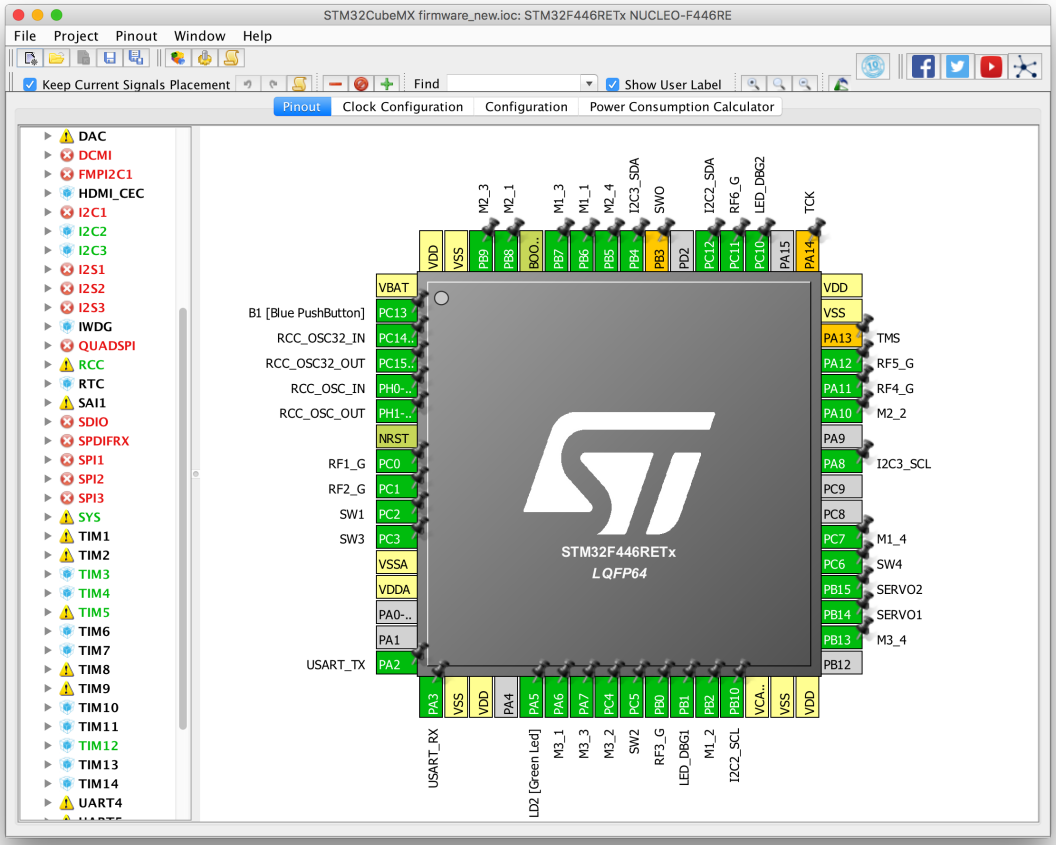


Figure 4.1: STM32CubeMX

The program greatly reduces development effort. Clicking a pin on the GUI brings up a menu of possible assignments. For example, clicking PB15, seen in Figure 4.2, shows that the pin can be used for the ADC, I2S2, as SPI2's MOSI, TIMER12's CH2 output, part of the USB differential data pair, as standard GPIO input or output, and more. As pins are assigned to various purposes, the program continually checks for compatibility issues and pin conflicts so a user can iteratively assign pins until all errors clear. For example, using a particular set of timers may actually prevent the use of I2C1 so either I2C2 or I2C3 must be used instead. This process is much faster and less error-prone than searching through the MCU's immense datasheet to manually check for assignment conflicts within every peripheral.

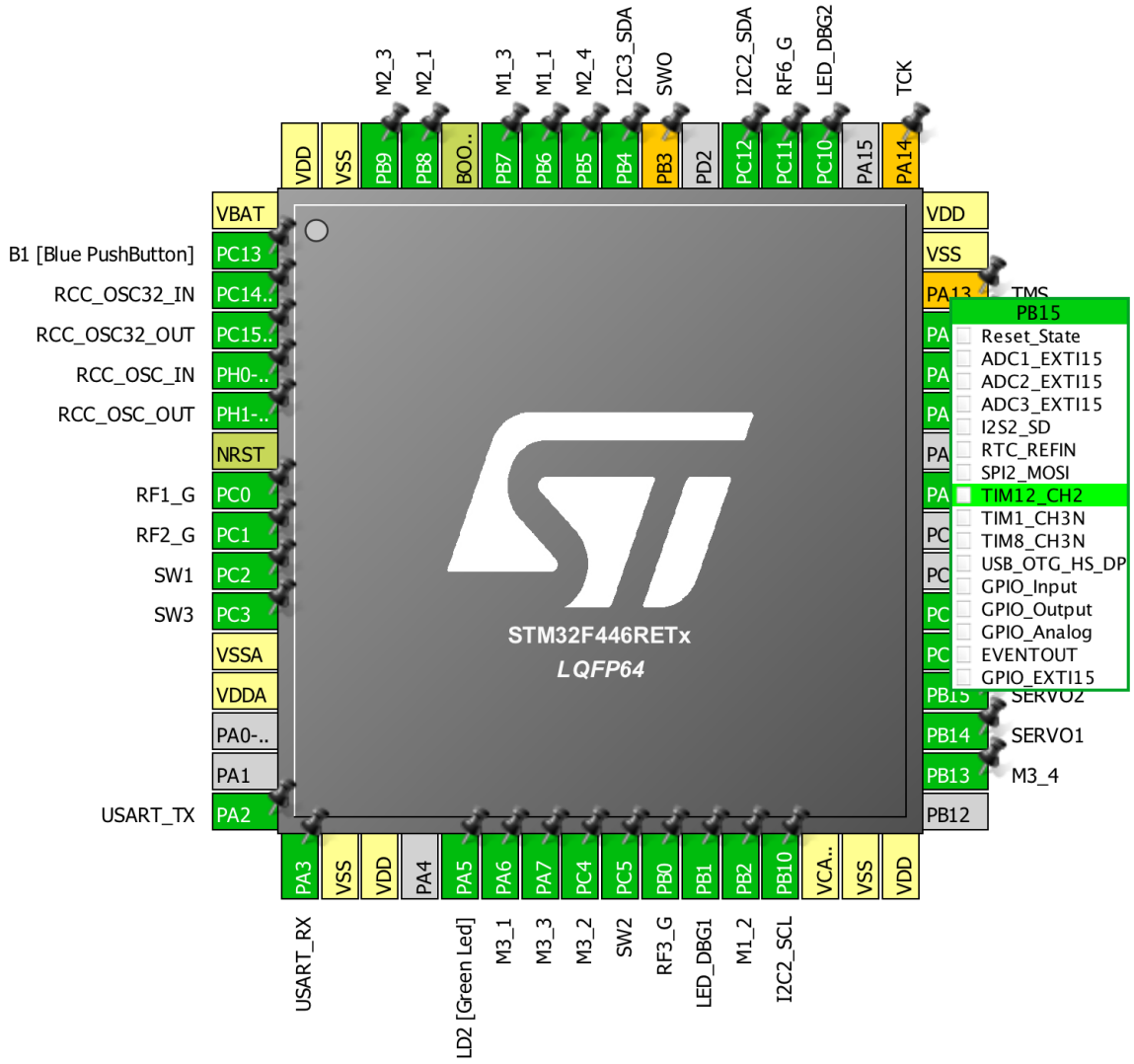


Figure 4.2: STM32CubeMX – Pin Menu

The software also provides device-wide clock and PLL configuration as shown in Figure 4.3. The MCU uses an external 8 MHz crystal to drive the internal PLL which then generates a 180 MHz system clock. Since the system is optimized for performance instead of power-saving, the advanced peripheral bus (APB) clocks run at 45 MHz and 90 MHz for APB1 and APB2, respectively [54].

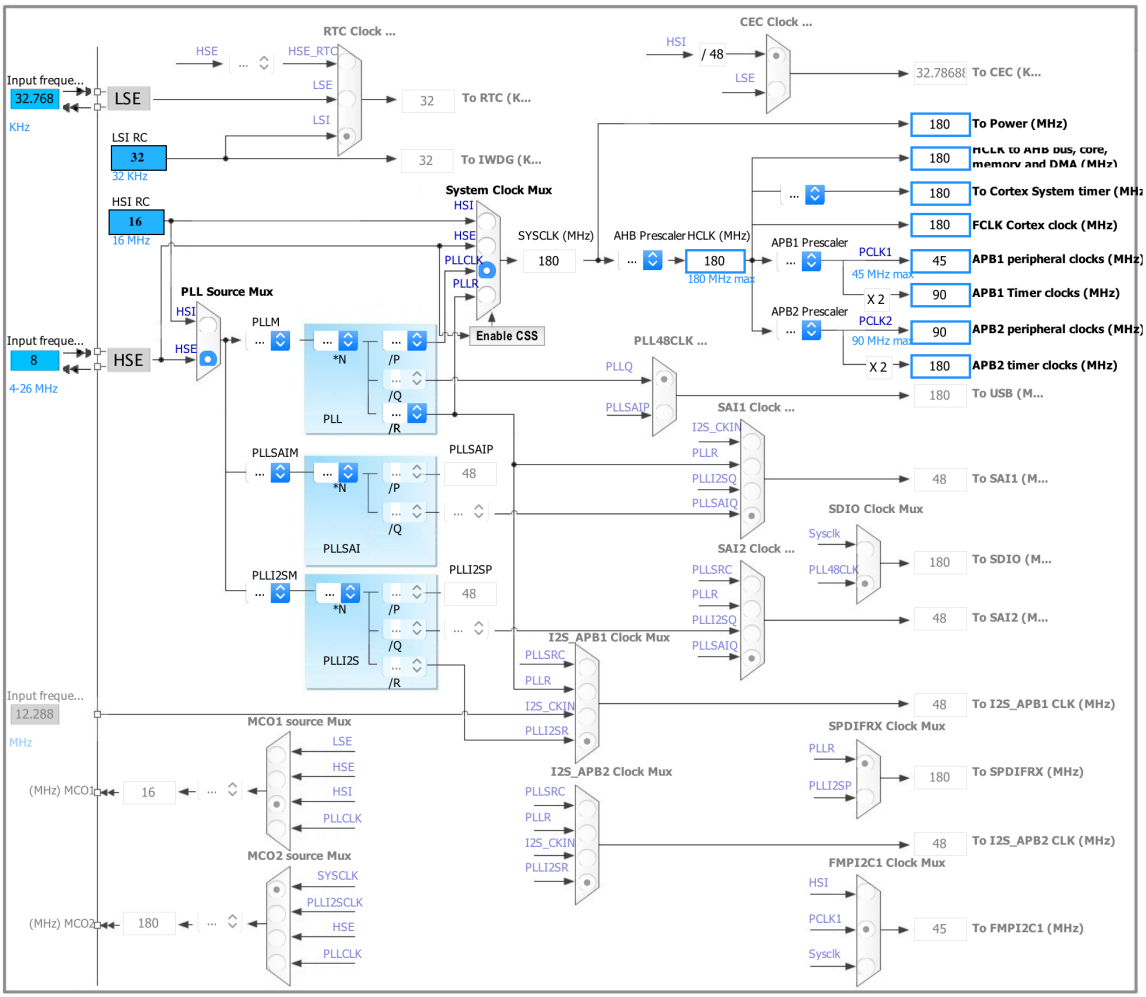


Figure 4.3: STM32CubeMX – Clock Configurator

Another tab within STM32CubeMX provides detailed peripheral configuration, shown in Figure 4.4. Both I^2C buses are set to 400 kHz with a 2:1 T_{low}:T_{high} duty cycle to meet the minimum pulse width requirement of slave devices. The USART peripheral is configured to 921,600 bits/s, 8-bit words, no parity, and one stop bit. The direct memory access (DMA) controller is set to process requests from the I^2C and UART peripherals to reduce processor load.

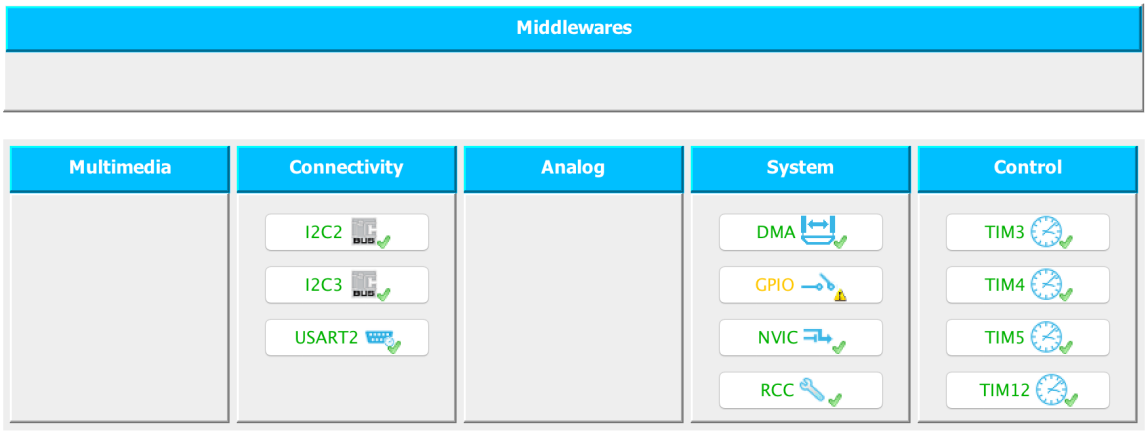


Figure 4.4: STM32CubeMX – Peripheral Configurator

The robot uses four of the MCU's timers: TIMER3, TIMER4, TIMER5, and TIMER12. Each of these timers sources its base clock from the APB1 clock which is 90 MHz. TIMER3 and TIMER4 output 6 PWM control signals for the motor drivers. They are set to count upward and reset after the counter reaches 2047 to produce a 43.9 kHz PWM frequency in the ultrasonic range. TIMER5, used for delay and timing functions, employs a clock prescaler of 9000 so the counter increases every 0.1 ms and never resets; the 32-bit counter has a maximum value of 4,294,967,295 corresponding to a counter rollover period of about 5 days. Finally, TIMER12 drives the PWM signals for servo control. It uses a 45 prescaler and a 50,000 counter period to produce a 40 Hz PWM frequency.

After configuring the various items above, STM32CubeMX can generate common boilerplate plate code to initialize and configure all the devices. Include and source files are generated on a by-peripheral basis to compartmentalize code. The program also generates a report of the device configuration, attached in Appendix E.

4.2 Microcontroller Firmware

The firmware, written in C and compiled with gcc, consists of a main file in conjunction with peripheral driver files [20]. The dependency graph is shown in Figure 4.5 with file descriptions shown in Table 4.1. STM32CubeMX automatically generated several files to which user code was added for specific robot functions. The VL53L0X API, provided by STMicroelectronics, was also modified for the STM32F446 platform. Table 4.2 lists the firmware requirements.

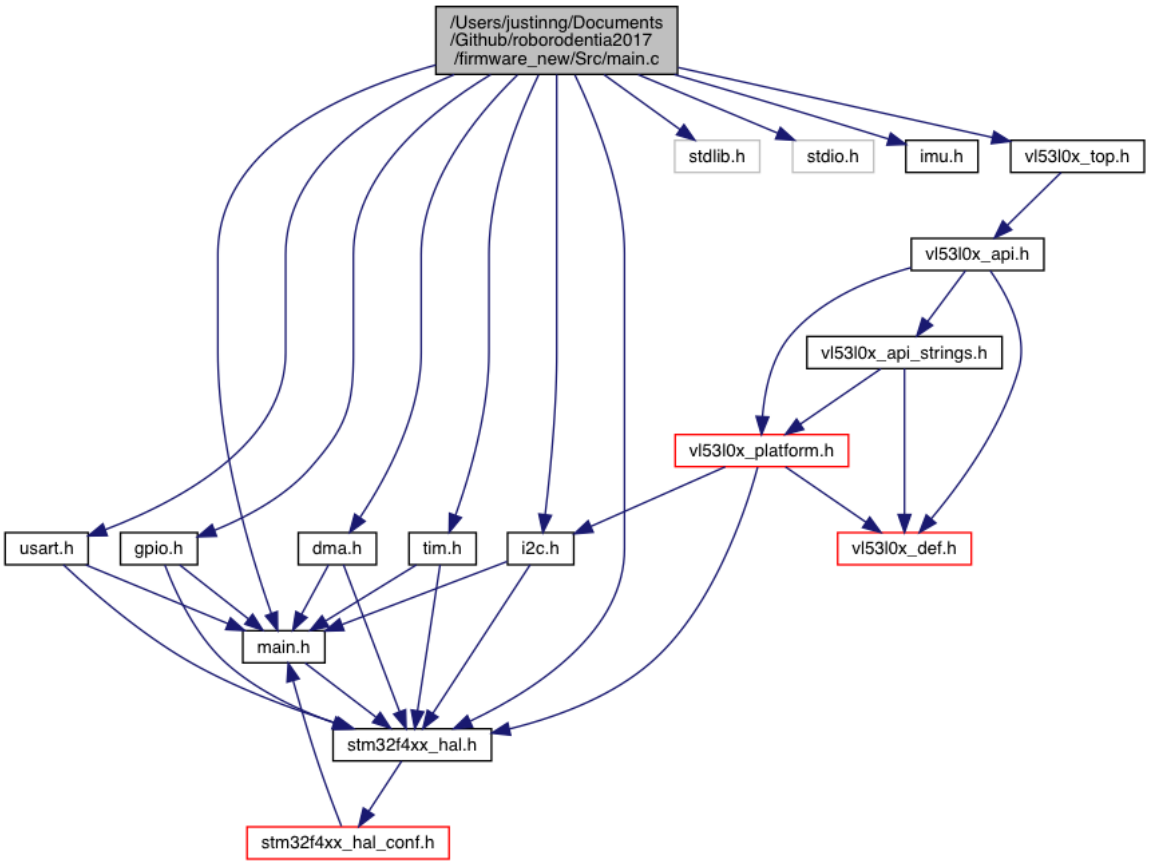


Figure 4.5: Firmware Dependency Graph for main.c

Table 4.1: Firmware – Source File Summary

File (.h/.c)	Description
main	Initialization and main loop.
stm32f4xx_hal	Hardware abstraction layer driver.
dma	Sets up DMA priorities and enables DMA interrupts.
i2c	Initializes I^2C buses, read/write, finds active devices.
tim	Initializes timers 3, 4, 5, and 12.
usart	Initializes UART, functions to write to bus, DMA callbacks.
gpio	Configures GPIO modes (in/out), speeds, and pull-ups/pull-downs.
vl53l0x_top	STMicroelectronics API for configuring and reading rangefinders.
imu	Initializes IMU; read gyroscope, accelerometer, and magnetometer.

Table 4.2: Firmware – Requirements

	Requirements
1.	Initialize I^2C sensors including rangefinders and IMU.
2.	Read and store data from sensors.
3.	Filter data from sensors to reduce noise.
4.	Generate PWM signal to drive servo.
5.	Generate control signals to motor drivers.
6.	Sequence launcher motors, blower fan, and servo to fire balls.
7.	Flash LED to indicate error state.
8.	Communicate with computer over UART link. Add'l specifications in 4.3.

4.2.1 UART Commands

The MCU responds to UART commands from a computer. The command syntax generally involves a sequence of one or more 8-bit characters. Table 4.3 lists the available commands.

UART Receiving

After receiving a character on the UART bus, the DMA places it into `rxBuffer` and fires an interrupt. The MCU enters the UART receive complete DMA callback, shown in Listing 4.1, which collects the received characters into a command string stored in `stringBuffer`. If the UART receives a new line or line return character, the callback appends a null character to the string and calls `consoleCommand()` to parse and execute the command. The define `UART_RX_WRITEBACK` controls whether received characters and commands are echoed.

Listing 4.1: UART Receive Callback

```
1  #define RX_BUFFER_MAX_LENGTH 32
2  #define UART_RX_WRITEBACK 0
3
4  uint8_t rxBuffer = '\000';
5  uint8_t stringBuffer[RX_BUFFER_MAX_LENGTH];
6  uint8_t stringBufferIndex;
7
8  void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
9  {
10     uint8_t i;
11
12     // Clear buffer
13     if (stringBufferIndex == 0) {for (i = 0; i < RX_BUFFER_MAX_LENGTH; i++)
           stringBuffer[i] = 0;}
14
15     if ((rxBuffer != 10) && (rxBuffer != 13))   //if received data different from
           ascii 13 (enter)
16     {
17         if (stringBufferIndex < RX_BUFFER_MAX_LENGTH){
18             stringBuffer[stringBufferIndex++] = rxBuffer; //add data to stringBuffer
19         }
20     }
21     else // If received data = 10 or 13
22     {
23         if (stringBufferIndex < RX_BUFFER_MAX_LENGTH){
24             stringBuffer[stringBufferIndex++] = '\0'; //add null char
25         } else {
26             stringBuffer[RX_BUFFER_MAX_LENGTH] = '\0'; //add null char
27         }
28
```

Table 4.3: Firmware – UART Commands

Syntax	Command
A	Start reading all rangefinders.
B	Returns sensor data in binary form (machine-readable).
D	Return sensor data in text form (human-readable).
I<BUS>	Scans I^2C bus and returns addresses of active devices. <BUS> valid values: 2 : Scan I2C2. 3 : Scan I2C3.
L<SIDE>	Launch balls from one side of the hopper. <SIDE> valid values: L : Launch balls from left side. R : Launch balls from right side. S : Stop all motors.
M<FL><BL> <FR>	Sets motor duty cycle calculated as <arg> divided by 2047. Each argument accepts a integer from 0 to 2047. <FL> : PWM value for front left motor. <BL> : PWM value for back left motor. : PWM value for back right motor. <FR> : PWM value for front right motor.
V	Returns firmware version info.
Z	Return debug button pressed flag.

```
29          if (UART_RX_WRITEBACK){
30              HAL_UART_Transmit(&huart2, (uint8_t *)&stringBuffer, stringBufferIndex, 0
                    xFFFF);
31              printf("\r\n");
32          }
33          consoleCommand((uint8_t *)&stringBuffer, stringBufferIndex);
34          stringBufferIndex = 0;
35      }
36
37      if (UART_RX_WRITEBACK){
38          HAL_UART_Transmit(&huart2, (uint8_t *)&rxBuffer, 1, 1);
39      }
40  }
```

UART Transmitting

The `_write()` function is overridden to redirect `printf()` output to UART using the code in Listing 4.2.

Listing 4.2: Redirect printf()

```
1  // Redirect printf to UART
2  int _write (int fd, char *ptr, int len)
3  {
4     transmitUART(ptr, len);
5     return len;
6  }
```

To avoid spending CPU cycles waiting for UART transmission to complete, the DMA buffers the character array directly to the peripheral. A flag, `txInProg`, prevents subsequent transmission until the UART transmit DMA callback, `HAL_UART_TxCpltCallback()`, clears the flag when the current transmission completes.

Listing 4.3: UART Transmit

```
1  #define TX_BUFFER_MAX_LENGTH 2000
2
3  uint8_t txBuffer[TX_BUFFER_MAX_LENGTH];
4  volatile uint8_t txInProg = 0;
5
6  // Transmit characters over UART
7  void transmitUART(char *ptr, int len)
8  {
9      // Check that the input isn't longer than our buffer
10     if (len > TX_BUFFER_MAX_LENGTH){
11         _Error_Handler(__FILE__, __LINE__);
12     }
13     // Wait until UART TX is finished
14     while(txInProg == 1){}
15     txInProg = 1;
```

```
16      // Transfer the data to be sent into the txBuffer
17      memcpy(txBuffer,(uint8_t *)ptr,len);
18      HAL_UART_Transmit_DMA(&huart2, txBuffer, len);
19  }
20
21  void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
22  {
23      // Clear TX in progress flag when complete with transfer
24      txInProg= 0;
25  }
```

4.2.2 MCU Execution Flow

The execution flow is as follows:

1. Define and initialize a struct for motor pins and timer channels.

Listing 4.4: Motor Struct

```
1   struct motor_t {
2       GPIO_TypeDef *GPIOx;
3       uint16_t GPIO_Pin;
4       GPIO_PinState PinState;
5
6       TIM_HandleTypeDef *TIM_Handle;
7       TIM_OC_InitTypeDef sConfigOC;
8       uint32_t TIM_Channel;
9   };
10
11  struct motor_t motorConfigs[4] = {
12      {MOTOR_FL_DIR_GPIO_Port, MOTOR_FL_DIR_Pin, GPIO_PIN_RESET, 0, {0},
            TIM_CHANNEL_2},
13      {MOTOR_FR_DIR_GPIO_Port, MOTOR_FR_DIR_Pin, GPIO_PIN_RESET, 0, {0},
            TIM_CHANNEL_4},
14      {MOTOR_BR_DIR_GPIO_Port, MOTOR_BR_DIR_Pin, GPIO_PIN_RESET, 0, {0},
            TIM_CHANNEL_1},
15      {MOTOR_BL_DIR_GPIO_Port, MOTOR_BL_DIR_Pin, GPIO_PIN_RESET, 0, {0},
            TIM_CHANNEL_3}};
16  struct motor_t* motorConfigsPtr = motorConfigs;
```

2. Instantiate some variables.

Listing 4.5: Communications Variables Initialization

```
1   unsigned char data_packet[21];            // Holds sensor data for UART
        transmission
2   volatile uint8_t wd_reset = 0;            // Watchdog reset flag
3   volatile uint8_t sensor_update_req = 0;      // Queue rangefinder read flag
4   volatile uint8_t dbg_btn_latch = 0;       // Debug button latch
```

3. Enable all clocks since power saving not necessary.

Listing 4.6: Enabling Clocks

```
RCC->AHB1ENR |= 0xFFFFFFFF;
RCC->AHB2ENR |= 0xFFFFFFFF;
RCC->AHB3ENR |= 0xFFFFFFFF;
RCC->APB1ENR |= 0xFFFFFFFF;
RCC->APB2ENR |= 0xFFFFFFFF;
```

4. Reset peripherals, initialize SysTick, set up PLL and clocks.

Listing 4.7: HAL and System Clock Initialization

```
HAL_Init();
SystemClock_Config();
```

5. Initialize configured peripherals.

Listing 4.8: Peripheral Initialization

```
MX_GPIO_Init();
MX_DMA_Init();
MX_I2C2_Init();
MX_I2C3_Init();
MX_TIM3_Init();
MX_TIM4_Init();
MX_TIM12_Init();
MX_USART2_UART_Init();
MX_TIM5_Init();
```

6. Reset system timer, enable UART receive DMA, initialize sensors.

Listing 4.9: UART Service and Sensor Initialization

```
TimeStamp_Reset();
serviceUART();
printf("Enabling IMU...\r\n");
IMU_begin();
printf("Initializing rangefinders...\r\n");
VL53L0X_begin();
```

7. Set servo to starting position.

Listing 4.10: Servo Default Position

```
if (HAL_TIM_PWM_Start(&htim12, TIM_CHANNEL_1) != HAL_OK){ Error_Handler(); }
```

8. Initialize variables and begin main loop.

Listing 4.11: Loop Variables and Start

```
uint32_t wd_start = 0;     // Watchdog start time
```

```
2  uint32_t cur_time = 0;      // Current time
3  uint8_t wd_en = 0;          // Watchdog enable
4  data_packet[20] = '\n';     // 21 byte sensor data packet to send over UART
5
6  while (1)
7  {
```

9. Watchdog prevents motors from running when UART is inactive. Receiving any UART command sets `wd_reset` to 1.

Listing 4.12: Motor Watchdog

```
1     // Reset watchdog start time if flag is set
2     if (wd_reset == 1){
3        wd_start = TimeStamp_Get();  // Units of 0.1 ms based on Timer5
4        wd_en = 1;
5        wd_reset = 0;
6     } else if (wd_en == 1){
7        // Trigger if watchdog is enabled and expires
8        cur_time = TimeStamp_Get();
9        if ((cur_time - wd_start) > WD_LEN){
10          wd_en = 0;
11
12          // Disable motors
13          int i;
14          for (i=0; i<4; i++) {
15             motorConfigs[i].PinState = GPIO_PIN_RESET;
16             motorConfigs[i].sConfigOC.Pulse = 0;
17          }
18          for (i=0; i<4; i++) {
19             // Alter the PWM duty cycle and start PWM again
20             if (HAL_TIM_PWM_ConfigChannel(motorConfigs[i].TIM_Handle, &
                   motorConfigs[i].sConfigOC,
21                motorConfigs[i].TIM_Channel) != HAL_OK) { Error_Handler(); }
22             if (HAL_TIM_PWM_Start(motorConfigs[i].TIM_Handle, motorConfigs[i].
                   TIM_Channel)
23                != HAL_OK){ Error_Handler(); }
24
25             // Set the direction pin
26             HAL_GPIO_WritePin(motorConfigs[i].GPIOx, motorConfigs[i].GPIO_Pin,
                   motorConfigs[i].PinState);
27          }
28       }
29    }
```

10. Read debug tactile button state and latch it until read by UART.

Listing 4.13: Debug Button Latch

```
1     // Read debug button
2     dbg_btn_latch = HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) && dbg_btn_latch;
```

11. If requested over UART, read all four rangefinders and store the data in the `data_packet`.

Listing 4.14: Rangefinder Read

```
1     if (sensor_update_req == 1){
2        int i;
3        for (i = 0; i < 4; i++){
4           if (rangefinderRead(i)){
5              data_packet[i*2  ] = (rangeData[i] >> 8) & 0xFF;
6              data_packet[i*2+1] =  rangeData[i]       & 0xFF;
7           }
8        }
9        sensor_update_req = 0;
10    }
```

12. Read the magnetometer and store the data in the data packet.

Listing 4.15: Magnetometer Read

```
1     magnetometer_read();
2     data_packet[ 8] = (magData.x >> 8) & 0xFF;
3     data_packet[ 9] =  magData.x       & 0xFF;
4     data_packet[10] = (magData.y >> 8) & 0xFF;
5     data_packet[11] =  magData.y       & 0xFF;
6     data_packet[12] = (magData.z >> 8) & 0xFF;
7     data_packet[13] =  magData.z       & 0xFF;
```

13. Read the accelerometer and store the data in the data packet.

Listing 4.16: Accelerometer Read

```
1     accelerometer_read();
2     data_packet[14] = (accelData.x >> 8) & 0xFF;
3     data_packet[15] =  accelData.x       & 0xFF;
4     data_packet[16] = (accelData.y >> 8) & 0xFF;
5     data_packet[17] =  accelData.y       & 0xFF;
6     data_packet[18] = (accelData.z >> 8) & 0xFF;
7     data_packet[19] =  accelData.z       & 0xFF;
8  }
```

14. Return to start of loop.

4.2.3 Error Handler

During errors or faults, the MCU enters the error handler, prints an error message, and flashes the debug LED at 5 Hz.

Listing 4.17: Error Handler

```
1  void _Error_Handler(char * file, int line)
2  {
3     printf("Entered error handler.\r\n");
4     while(1)
5     {
```

```
6         HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_1);
7         HAL_Delay(100);
8     }
9 }
```

Chapter 5

REINFORCEMENT LEARNING

The robot controls its motors using artificial neural networks (ANN) and the deep deterministic policy gradient (DDPG) algorithm, developed by Lillicrap et al. [30]. The system runs in Python 3 [47], leaning heavily on the TensorFlow library with GPU support [62] for artificial neural network instantiation, updating, and saving. After training, the system actuates each of the four drive motors to move the robot to a desired position and orientation in the Roborodentia field while minimizing the velocity and effort applied (i.e. energy spent). Two different approaches, the "three actor" and "single actor" variants, are implemented and compared. Before delving into the implementation and results, the following sections briefly cover reinforcement learning (RL) techniques that form the foundation of DDPG. It is worth noting that Sutton's introductory book on RL provides excellent background on many of the following algorithms and more [60].

5.1 Reinforcement Learning Background

Reinforcement learning (RL) is a subset of machine learning that aims to solve control and action selection problems rather than to perform classification or data clustering. In other words, the concern lies in determining which actions an actor should take in an environment to receive the greatest reward. Most reinforcement learning problems involve six main elements: an actor, the environment, rewards, a policy, a value function, and sometimes a model. An actor (sometimes called an agent) takes actions in an environment which returns a state and reward, illustrated in Figure 5.1. The actor's primary goal is to maximize the accumulated reward received from the

environment. A policy determines which actions the actor takes given the current state and can be considered a mapping from state to action. The value function indicates the long-term reward expected from a state for all possible states. To compared reward with value, even if a state only has a small immediate reward, it may possess high value since it leads to future high reward states. Some algorithms involve a model of the environment, allowing prediction of the next state from the current state and action. Table 5.1 summarizes some common terms and definitions used in the reinforcement learning literature.

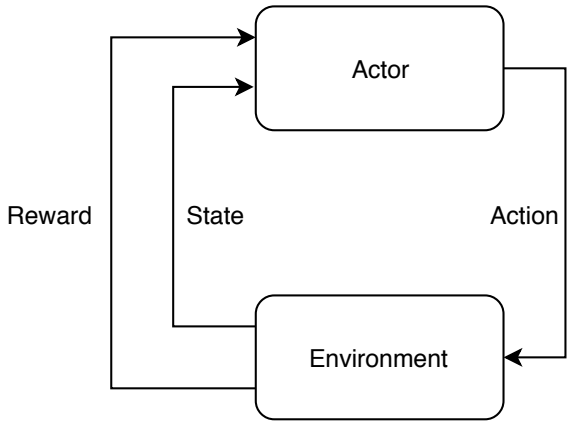


Figure 5.1: Actor-Environment Feedback Loop

To better conceptualize these terms, consider a game of tic-tac-toe as shown in Figure 5.2. The **environment** is the game itself, including the rules and game board. The two **actors** (players) sequentially take the **action** of placing marks in the boxes. The actions are encoded into some numeric form as defined by the designer. For example, $a = 0$ might represent putting an X in the top left corner while $a = 9$ might mean an X in the bottom right corner. Winning the game would grant a good **reward** while losing might yield a bad reward. Note that the descriptors "good" and "bad" are deliberately vague as numerous valid implementations of the reward exist. Conceptually, the reward allows the actor to differentiate between desirable and undesirable actions and states. Each of the nine spaces can take one of three

values (blank, X, or O) so the game has $3^9 = 19,683$ possible **states** (although some states are not achievable as the game would end once a player gets three in a row). Like the action, the rewards and states are represented numerically.

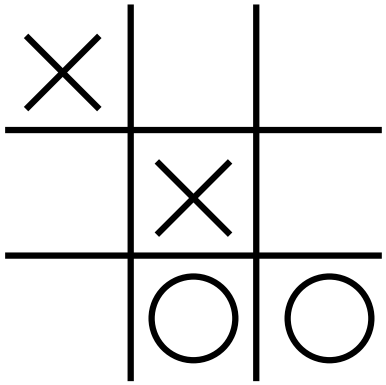


Figure 5.2: A Game of Tic Tac Toe

5.1.1 Reward

Actors strive to maximize the long-term discounted reward, G_t, where the subscript t denotes the time step [60]. In RL problems with a definitive end (denoted by $t = T$) such as a game of chess, G_t is finite and can be calculated simply as the sum of all future rewards (5.1).

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T = \sum_{k=t}^{T-1} R_{k+1} \tag{5.1}$$

However, continuous problems, such as maintaining the temperature of a refrigerator, have no maximum time step and therefore, a possibly infinite G_t. Shown in Equation 5.2, the introduction of a discount factor, γ, makes such a situation manageable. The discount factor weights the worth of future rewards exponentially by their distance into the future and ranges from 0–1 where 0 completely devalues future rewards while a discount factor of 1 weights all future rewards equally. In practice, the discount factor is less than 1 in order to allow G_t to converge [60]. The discount factor can be considered a knob that sets the algorithm's outlook between myopic

(short-sighted, only values immediate rewards) and far-sighted (willing to sacrifice immediate reward for greater long-term returns).

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{5.2}$$

5.1.2 Value Functions

The **value** function $V_\pi(s)$ is the expected long-term reward G_t as a function of the state s under a certain policy π (5.3). The $\mathbb{E}_\pi$ operator evaluates the expected value provided the actor follows policy π.

$$V_\pi(s) = \mathbb{E}_\pi[G_t|s] \tag{5.3}$$

Related but slightly different, the **Q-function** or **action-value** function $Q_\pi(s, a)$ is the expected long-term reward G_t as a function of the state s **and action** a under a certain policy π (5.4). The important distinction from the value function is dependency on the action. Since the Q-function uses three variables (s, a, and Q), it can be plotted in three dimensions as in Figure 5.3. Again considering tic tac toe, if $s = 0$ represents the starting, i.e. blank, state of the game, the red slice of Figure 5.3 represents the Q-values or expected long-term reward for each of the nine possible starting actions. Note that the actual plot of tic tac toe's Q-function would use discrete points rather than the continuous function shown. The Q-function is defined for all possible states and actions so tic tac toe contains $19,683$ states $\cdot$ 9 actions $= 177,147$ discrete Q-values. Notice that the Q-function is non-obvious, highly non-linear, and may be continuous or discrete depending on the specific problem.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s, a] \tag{5.4}$$

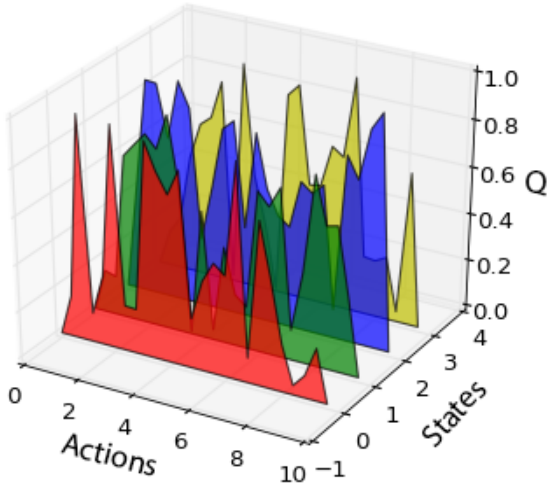


Figure 5.3: 3D Q-Function Example

Table 5.1: Reinforcement Learning Terms and Definitions [18][23][34][60]

Term	Definition
Action	Performed by the actor in the environment to move to the next state.
Actor/Agent	Chooses actions based on a policy.
Credit assignment problem	The challenge in determining which action was responsible for the long term reward.
Critic	Evaluates the policy.
Discount factor (γ)	A lower discount factor reduces the value of long-term rewards, favoring more immediate rewards.
Environment	The various states and associated rewards through which the actor navigates.
Episode	A sequence of actions, states, and rewards from initial to terminal state.

Term	Definition
Experience replay	Training technique in which transitions sampled randomly from past transitions are used to update the network in order to break up data correlation and stabilize convergence.
Feature	Numerical-valued input to an artificial neural network.
Greedy	A policy which chooses actions to maximize the immediate next reward.
Learning rate	A hyper-parameter determining how quickly network weights are adjusted.
Model	Allows prediction of the next state from the current state and action.
Model-based	Algorithm learns a model of the environment and optimizes its policy based on the model.
Model-free	Algorithm optimizes its policy without requiring a model of the environment.
Policy (π)	The strategy by which the actor chooses its actions, e.g. random, exploratory, greedy.
Off-policy	Values are learned based on a policy independent from the one used by the actor.
On-policy	Values are learned based on the current policy used by the actor.
Reinforcement learning	Training examples are accompanied by time-delayed rewards, actor learns to maximize its long-term reward in an environment.

Term	Definition
Reward/return (r)	A number emitted by the environment denoting "goodness" of the state.
Supervised learning	Training examples are accompanied by the desired label or outcome, actor learns to produce the correct label, classification.
State (s)	A representation of the situation in the environment.
Transition	A set of state, action, reward, and next state for a single time step.
Unsupervised learning	Training examples do not have associated labels, actor finds hidden patterns within data, clustering.
Q-value (Q)	The expected long-term discounted reward of a state and action.
Value (V)	The expected long-term discounted reward of a state.

5.1.3 Policies

Various policies exist but the most commonly referred to in the literature include optimal, greedy, ϵ-greedy, and random. The optimal policy π_*, by definition, produces greater or equal value for every possible state than any other policy π, expressed in Equation 5.5, where v_{π_*} represents the value function of the optimal policy, v_π is the value function of any other policy π, and $\mathcal{S}$ is the set of all possible states. An actor under the random policy always takes random actions. The greedy policy directs the actor to take the action with the highest value or Q-value depending on implementation. Although greediness does lead to maximizing reward, an actor following such a policy may not sufficiently explore the environment, i.e. try new things to discover better actions or states. To allow a balance between exploration

(discovering better rewards) and exploitation (obtaining the most reward), the ϵ-greedy policy tells an actor to pick the greedy action with probability $1 - \epsilon$ and a random action with probability ϵ where ϵ ranges between 0 and 1.

$$v_{\pi_*}(s) \geq v_{\pi}(s), \forall s \in \mathcal{S} \tag{5.5}$$

5.2 Reinforcement Learning Algorithms

Many reinforcement learning algorithms have been applied to action selection and control problems including Q-Learning, State-Action-Reward-State-Action (SARSA), deep Q-network (DQN), policy gradients (PG), deep policy gradients (DPG), and deep deterministic policy gradients (DDPG), among others [60][59][52][51][30]. Note that these methods differ from evolutionary algorithms in that they actively learn as they interact with the environment.

RL algorithms can be classified by their use of models (model-based vs. model-free) and actor policy (on-policy vs. off-policy) [60]. Model-based strategies first develop a model of the environment and then use a planning algorithm along with the model to create a controller while model-free algorithms forgo the model entirely. On-policy strategies learn the policy followed by the actor while off-policy techniques learn a policy different from the one followed by the actor. For example, in Q-learning, the Q-function learns the greedy policy even if the actor is following the random policy [60].

5.2.1 Q-Learning

Q-Learning is an off-model, off-policy algorithm that estimates the Q-function of the environment [60]. The Q-function represents the "quality" of every possible action in every possible state. Given a particular state, the Q-function returns the quality,

i.e. predicted goodness, of each possible action the player could take. Therefore, the optimal action to take is simply the one with the highest Q-value.

Clearly, the Q-function always exists but not necessarily in an analytic or obvious form. So-called tabular Q-learning methods use a 2-D matrix to represent the Q-function with rows for all possible states and columns for all possible actions [35]. The table is initialized with guesses (possibly randomly or with all elements set to 0) and iteratively updated to approximate the true Q-function Q^* using the Bellman equation (5.6). The equation states that the expected long-term reward Q for state s and action a is equal to the reward received from being in state s plus the discounted (γ) maximum Q of the next state s'. For clarity, the $\max_{a'}(\cdot)$ operator chooses a' to maximize its argument $(\cdot)$ so $\max_{a'}(Q(s', a'))$ means the maximum Q of the next state s' irrespective of the action a'. The proof of the Bellman equation is beyond the scope of this thesis, but Sutton's introductory book, mentioned previously, provides further explanation.

$$Q(s, a) = R(s) + \gamma(\max_{a'}(Q(s', a'))) \tag{5.6}$$

To improve convergence, i.e. when the estimated Q-function approaches the optimal Q^*, Equation 5.6 is modified to include the learning rate α which reduces the change in the Q-function with each iteration. The pseudocode for Q-learning presented by Sutton is reproduced in Table 5.2. Some environments, like chess, eventually terminate (at the end of the game) while others, such as maintaining a refrigerator's temperature, may never actually terminate.

Notice the algorithm recursively updates the Q-value, $Q(s, a)$, from the Q-value of the next state and greedy policy action, $\max_{a'} Q(s', a')$, regardless of the policy and action taken by the actor. This characteristic makes Q-learning an off-policy algorithm. The Q-value for a particular state and action only updates when the actor visits said state and action so the actor must still explore (i.e. not always take greedy

Table 5.2: Q-Learning Pseudocode

Initialize $Q(s, a)$ arbitrarily.

Repeat (for each episode):

    Initialize s

    Repeat (for each step of episode):

        Choose a from s using policy derived from Q (e.g., ϵ-greedy)

        Take action a, observe r, s'

        $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma \max_{a'}(Q(s', a'))]$

        $s \leftarrow s'$;

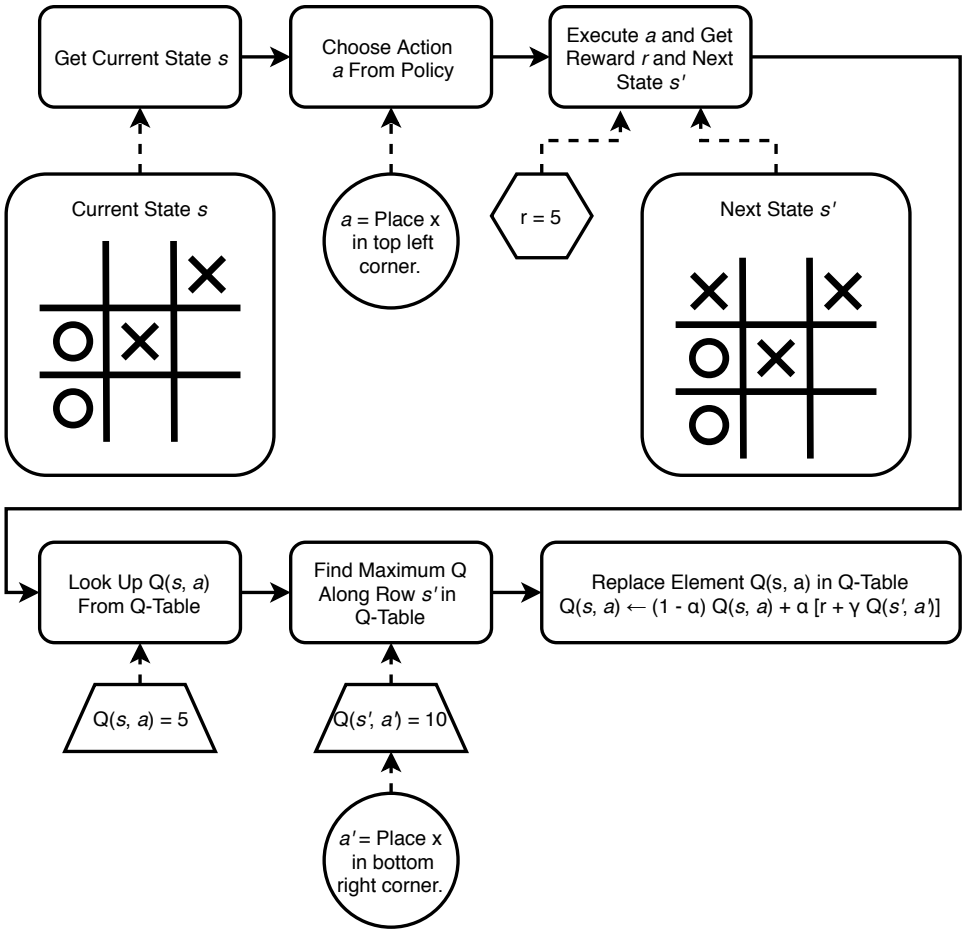
    until s is terminal

actions) for Q-learning to converge. Figure 5.4 illustrates the recursive Q-function update using the tic tac toe example.

Clearly, the tabular Q method quickly becomes unfeasible for more complex problems, especially when the states or actions are continuous rather than discrete. When the action or state spaces are continuous, the algorithm must discretize them, forcing a tradeoff between resolution and tractability. For the tic-tac-toe example, the table would be 19,683 rows by 9 columns for a total of 177,147 Q-values while the Q-value table for a game of chess would possess more elements than there are atoms in the known universe. Hence, methods such as the yet-to-be-discussed Deep Q-Network Method represent the Q-function using other structures like artificial neural networks.

5.2.2 State-Action-Reward-State-Action (SARSA)

SARSA is an on-policy algorithm which shares many characteristics with Q-learning [60]. The name comes from the fact that the Q update equation uses the current **s**tate and **a**ction as well as the next **r**eward, **s**tate, and **a**ction. The pseudocode for SARSA presented by Sutton is reproduced in Table 5.3.



Q-Table

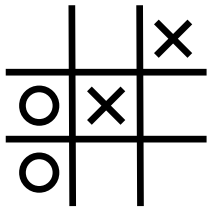
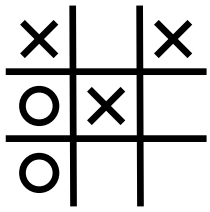
State \ Action	Place x in top left corner.	Place x in bottom right corner.
	5	-5
	0	10

Figure 5.4: Tabular Q Update Example

Table 5.3: SARSA Pseudocode

Initialize $Q(s, a)$ arbitrarily.

Repeat (for each episode):

    Initialize s

    Choose a from s using policy derived from Q (e.g., ϵ-greedy)

    Repeat (for each step of episode):

        Take action a, observe r, s'

        Choose a' from s' using policy derived from Q (e.g., ϵ-greedy)

        $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma Q(s', a')]$

        $s \leftarrow s'; a \leftarrow a';$

    until s is terminal

Unlike Q-learning, SARSA updates Q-values from the next state and next action $Q(s', a')$, making it an on-policy strategy. Like Q-learning, the actor must take exploratory actions for the algorithm to update Q at all states and actions and achieve convergence.

5.2.3 Deep Q Network (DQN)

Deep Q Networks, developed by Mnih et al. [37], aim to solve two significant problems with Q-learning and SARSA: the inability to handle situations with large state and action spaces and the inability to generalize learnings to new situations [59]. As mentioned previously, maintaining Q-values for the entire state and action space of a complex problem requires enormous memory. The second problem has to do with how the Q-value table updates. In both Q-learning and SARSA, the algorithms iteratively update Q-values for visited (state, action) pairs. However, to fully update the Q-value table means exploring every possible state and action combination multiple times, a non-trivial task. In other words, the algorithms cannot provide reasonable Q-value

estimates for unvisited situations.

To overcome these limitations, DQN replaces the tabular Q concept with an artificial neural network (ANN) where the inputs are the state and action and the output is the Q-value. The Q-network is denoted as $Q(s, a; \theta_i)$ where θ_i represents the network weights. The loss function (5.7) is the squared error between the Q-network output and the "target Q-value" as defined by the Bellman equation, $y_j = r + \gamma \max_{a'} Q(s', a'; \theta_i^-)$ [37]. For complete clarity, $Q(s, a; \theta_i^-)$ represents an identical but separate copy of Q-network $Q(s, a; \theta_i)$ except with weights θ_i^- from a previous iteration, explained later.

$$L_i(\theta_i) = (r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2 \tag{5.7}$$

To assist DQN training, Mnih et al. used a technique called experience replay [37] As the actor explores within environment, the algorithm records transitions consisting of the current state, action taken, reward, and next state at each time step into a so-called experience replay buffer. After collecting a minimum number of experiences, the artificial neural network trains from data sampled randomly from the buffer, hence the name "experience replay". The technique removes time correlations in the training data, improves convergence, and also allows experience reuse, advantageous when obtaining data is difficult or costly. In the context of ANNs, convergence refers to when the ANN output approaches the function being approximated by the network.

The second technique Mnih et al. used to counter training instability, i.e. when the network weights oscillate instead of settling, involves generating target Q-values, y_j, from an identical but separate ANN, called the target Q-network $\hat{Q}$ with weights θ_i^-. In effect, this creates a time delay between when the weights of Q are updated and when those updates affect the policy and target Q-network, reducing the likelihood of policy instability. For example, using a 64 transition mini-batch, the network randomly samples 64 transitions from the experience replay buffer and trains the

network Q with 64 iterations as in 5.7. After, the target network $\hat{Q}$ clones the weights θ_i from Q and the process repeats.

The DQN method replaces the tabular Q with an artificial neural network to solve RL problems with large Q spaces. However, the algorithm's policy still uses a discrete action space, posing a problem for situations requiring continuously variable actions. For example, a system with five continuous actions discretized into a mere four bits each already produces $(2^4)^5 = 1,048,576$ different action combinations. Policy gradient methods can overcome this limitation as described next.

5.2.4 Stochastic Policy Gradient Method

Q-Learning, SARSA, and DQN are value-based methods as they rely on finding the environment's underlying Q-function and produce a policy to maximize Q. Policy gradient methods find the optimal policy directly, making them policy-based strategies [59]. DeepMind's AlphaGo demonstrated PG's viability by famously defeating grandmaster Go player Fan Hui in 2015 and Lee Sedol in 2016 [52].

The core of the policy gradient method is the parametrized probabilistic action distribution $P[a|s;\theta]$. Essentially, the distribution defines the probability of any single action in the set of all possible actions for a particular given state s. For example, if an actor has three possible actions, they might occur 10%, 30%, and 60% of the time, respectively. However, since $P[a|s;\theta]$ is continuous, the policy would produce continuous-valued actions rather than three discrete ones. The parameters θ adjust the distribution's shape and therefore the likelihoods of each action.

The underlying premise of stochastic policy gradients is simple: get a state, take a particular action with probability $P[a|s;\theta]$, and record the reward. If the reward is "good", increase the probability of taking that action by modifying θ or decrease it if the reward is "bad". Note that the terms "good" and "bad" are intentionally vague

as to not constrain them to one particular interpretation. After many iterations, the actor will most likely take actions that produce good rewards. But of course, not all is so simple. Much like the so-called "butterfly effect" [31], a single action can cascade into a multitude of unknown futures meaning the worth of a particular action cannot be determined by the immediate reward produced.

Instead, the rewards are accumulated for a long period in episodes, at the end of which actions are judged based on the total reward. Now, a different issue arises: the credit assignment problem [21]. If an episode contains 1,000 actions, which ones ultimately produced the good reward and which were inconsequential or detrimental? Rather than determining the individual ones responsible, the algorithm deems all the actions in the episode culpable so if the episode's outcome is good, all actions taken become more likely and if not, the inverse occurs [27]. Like AlphaGo, many policy gradient implementations use an artificial neural network to represent the action probability distribution where the inputs are states and outputs are action probabilities [52]. Therefore, the act of making actions more or less probable is carried out with gradient descent similar to back propagation for ANNs [27]. However, the specific implementation details are beyond the scope of the thesis.

5.2.5 Deterministic Policy Gradient Method (DPG)

The deterministic policy gradient method (DPG), developed by Silver et al., shares the same foundation as stochastic policy gradients, but instead of representing the policy as a probability distribution, the policy deterministically chooses an action given a particular state [51]. Silver et al. have shown that the DPG is actually a limiting case of stochastic policy gradients where the policy distribution variance is 0. Another key difference is that while the stochastic policy gradient integrates over the state and action spaces, the DPG only integrates over the state space. Consequently,

the stochastic case requires more samples to compute. Specific details can be found in the original paper. Finally, while the stochastic policy inherently chooses exploratory actions due to its probabilistic nature, the deterministic case is more akin to a greedy policy. Therefore, Silver developed an off-policy learning algorithm using a stochastic policy to ensure sufficient exploration despite producing a deterministic policy.

5.2.6 Deep Deterministic Policy Gradient (DDPG)

The robot's motors are controlled with continuous actions so the policy gradient method provides a decent solution. However, PG methods converge more slowly and have less learning stability than the DQN algorithm, as compared in Table 5.4 [68]. Therefore, the deep deterministic policy gradient (DDPG), developed by Lillicrap et al., augments the DPG algorithm with techniques from DQN to obtain the best of both worlds.

Table 5.4: DQN and PG Comparison [68]

	Deep Q-Network Method	Policy Gradient Method
Policy Action Space	Discrete	Continuous
Policy State Space	Continuous	Continuous
Q-Function Action Space	Continuous	n/a
Q-Function State Space	Continuous	n/a
Learning Stability	More Stable	Less Stable
Convergence Speed	Faster	Slower

The DDPG algorithm is a model-free, off-policy actor-critic strategy based on the Silver et al.'s DPG algorithm combined with learnings from Mnih et al.'s work on the DQN algorithm [30][37][51]. DDPG improves on DPG by representing the actor's policy with an ANN and applying the experience replay and target network

techniques from DQN.

Additionally, Lillicrap et al. adapted batch normalization from Ioffe's work to allow ANN hyper-parameter generalization for environments with features of varying magnitude [26]; each feature in a minibatch is normalized to unit mean and variance. In other words, the DDPG algorithm can use the same set of hyper-parameters, e.g. learning rates, discount factor, update parameter τ, etc., for widely different environments. In their implementation, batch normalization was applied to network inputs, all layers of the policy network, and all layers of the Q network before the action input (detailed later).

To ensure adequate exploration, Lillicrap et al. added Ornstein-Uhlenbeck process noise to the policy output $\mu(s_t|\theta_t^\mu)$ to create the exploration policy $\mu'(s_t)$ as shown in Equation 5.8 [30]. The Ornstein-Uhlenbeck process satisfies the condition shown in Equation 5.9 where x_t is the process position at time t, θ and σ are parameters, and W_t is the Weiner process [65]. The process produces random, time-correlated values that drift toward a long-term mean, in this case 0. The Ornstein-Uhlenbeck action noise class shown in Listing 5.1 is provided by OpenAI under the MIT License [42].

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N} \tag{5.8}$$

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t, \theta > 0, \sigma > 0 \tag{5.9}$$

Listing 5.1: Ornstein-Uhlenbeck Action Noise [42]

```
class OrnsteinUhlenbeckActionNoise:
    def __init__(self, mu, sigma=0.3, theta=.15, dt=0.05, x0=None):
        self.theta = theta
        self.mu = mu
        self.sigma = sigma
        self.dt = dt
        self.x0 = x0
        self.reset()

    def __call__(self):
        x = self.x_prev + self.theta * (self.mu - self.x_prev) * self.dt + \
                self.sigma * np.sqrt(self.dt) * np.random.normal(size=self.mu.shape)
        self.x_prev = x
```

```
14          return x
15
16      def reset(self):
17          self.x_prev = self.x0 if self.x0 is not None else np.zeros_like(self.mu)
18
19      def __repr__(self):
20          return 'OrnsteinUhlenbeckActionNoise(mu={}, sigma={})'.format(self.mu, self.
                sigma)
```

Lillicrap et al. used DDPQ to solve over 25 different simulated physics environments such as cart pole swing-up and driving using the same network architecture and hyper-parameters, demonstrating the generalizability of the technique. For details of the environments, see OpenAI Gym [44]. Although the approach required about 2.5 million steps of experience to solve most problems, it needed 20 times less than DQN while still providing better performance in most cases.

To reiterate, the important advantage of the DDPG technique is its ability to handle both continuous action and state spaces of varying magnitude, critical to many real-life control problems. Specific algorithm details are covered in the Implementation section.

5.3 Implementation

All code is implemented in Python 3.6 and uses modules from TensorFlow [62] (ANN implementation), OpenAI Gym (Ornstein-Uhlenbeck action noise) [44], and Pyglet (environment rendering). An implementation of the Lillicrap's DDPG algorithm from Patrick Emami's wonderful reinforcement learning primer formed the initial code base to which modifications and new developments were added [18].

5.3.1 Coordinate Definitions

The Roborodentia field is 2438.4 mm long in the x direction and 1219.2 mm wide in y as shown in Figure 5.5 [12]. Viewed from above, the coordinate (0 mm, 0 mm) is the

bottom-left corner while (2438.4 mm, 1219.2 mm) is the top-right. Units of position (x and y), orientation, linear velocity, and rotational velocity are mm, mm/s, radians, and radians/s, respectively.

The robot position and orientation, (x, y, θ), is represented as a vector in the field plane where the vector's tail is positioned at the robot center and tip at the robot's front. Note that the vector orientation deviates from standard notation. At 0 radians, the vector points in the $+y$ direction, and at $+\pi/2$ radians, it points towards $-x$.

The x and y velocity of the robot refer to motion parallel to the field's x-axis and y-axis while the rotational velocity refers to the robot's rotation about its center. They are defined as $\dot{x}$, $\dot{y}$, and $\dot{\theta}$.

Three control inputs u_x, u_y, and u_θ correspond to the force applied to the robot and range between -2 to +2. For absolute clarity, positive u_x moves the robot in the direction of the positive x-axis, positive u_y pushes towards the positive y-axis, and positive u_θ torques the robot in the positive θ direction.

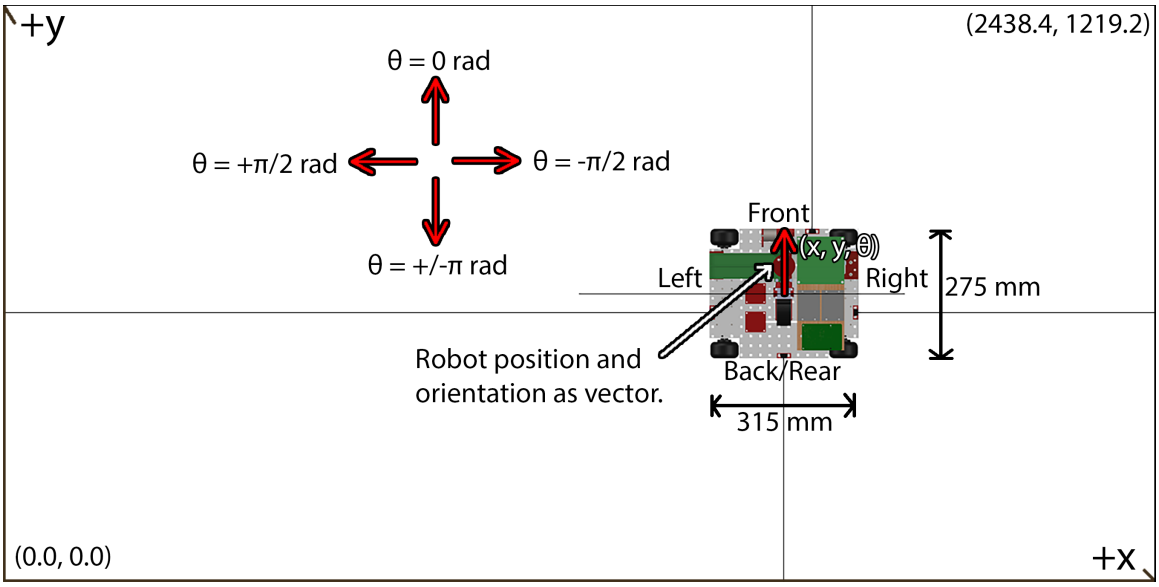


Figure 5.5: Roborodentia Field Definitions

5.3.2 Robot Simulation

Since training the network using the real-life robot is difficult and time-costly, episodes take place in a simulation of the robot and competition field. While an ideal simulation would model many kinematics and dynamics of the robot, robust system modeling is beyond the scope of the thesis. Instead, a rudimentary model serves to demonstrate the efficacy of the DDPG algorithm.

The simulation accounts for maximum wheel velocity and acceleration and calculates robot movement as a function of the four wheel velocities using the mecanum wheel equations [29][48]. It also handles collision with the environment walls. The simulation does not model unequal loading and friction between wheels, the momentum of the robot as a whole, or small differences between the motors and wheels.

5.3.3 Reward Assignment

The reward assignment must reflect the actor's goal: to move the robot to a desired position and orientation in the field. The resulting policy after training depends greatly on careful selection of which actions and/or states receive better or worse rewards. The rewards as implemented, shown in Equations 5.10 to 5.18, consist of three categories: distance from the set point, velocity, and action value. Note that the norm$(\cdot)$ function normalizes the angle to between $-\pi$ and $+\pi$. Each of the nine reward equations produces a maximum reward of 0 and includes squared terms to introduce quadratic reward scaling.

$$r_x = -0.00001(x - x_{desired})^2 \tag{5.10}$$

$$r_y = -0.00001(y - y_{desired})^2 \tag{5.11}$$

$$r_\theta = -1.0(norm(\theta) - norm(\theta_{desired}))^2 \tag{5.12}$$

$$r_{\dot{x}} = -0.0000005\dot{x}^2 \tag{5.13}$$

$$r_{\dot{y}} = -0.0000005\dot{y}^2 \tag{5.14}$$

$$r_{\dot{\theta}} = -0.1\dot{\theta}^2 \tag{5.15}$$

$$r_{u_x} = -0.001u_x^2 \tag{5.16}$$

$$r_{u_y} = -0.001u_y^2 \tag{5.17}$$

$$r_{u_\theta} = -0.001u_\theta^2 \tag{5.18}$$

The coefficients for r_θ, $r_{\dot{\theta}}$, and r_{u_θ} come from OpenAI Gym's pendulum environment [43]. The other coefficients scale the reward magnitudes within each category close to each other. Finally, the coefficients were adjusted through trial-and-error: the algorithm was run multiple times with slightly varied coefficients until the author felt the policy was "good enough". Future work includes a more systematic way to determine these coefficients.

5.3.4 Artificial Neural Networks

Critic Network

Listing 5.2 implements the critic network class. The function `create_critic_network()` at line 55 instantiates the TensorFlow network graph. The critic consists of a `s_dim` -node state input layer, another `a_dim`-node action input layer, two fully connected hidden layers, and a 1-node output layer, illustrated in Figure 5.6. The state input layer enters into the first hidden layer while the action input layer connects to the

second hidden layer. The first hidden layer uses a fully connected network layer of weights with biases, a batch normalization layer described previously, and the rectified linear unit (ReLU) activation function $ReLU(x) = max(0, x)$. Deep neural networks widely use the ReLU function for speeding up large computations both in forward and backward passes of the network [49]. The derivative of the ReLU function is simply 1 for positive inputs and 0 otherwise, greatly simplifying the back propagation algorithm. The second hidden layer consists of 600 nodes, half of which connect to the first hidden layer output and half that connect to the action input layer. Finally, the output layer is a single node with no activation function to obtain linear output.

The `action_gradients()` function returns a list of the gradients of the output with respect to each action, used by the actor network to update its weights. Finally, the `train()` function optimizes network weights and biases to minimize the loss function using the Adam optimizer [63]. The loss is defined as the mean square value of the difference between the mini-batch of critic outputs and the mini-batch of predicted Q-value from the target network equivalent to Equation 5.7. The Adam method, developed by Kingma and Ba, stands for "adaptive moment estimation" and possesses many of the benefits afforded by the AdaGrad and RMSProp algorithms including compatibility with sparse gradients, parameter update magnitudes invariant to gradient rescaling, step size annealing, and bounded step size [63][17][64].

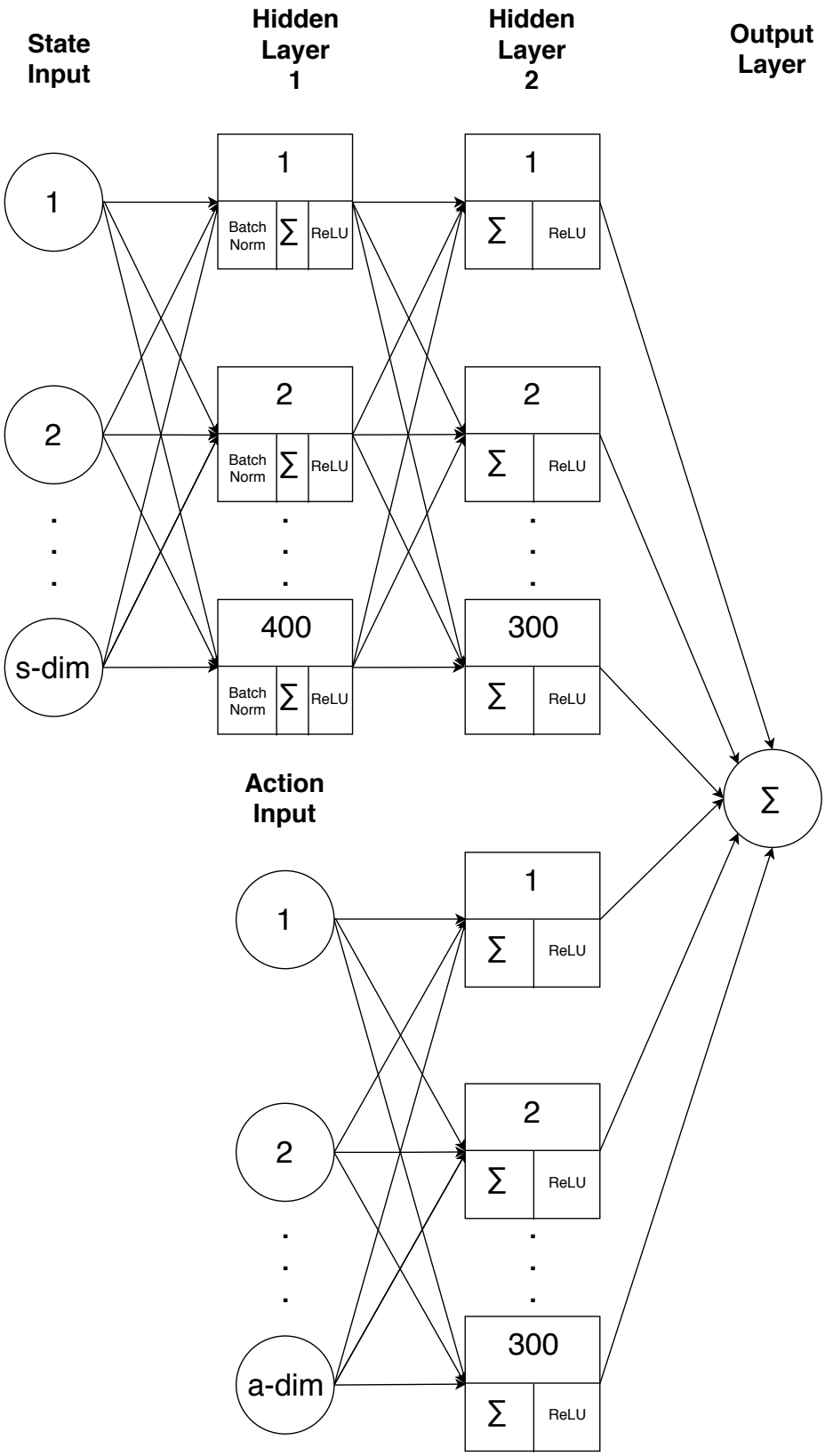


Figure 5.6: Critic ANN Structure

Listing 5.2: Critic Network Class

```
CRITIC_L1_NODES = 400
CRITIC_L2_NODES = 300

class CriticNetwork(object):
    """
    Input to the network is the state and action, output is Q(s,a).
    The action must be obtained from the output of the Actor network.

    """

    def __init__(self, sess, state_dim, action_dim, learning_rate, tau, gamma,
        num_actor_vars):
        self.sess = sess
        self.s_dim = state_dim
        self.a_dim = action_dim
        self.learning_rate = learning_rate
        self.tau = tau
        self.gamma = gamma

        # Create the critic network
        self.inputs, self.action, self.out = self.create_critic_network()

        self.network_params = tf.trainable_variables()[num_actor_vars:]

        # Target Network
        self.target_inputs, self.target_action, self.target_out = self.
            create_critic_network()

        self.target_network_params = tf.trainable_variables()[(len(self.
            network_params) + num_actor_vars):]

        # Op for periodically updating target network with online network
        # weights with regularization
        self.update_target_network_params = [self.target_network_params[i].assign( \
            tf.multiply(self.network_params[i], self.tau) + \
            tf.multiply(self.target_network_params[i], 1. - self.tau)) \
                for i in range(len(self.target_network_params))]

        # Network target (y_i)
        self.predicted_q_value = tf.placeholder(tf.float32, [None, 1])

        # Define loss and optimization Op
        self.loss = tflearn.mean_square(self.predicted_q_value, self.out)
        self.optimize = tf.train.AdamOptimizer(
            self.learning_rate).minimize(self.loss)

        # Get the gradient of the net w.r.t. the action.
        # For each action in the mini-batch (i.e., for each x in xs),
        # this will sum up the gradients of each critic output in the mini-batch
        # w.r.t. that action. Each output is independent of all
        # actions except for one.
        self.action_grads = tf.gradients(self.out, self.action)

        self.num_trainable_vars = len(
            self.network_params) + len(self.target_network_params)


    def create_critic_network(self):
        inputs = tflearn.input_data(shape=[None, self.s_dim], name='CriticInputs')
        action = tflearn.input_data(shape=[None, self.a_dim], name='CriticAction')
        net = tflearn.fully_connected(inputs, CRITIC_L1_NODES, name='CriticInputsNet'
            )
        net = tflearn.layers.normalization.batch_normalization(net)
        net = tflearn.activations.relu(net)

        # Add the action tensor in the 2nd hidden layer
```

```
63          # Use two temp layers to get the corresponding weights and biases
64          t1 = tflearn.fully_connected(net, CRITIC_L2_NODES, name='CriticNetT1')
65          t2 = tflearn.fully_connected(action, CRITIC_L2_NODES, name='CriticActionT2')
66
67          net = tflearn.activation(
68              tf.matmul(net, t1.W) + tf.matmul(action, t2.W) + t2.b, activation='relu')
69
70          # linear layer connected to 1 output representing Q(s,a)
71          # Weights are init to Uniform[-3e-3, 3e-3]
72          w_init = tflearn.initializations.uniform(minval=-0.003, maxval=0.003)
73          out = tflearn.fully_connected(net,1, weights_init=w_init,name='CriticNetOut')
74          return inputs, action, out
75
76      def train(self, inputs, action, predicted_q_value):
77          return self.sess.run([self.out, self.optimize], feed_dict={
78              self.inputs: inputs,
79              self.action: action,
80              self.predicted_q_value: predicted_q_value
81          })
82
83      def predict(self, inputs, action):
84          return self.sess.run(self.out, feed_dict={
85              self.inputs: inputs,
86              self.action: action
87          })
88
89      def predict_target(self, inputs, action):
90          return self.sess.run(self.target_out, feed_dict={
91              self.target_inputs: inputs,
92              self.target_action: action
93          })
94
95      def action_gradients(self, inputs, actions):
96          return self.sess.run(self.action_grads, feed_dict={
97              self.inputs: inputs,
98              self.action: actions
99          })
100
101     def update_target_network(self):
102         self.sess.run(self.update_target_network_params)
```

Actor Network

Listing 5.3 implements the actor ANN. The function `create_actor_network()` at line 56 defines the TensorFlow graph of the network. The actor uses an `s_dim`-node input layer, two fully connected hidden layers, and a `a_dim`-node output layer, illustrated in Figure 5.7. The two hidden layers consist of a fully connected network layer of weights with biases, a batch normalization layer described previously, and the ReLU activation function. The two hidden layers use 400 and 300 nodes, respectively, matching Lillicrap's implementation [30]. Finally, the output layer is a fully connected network, tanh activation function to limit the output to $[-1, +1]$, and a multiplier to

scale the output to $[-\texttt{action\_bound}, +\texttt{action\_bound}]$.

The `train()` function implements the key idea of the policy gradient method. The gradient of the output with respect to the network weights and biases are calculated and weighted the by the action gradients obtained from the critic. The weighting step effectively makes "good" actions more likely and "bad" actions less likely. The gradients are normalized by the batch size and then applied to the network weights and biased according to the Adam optimization method.

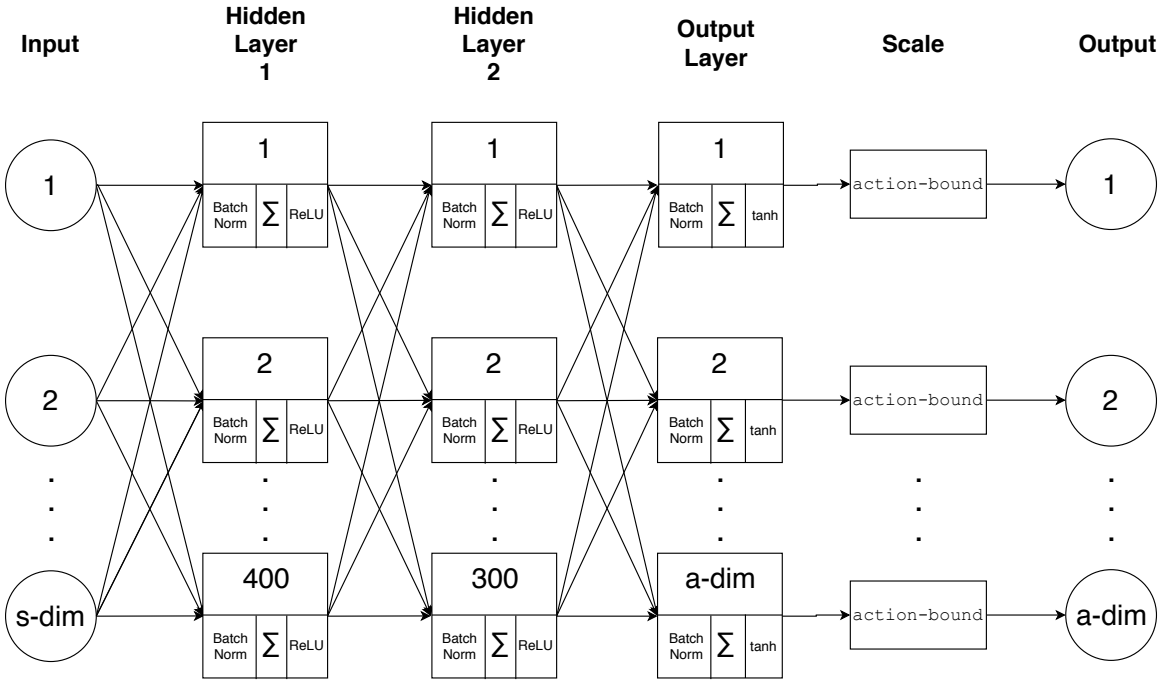


Figure 5.7: Actor ANN Structure

Listing 5.3: Actor Network Class

```
1  ACTOR_L1_NODES = 400
2  ACTOR_L2_NODES = 300
3
4  class ActorNetwork(object):
5      """
6      Input to the network is the state, output is the action
7      under a deterministic policy.
8
9      The output layer activation is a tanh to keep the action
10     between -action_bound and action_bound
11     """
12
13     def __init__(self, sess, state_dim, action_dim, action_bound, learning_rate, tau,
           batch_size):
```

```
14          self.sess = sess
15          self.s_dim = state_dim
16          self.a_dim = action_dim
17          self.action_bound = action_bound
18          self.learning_rate = learning_rate
19          self.tau = tau
20          self.batch_size = batch_size
21
22          # Actor Network
23          self.inputs, self.out, self.scaled_out = self.create_actor_network()
24
25          self.network_params = tf.trainable_variables()
26
27          # Target Network
28          self.target_inputs, self.target_out, self.target_scaled_out = self.
                create_actor_network()
29
30          self.target_network_params = tf.trainable_variables()[
31              len(self.network_params):]
32
33          # Op for periodically updating target network with online network
34          # weights
35          self.update_target_network_params = [self.target_network_params[i].assign( \
36                  tf.multiply(self.network_params[i], self.tau) + \
37                  tf.multiply(self.target_network_params[i], 1. - self.tau))
38                  for i in range(len(self.target_network_params))]
39
40          # This gradient will be provided by the critic network
41          self.action_gradient = tf.placeholder(tf.float32, [None, self.a_dim])
42
43          # Combine the gradients here
44          self.unnormalized_actor_gradients = tf.gradients(
45              self.scaled_out, self.network_params, -self.action_gradient)
46          self.actor_gradients = list(map(lambda x: tf.div(x, self.batch_size),
47              self.unnormalized_actor_gradients))
48
49          # Optimization Op
50          self.optimize = tf.train.AdamOptimizer(self.learning_rate).\
51              apply_gradients(zip(self.actor_gradients, self.network_params))
52
53          self.num_trainable_vars = len(
54              self.network_params) + len(self.target_network_params)
55
56      def create_actor_network(self):
57          inputs = tflearn.input_data(shape=[None, self.s_dim], name='ActorInputs')
58          net = tflearn.fully_connected(inputs, ACTOR_L1_NODES, name='ActorInputsNet')
59          net = tflearn.layers.normalization.batch_normalization(net, name='
                ActorBatchNorm1Net')
60          net = tflearn.activations.relu(net)
61          net = tflearn.fully_connected(net, ACTOR_L2_NODES, name='ActorNetNet')
62          net = tflearn.layers.normalization.batch_normalization(net, name='
                ActorBatchNorm2Net')
63          net = tflearn.activations.relu(net)
64          # Final layer weights are init to Uniform[-3e-3, 3e-3]
65          w_init = tflearn.initializations.uniform(minval=-0.003, maxval=0.003)
66          out = tflearn.fully_connected(
67              net, self.a_dim, activation='tanh', weights_init=w_init, name='
                    ActorOutNet')
68          # Scale output to -action_bound to action_bound
69          scaled_out = tf.multiply(out, self.action_bound)
70          return inputs, out, scaled_out
71
72      def train(self, inputs, a_gradient):
73          self.sess.run(self.optimize, feed_dict={
74              self.inputs: inputs,
75              self.action_gradient: a_gradient
76          })
77
```

```
78      def predict(self, inputs):
79          return self.sess.run(self.scaled_out, feed_dict={
80              self.inputs: inputs
81          })
82
83      def predict_target(self, inputs):
84          return self.sess.run(self.target_scaled_out, feed_dict={
85              self.target_inputs: inputs
86          })
87
88      def update_target_network(self):
89          self.sess.run(self.update_target_network_params)
90
91      def get_num_trainable_vars(self):
92          return self.num_trainable_vars
```

Both actor and critic classes create a regular and target network in the `__init__()` function. Each target network is structurally identical to its respective actor or critic network. Recall the network weights are denoted as θ_i while target network weights use θ_i^-. Each class provides a function `update_target_network()` that adjusts target network weights to be closer to the regular network weights by the factor $\tau = 0.001$ as in Equation 5.19. The `predict()` and `predict_targets()` functions return the forward pass output of the regular and target networks, respectively.

$$\theta_i^- \leftarrow \tau\theta_i + (1-\tau)\theta_i^- \tag{5.19}$$

5.3.5 DDPG

Figure 5.8 displays the DDPG training algorithm flowchart.

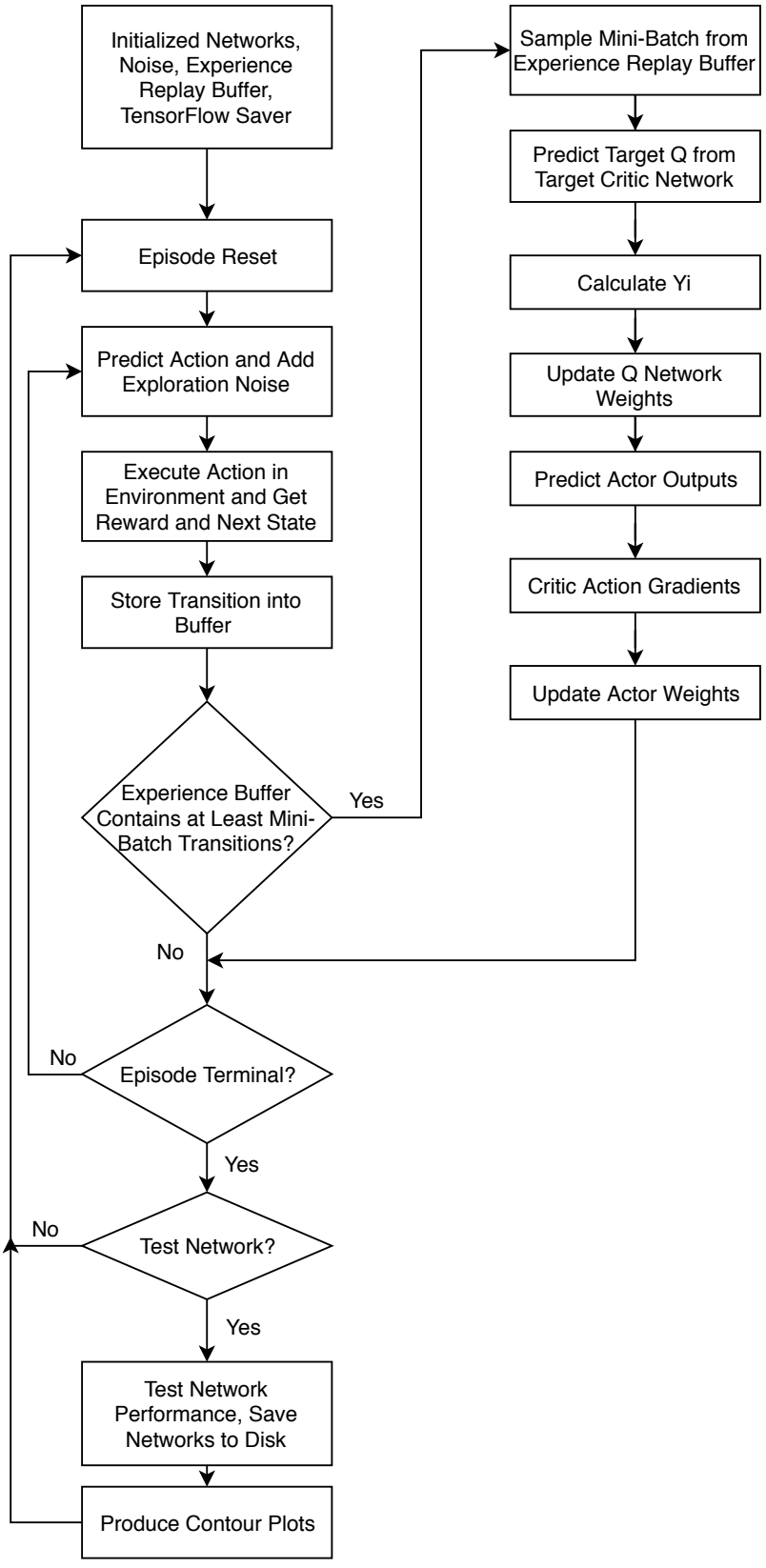


Figure 5.8: DDPG Flowchart

1. Set training parameters such as learning rates, discount factor γ, target network update parameter τ, and mini-batch size. The arguments use default values unless specified in the program arguments list.

Listing 5.4: Training Parameter Initialization

```
parser.add_argument('--actor-lr', help='actor network learning rate', default=0.001)
parser.add_argument('--critic-lr', help='critic network learning rate', default
    =0.0001)
parser.add_argument('--gamma', help='discount factor for critic updates', default
    =0.99)
parser.add_argument('--tau', help='soft target update parameter', default=0.001)
parser.add_argument('--buffer-size', help='max size of the replay buffer', default
    =1000000)
parser.add_argument('--minibatch-size', help='size of minibatch for minibatch-SGD',
    default=64)
```

2. Initialize the actor network, critic network, Ornstein-Uhlenbeck exploration noise, and experience replay buffer. Set the random seed to a specific value for repeatability.

Listing 5.5: Network, Noise, and Experience Replay Buffer Initialization

```
print("Instantiating actor...")
actor = ActorNetwork(sess, state_dim, action_dim, action_bound,
                     float(args['actor_lr']), float(args['tau']),
                     int(args['minibatch_size']))

print("Instantiating critic...")
critic = CriticNetwork(sess, state_dim, action_dim,
                       float(args['critic_lr']), float(args['tau']),
                       float(args['gamma']),
                       actor.get_num_trainable_vars())
actor_noise = OrnsteinUhlenbeckActionNoise(mu=np.zeros(action_dim), dt=env.dt)
replay_buffer = ReplayBuffer(int(args['buffer_size']), int(args['random_seed']))
```

3. Initialize TensorFlow variables and `tf.train.Saver()` for saving trained models to disk.

Listing 5.6: Saver Initialization

```
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
```

4. Start training loop. Each run of this loop is one episode.

   (a) Reset the environment randomly and get the initial state. Reset the

episode reward and average max Q for each step to track actor performance.

Listing 5.7: Episode Reset

```
s = env.reset()

# Track the episode reward and average max q
ep_reward = 0
ep_ave_max_q = 0
```

(b) Start step loop. Each run of this loop is one step in the episode.

i. Predict an action (i.e. forward pass through actor network) and add Ornstein-Uhlenbeck exploration noise. Execute the action and receive a reward, next state, and if the environment is terminating. Add the transition to the experience replay buffer.

Listing 5.8: Actor Predict and Step

```
# Predict action and add exploration noise
a = actor.predict(np.reshape(s, (1, actor.s_dim))) + actor_noise()
action = a[0]

# Execute action in environment to change state
s2, r, terminal, info = env.step(action)

replay_buffer.add(np.reshape(s, (actor.s_dim,)), \
        np.reshape(action, (actor.a_dim,)), r, terminal, \
        np.reshape(s2, (actor.s_dim,)))
```

ii. If the experience replay buffer contains at least the mini-batch number of transitions, sample a mini-batch of transitions. Use the target critic network to produce a target Q and calculate y_i from the mini-batch of transitions for use in the critic loss function. Update critic and actor network weights then target critic and actor weights.

Listing 5.9: Network Update

```
# Keep adding experience to the memory until
# there are at least minibatch size samples
if replay_buffer.size() > int(args['minibatch_size']):
   s_batch, a_batch, r_batch, t_batch, s2_batch = \
      replay_buffer.sample_batch(int(args['minibatch_size']))

   # Calculate targets
   target_q = critic.predict_target(
      s2_batch, actor.predict_target(s2_batch))
```

```
10
11      y_i = []
12      for k in range(int(args['minibatch_size'])):
13          if t_batch[k]:
14                y_i.append(r_batch[k])
15          else:
16                y_i.append(r_batch[k] + critic.gamma * target_q[k])
17
18      # Update the critic given the targets
19      predicted_q_value, _ = critic.train( s_batch, a_batch, \
20          np.reshape(y_i, (int(args['minibatch_size']), 1)))
21
22      ep_ave_max_q += np.amax(predicted_q_value)
23
24      # Update the actor policy using the sampled gradient
25      a_outs = actor.predict(s_batch)
26      grads = critic.action_gradients(s_batch, a_outs)
27      actor.train(s_batch, grads[0])
28
29      # Update target networks
30      actor.update_target_network()
31      critic.update_target_network()
```

(c) Update state for the next step and increment episode reward.

Listing 5.10: Step Cleanup

```
1  # Update state for next step
2  s = s2
3
4  # Increment episode reward
5  ep_reward += r
```

(d) If the environment is terminated, break out of the episode. Otherwise, loop back.

Listing 5.11: Episode Termination

```
1  # End of episode
2  if terminal:
3      break
```

(e) Print episode information to track progress.

Listing 5.12: Print Episode Results

```
1  print('Ep: %d | Reward: %d | Qmax: %0.4f' % \
2      (i, int(ep_reward), ep_ave_max_q / float(ep_len)))
```

(f) Periodically test the network performance (detailed in the subsection Actor Testing), record test results, save network weights to file, and produce contour plots of actor and critic outputs.

Listing 5.13: Network Evaluation

```
# Test the network's performance
if (i % test_period == 0):
    # Test the network and get the total reward
    print("Testing network in %d cases..." % (num_test_cases))
    test_reward, episodes = testNetworkPerformance(env, args, actor, num_test_cases)
    plotEpisodes(env.net_index, episodes, env.dt, i+1)

    # Save network session
    filepath = "./results/%s/%s_%d_%d/model.ckpt" % (sess_dir, args['env'], i+1,int(
        test_reward))
    save_path = saver.save(sess, filepath)

    # Contour plots of ANN output
    if (args['env'] != 'all'):
        plotANN(env.net_index, actor, i+1, 0)
        plotANN(env.net_index, critic, i+1, 1)
```

5.3.6 Two Approaches

Since the robot's mecanum wheels permit omni-directional movement, position and orientation control can be broken down into three separate DDPG actors, referred to as the "x translation", "y translation", and "rotation" or "angle" networks. Each actor receives two state inputs: position (x, y, or θ) and velocity ($\dot{x}$, $\dot{y}$, or $\dot{\theta}$) and outputs one of three orthogonal controls (u_x, u_y, and u_θ). This will be referred to as the "three actors" approach. Each of the three networks trains in isolation from the other two; for example, when training the x translation network, u_y and u_θ are fixed at 0.

To simultaneously apply the effects of each control, they are converted to four individual motor voltages (v_0, v_1, v_2, and v_3) by calculating the translation vector magnitude $V_d = \sqrt{u_x^2 + u_y^2}/2$ and angle $\theta_d = \text{atan2}(u_y/u_x)$ then using Equations 5.20 through 5.23 [29][48].

$$v_0 = V_d \sin(\theta_d + \pi/4) - u_\theta \tag{5.20}$$

$$v_1 = V_d \cos(\theta_d + \pi/4) + u_\theta \tag{5.21}$$

$$v_2 = V_d \sin(\theta_d + \pi/4) + u_\theta \tag{5.22}$$

$$v_3 = V_d \cos(\theta_d + \pi/4) - u_\theta \tag{5.23}$$

In the alternative "single actor" approach, a single actor receives x, y, θ, $\dot{x}$, $\dot{y}$, and $\dot{\theta}$ then produces the three control components u_x, u_y, and u_θ.

While using three actors improves training convergence speed, it does not account for possible effects of simultaneously-applied control components as with the single actor case. However, it does allow for partial system retraining. For example, if the y translation network does not meet the design specification, it can be retrained independently whereas the single actor would require complete retraining. Additionally, separate actors simplifies troubleshooting and reward assignment tuning by isolating effects and reducing problem complexity. The primary downside to three actors is reduced speed due to processing three networks at each step versus just one.

A hybrid approach would start with the three actors approach, to tune and refine the reward assignments and hyper-parameters, and finish with the single actor to reap the processing speed advantage as shown in Figure 5.9.

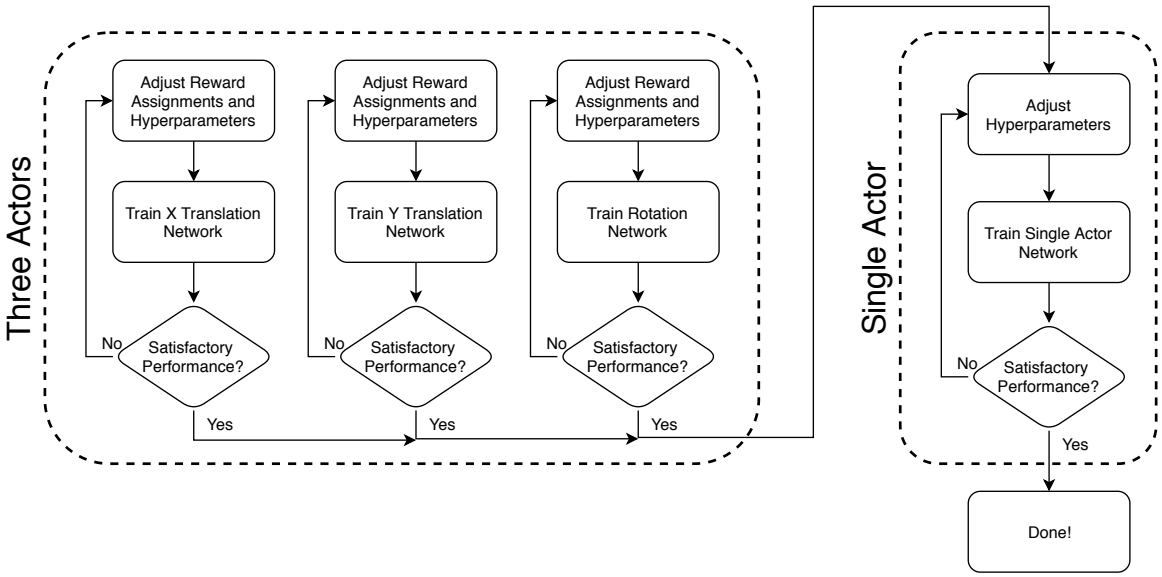


Figure 5.9: Hybrid Approach Flow

5.3.7 Training

Each episode lasts 10 seconds with 0.05 second steps for a total of 200 steps per episode. The learning rates, discount factor, target network update parameter τ, and mini-batch size for the three actors approach come from the original DDPG publication. The learning rates and τ for the single actor variant are 10 times smaller to improve stability. Table 5.5 summarizes the training parameters.

Table 5.5: Training Parameters

Parameter	Three Actors	Single Actor
Episode Length (s)	10	10
Episode Step (s)	0.05	0.05
Actor Learning Rate α_{actor}	10^{-4}	10^{-5}
Critic Learning Rate α_{critic}	10^{-3}	10^{-4}
Discount Factor γ	0.99	0.99
Target Network Update Parameter τ	10^{-3}	10^{-4}
Mini-Batch Size	64	128

Training took place on a desktop computer with an Intel i5-4670K CPU and Nvidia GeForce GTX 980 Ti GPU running Microsoft Windows 10 Pro [25][41][67]. The computer processed 21,200 episodes of single actor training in about 23 hours and 51 minutes, achieving an average rate of 1 episode every 4.08 seconds.

5.3.8 Testing

The training loop periodically tests the actor to evaluate performance with the `testNetworkPerformance()` function shown in Listing 5.14. The actor steps through a series of predefined test cases. For example, the robot orientation θ ranges from $-\pi$

to $+\pi$. If using 11 test cases, the robot would begin at $\theta = [-\pi, -0.8\pi, -0.6\pi, -0.4\pi, -0.2\pi, 0, +0.2\pi, +0.4\pi, +0.6\pi, +0.8\pi, \pi]$ to evaluate behavior at various states. Additionally, the action does not receive Ornstein-Uhlenbeck exploration noise during testing so the actor only takes greedy actions. The function returns the average reward per episode as the measure of actor performance. The testing procedure as implemented uses 41 test cases.

Listing 5.14: Actor Testing Function

```
# Tests the actor network against a number of test cases
def testNetworkPerformance(env, args, actor, num_test_cases = 10, render = False):
    test_total_reward = 0.0
    episodes = []

    # Test the network against random scenarios
    env.setWallCollision(True)
    for m in range(num_test_cases + 1):
        transitions = []
        s = env.reset(False, True, m, num_test_cases)
        ep_reward = 0.0
        for n in range(int(args['max_episode_len'])):
            if (args['render_env'] and render == True):
                env.render()

            # Choose action based on inputs
            a = actor.predict(np.reshape(s, (1, actor.s_dim)))
            action = a[0]

            # Execute action and get new state, reward
            s, r, terminal, info = env.step(action)
            ep_reward += r

            transition = (action, s, r)
            transitions.append(transition)
            if terminal:
                break
        episodes.append(transitions)
        test_total_reward += ep_reward

    env.setWallCollision(False)

    # Return the average test reward
    return (test_total_reward / (m+1), episodes)
```

5.4 Results

X Translation Network

The x translation network takes in the robot's x error, the difference between the desired and actual x positions, and velocity $\dot{x}$ and outputs control u_x. The reward is

calculated as shown in Equation 5.24. The actor receives a higher reward for keeping the robot near the desired x set point, minimizing the x velocity, and minimizing the effort (i.e. low u_x values).

$$r = -0.00001(x - x_{desired})^2 - 0.0000005\dot{x}^2 - 0.001u_x^2 \tag{5.24}$$

As described above in the Testing section, the networks are periodically tested to observe their change in performance with more training. Figures 5.10 and 5.11 show the average testing episode reward produced by the `testNetworkPerformance()` function versus the number of episodes trained. Figure 5.12 shows the average max Q versus number of episodes trained. The average max Q is calculated as the average maximum Q within each mini-batch of predicted Q's. Since the algorithm is model-free and the actor starts with zero knowledge of the environment, the initial test rewards remain near -2000. However, the actor quickly learns and achieves a -58 test reward after 161 episodes. After 181 episodes, the test reward begins to decrease, indicating the possibility of over-training. Interestingly, the critic produces invalid Q estimates; the maximum reward for any step, and therefore Q, is 0. However, this does not impact the actor's ability to determine desirable actions.

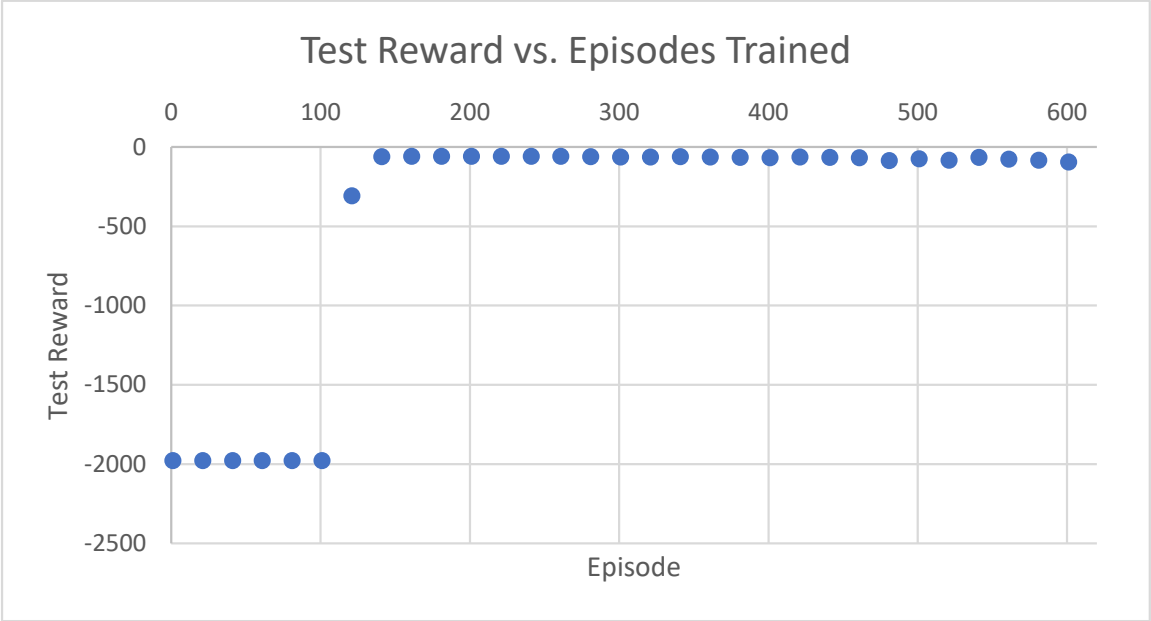


Figure 5.10: X Translation Test Reward

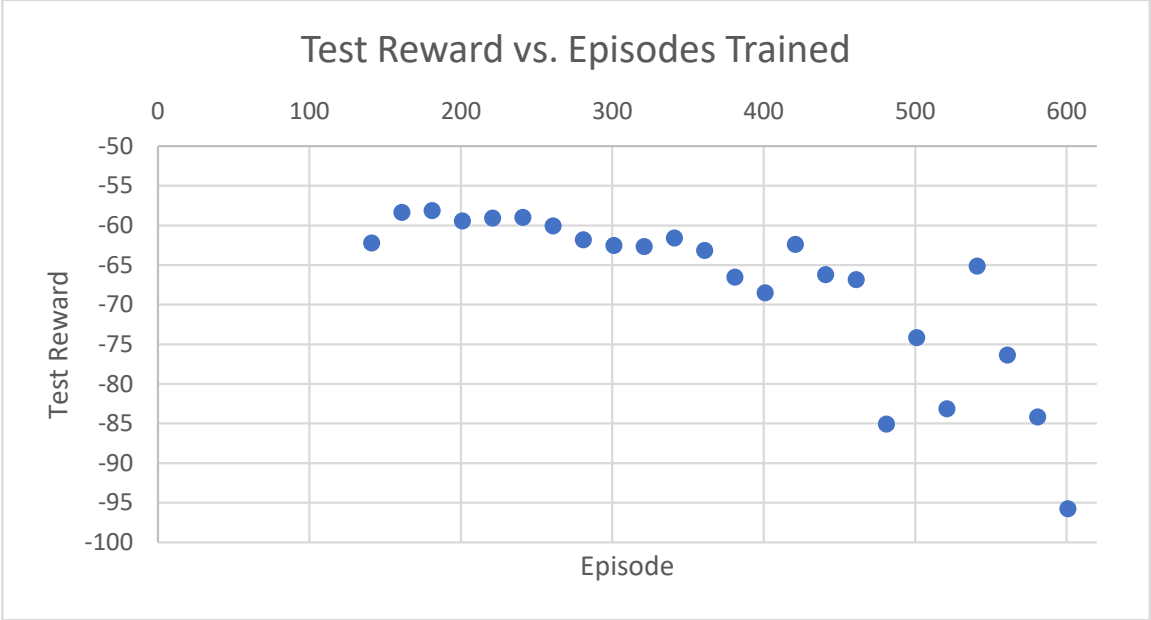


Figure 5.11: X Translation Test Reward Zoomed

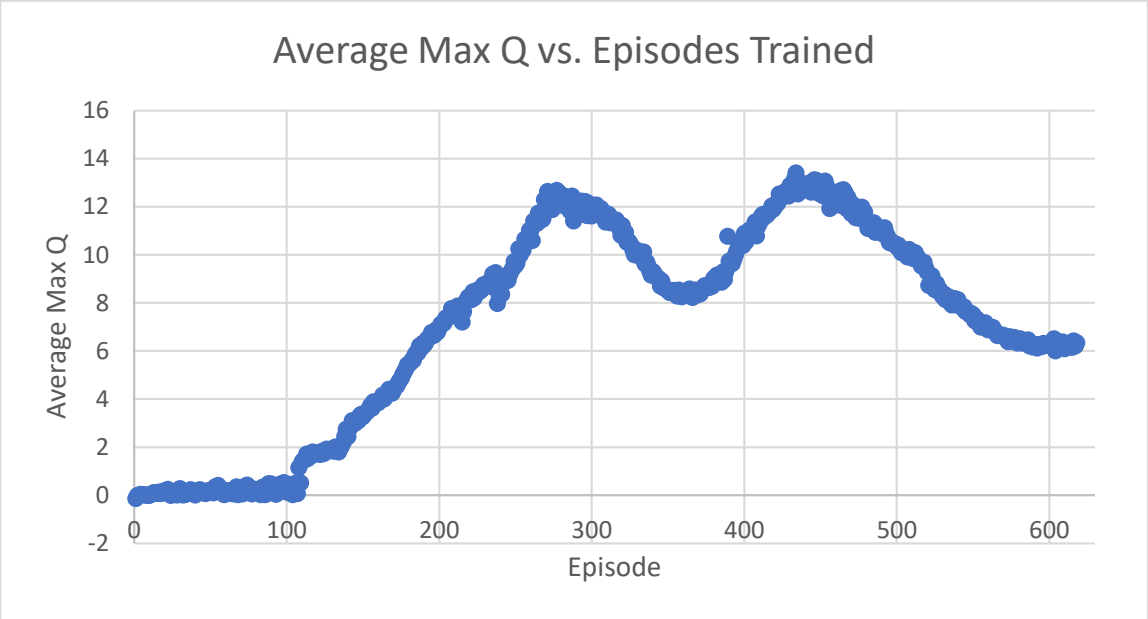


Figure 5.12: X Translation Network Average Max Q

Figure 5.13 shows the action u_x, error from the set point, and reward versus time for eight episodes with different initial conditions after 161 episodes trained. Appendix F provides additional plots for different numbers of episodes trained. Regardless of the starting distance from the set point, the actor appears to drive the robot at maximum speed to reach the set point as quickly as possible since the positional error affects the reward most drastically. Soon after reaching the desired x position, the action decays to zero.

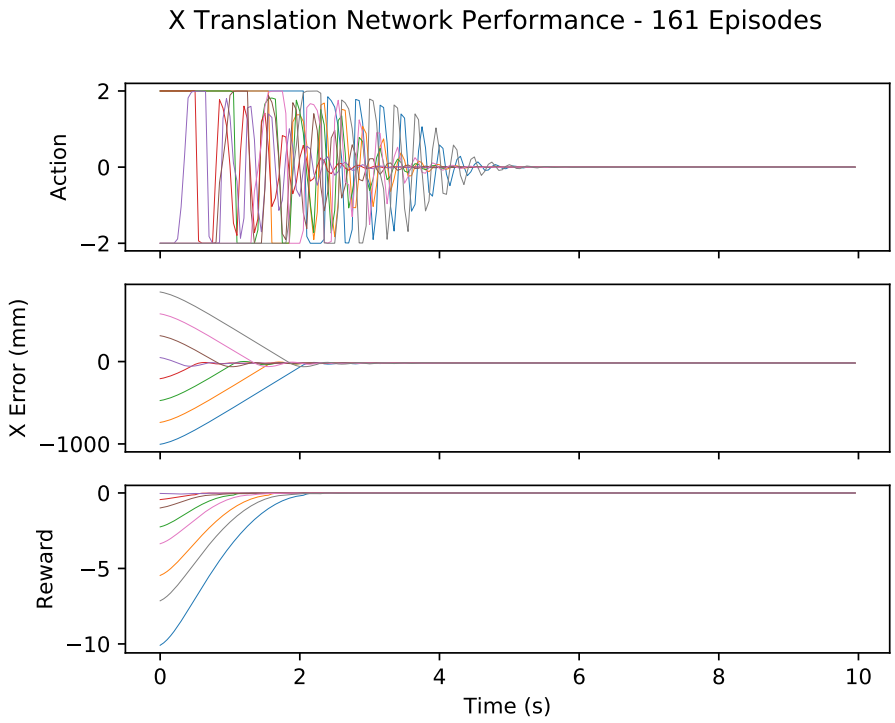


Figure 5.13: X Translation Network Performance – 161 Episodes

Figure 5.14 displays a series of contour plots of the actor outputs as functions of the two inputs over different numbers of episodes trained where the color indicates the action u_x. The plots reveal the trained actor's output as highly polarized with only a sliver of low-valued u_x. Figure 5.15 shows a similar series of plots for the critic output. Since Q is a function of both the state and action, each point represents the highest Q-value among all actions at the particular state. The plot of Q follows intuition: the nearer the robot to the desired x location, the greater the expected long-term reward.

Y Translation Network

The y translation network is nearly identical to the x translation but with a variable change so refer to the above for details. Figures 5.19 through 5.18 contain plots

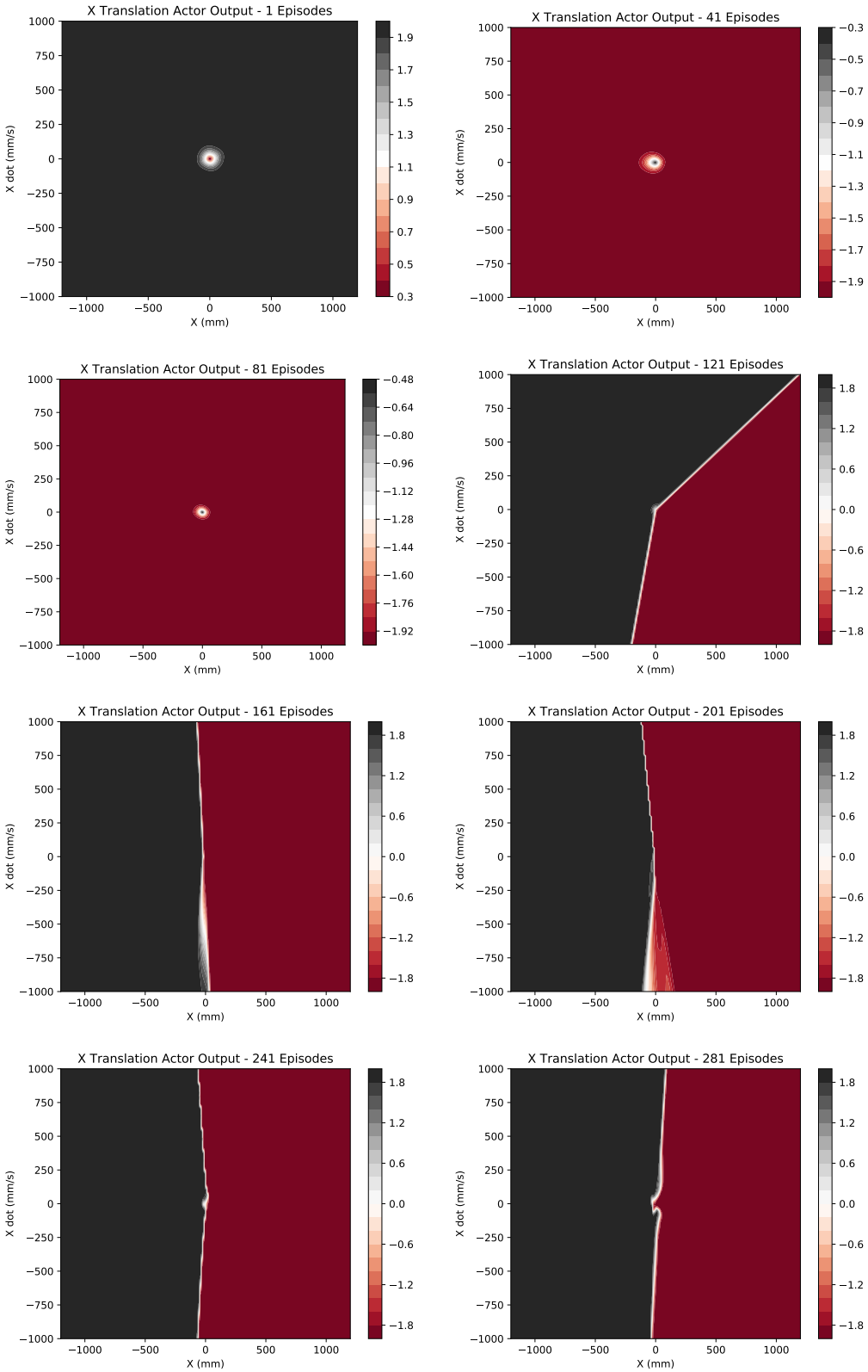


Figure 5.14: X Translation Actor Output Progression

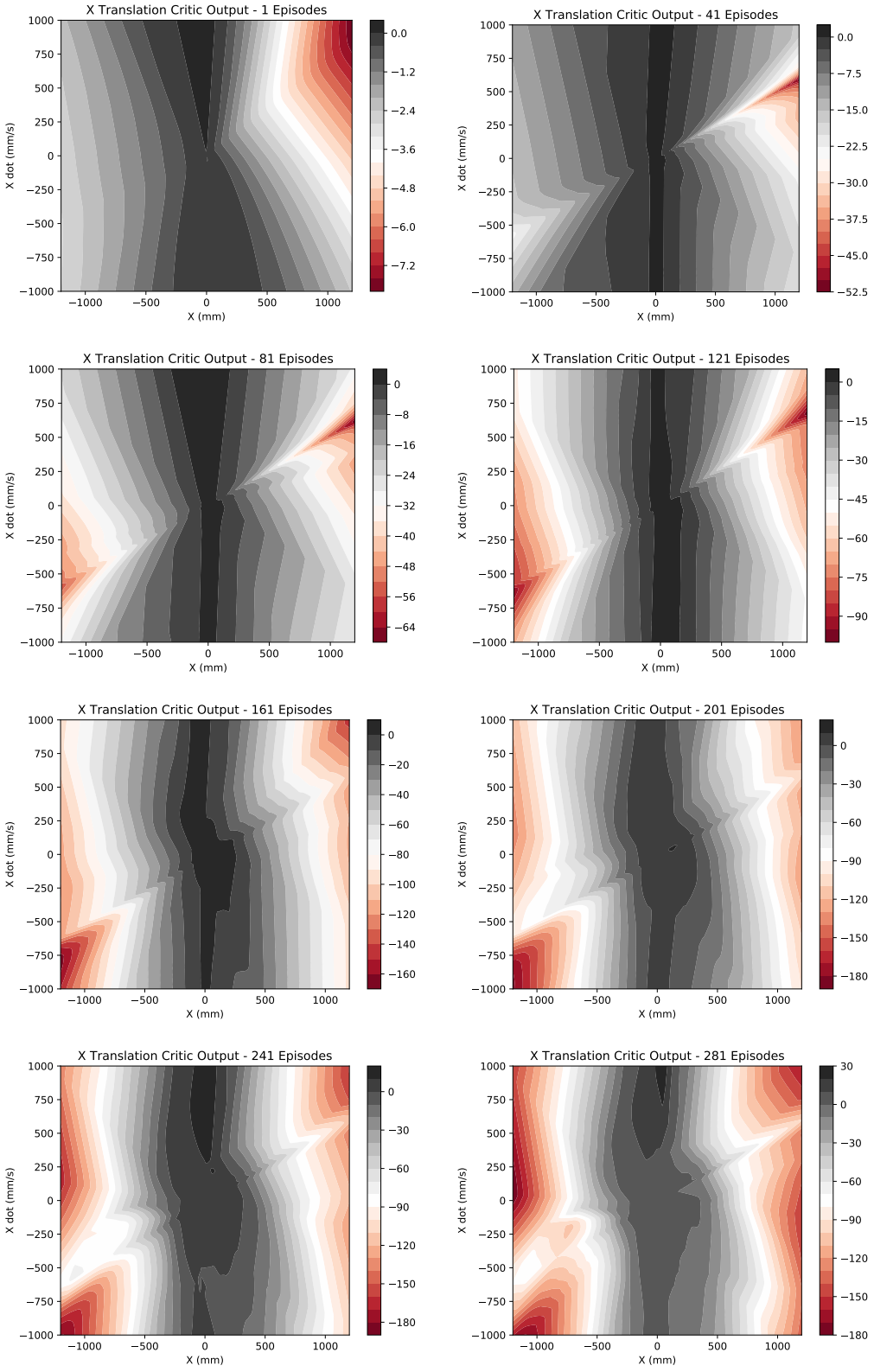


Figure 5.15: X Translation Critic Output Progression

equivalent to those shown for the x translation network. The test reward shows the same pattern of low test reward followed by a steep climb to a plateau. The jump in test reward aligns with the jump in average max Q at episode 161. The network also suffers from a decline in test reward from over training.

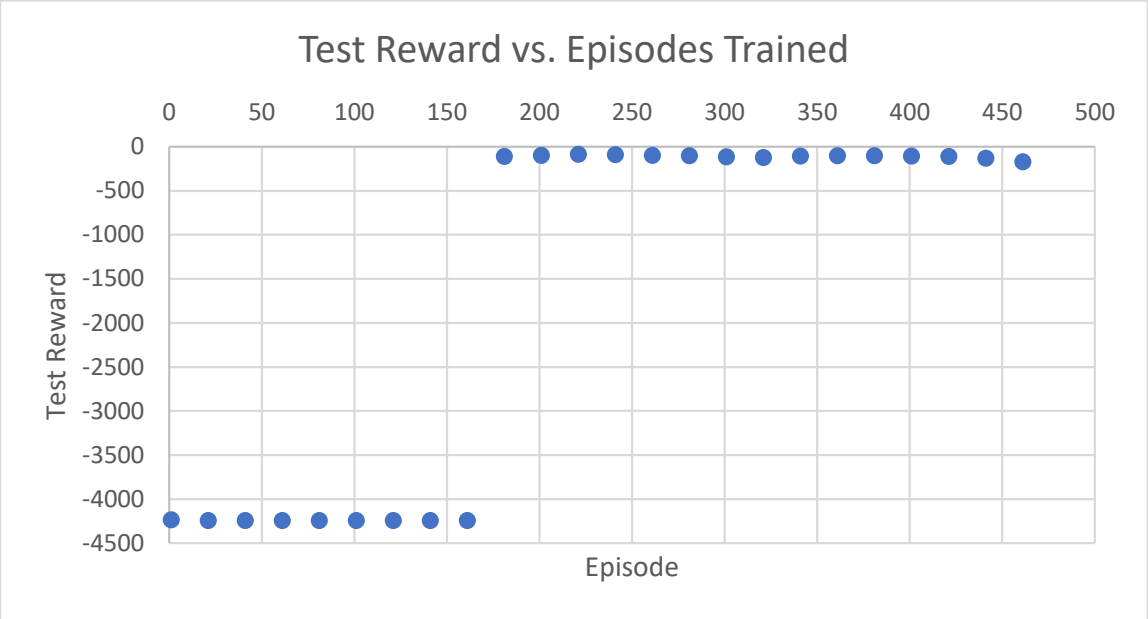


Figure 5.16: Y Translation Test Reward

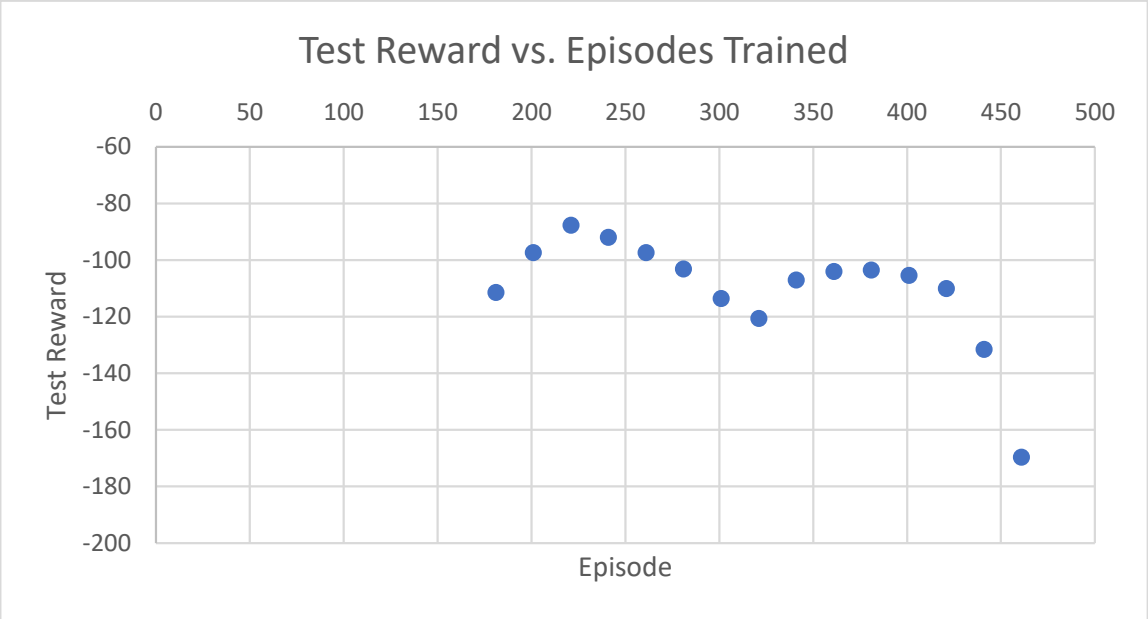


Figure 5.17: Y Translation Test Reward Zoomed

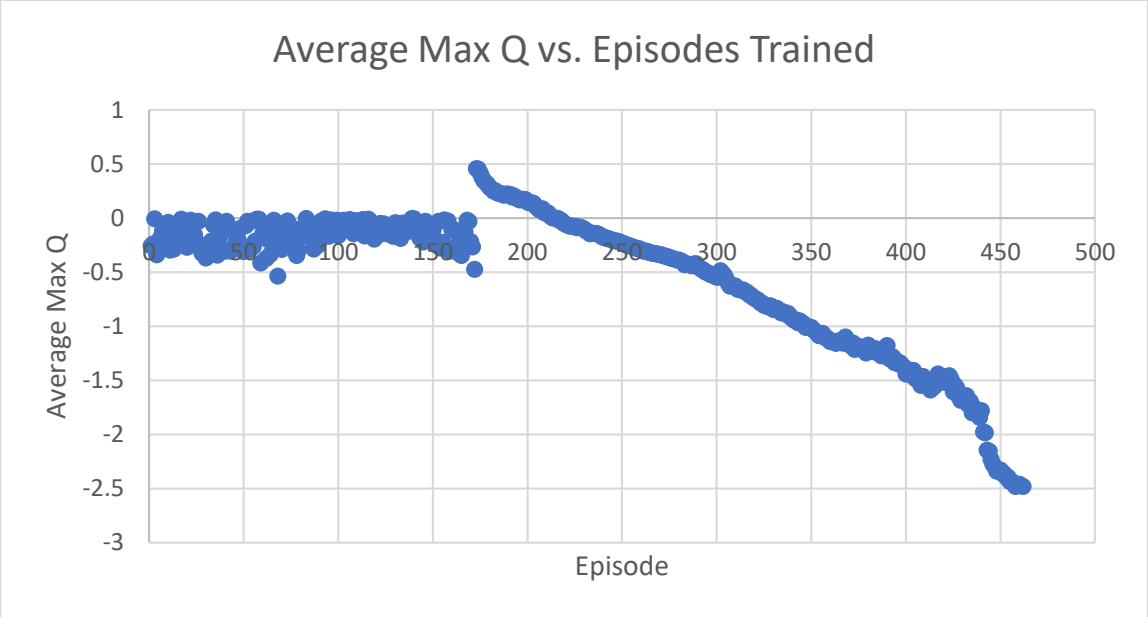


Figure 5.18: Y Translation Network Average Max Q

Figure 5.19 displays the actor's transient response from eight different initial starting points after 221 episodes trained. Appendix G contains additional plots for other

episodes trained. The robot reaches the set point with a small overshoot, and the action decays to zero as desired, indicating a successful policy.

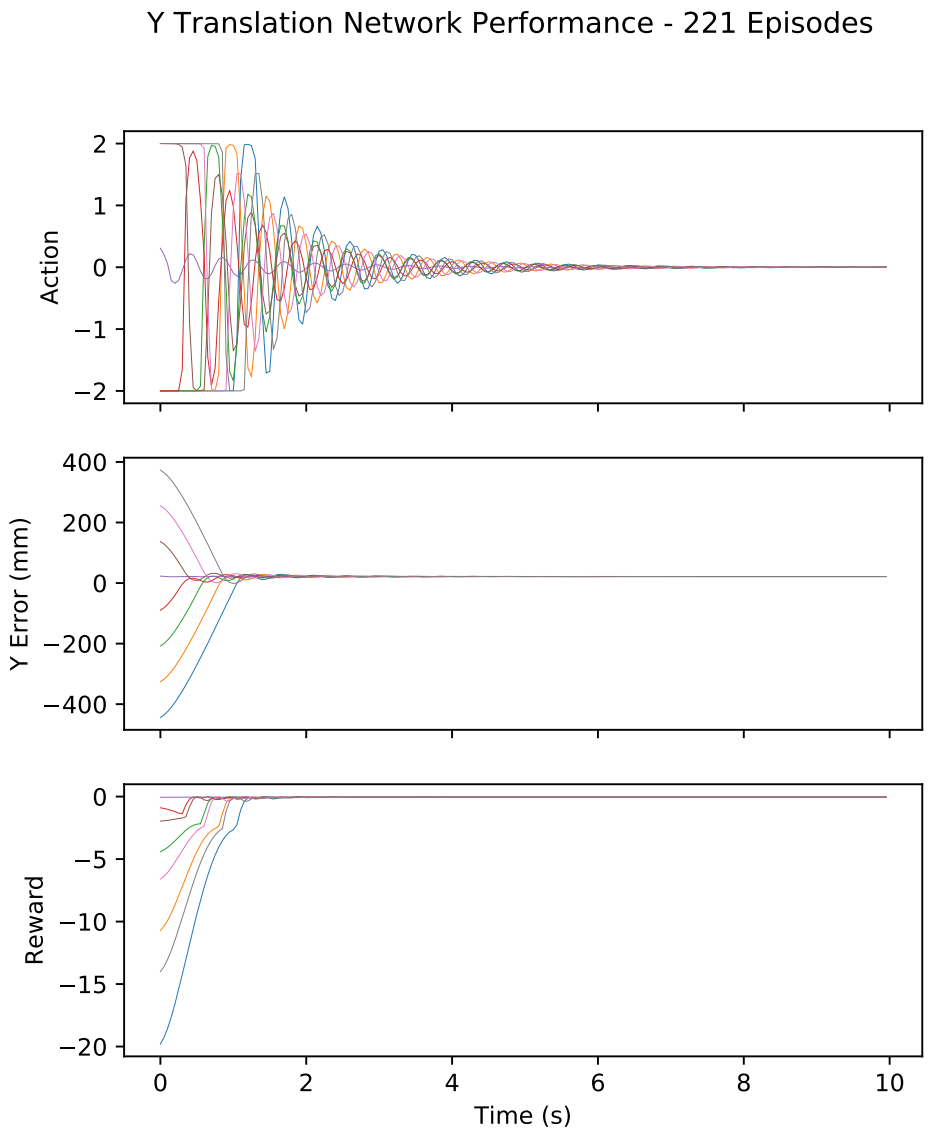


Figure 5.19: Y Translation Network Performance – 221 Episodes

Figures 5.20 and 5.21 show the actor and critic contour plots, respectively. As expected, the contours are highly reminiscent of those for the x translation network.

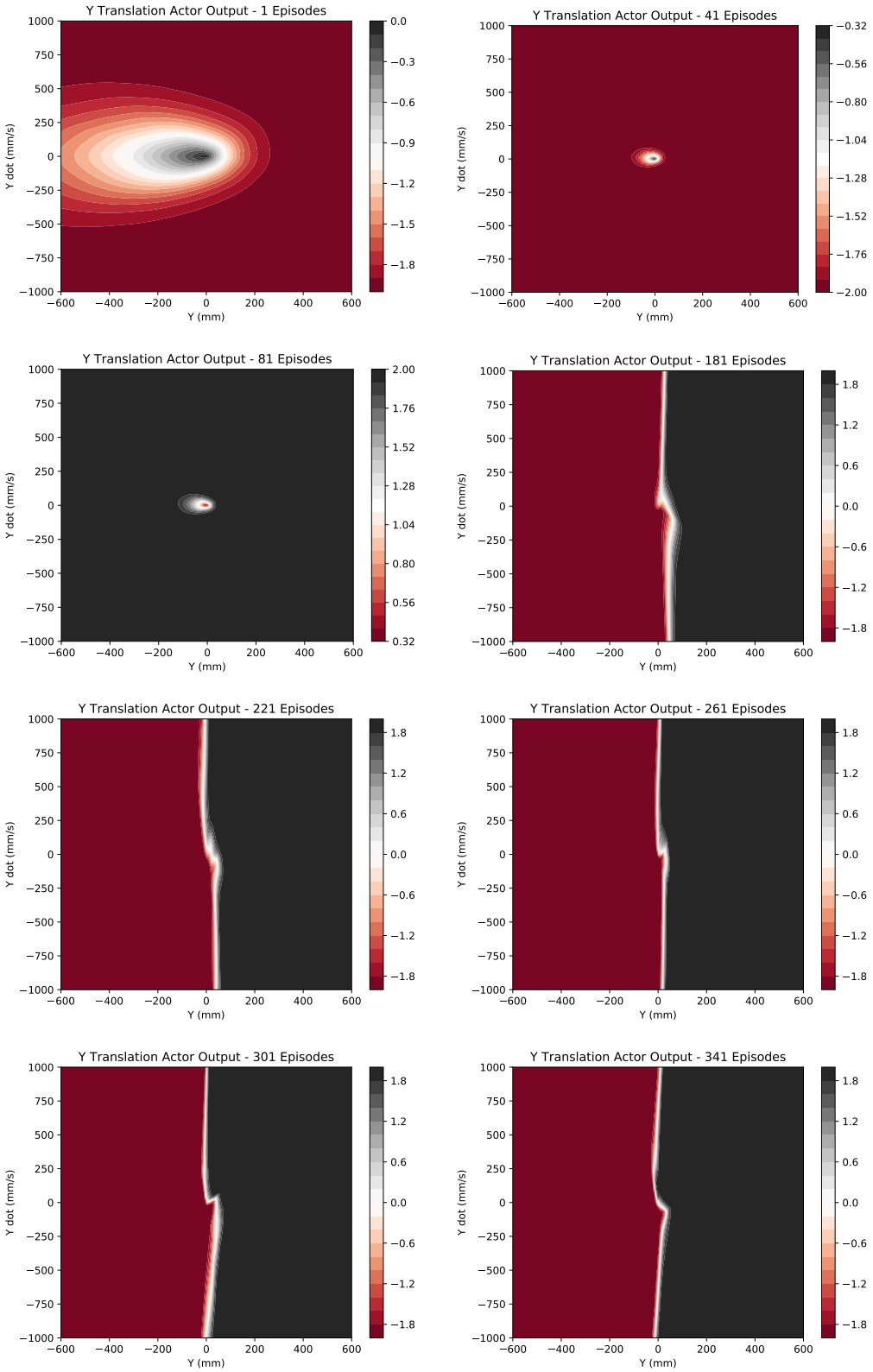


Figure 5.20: Y Translation Actor Output Progression

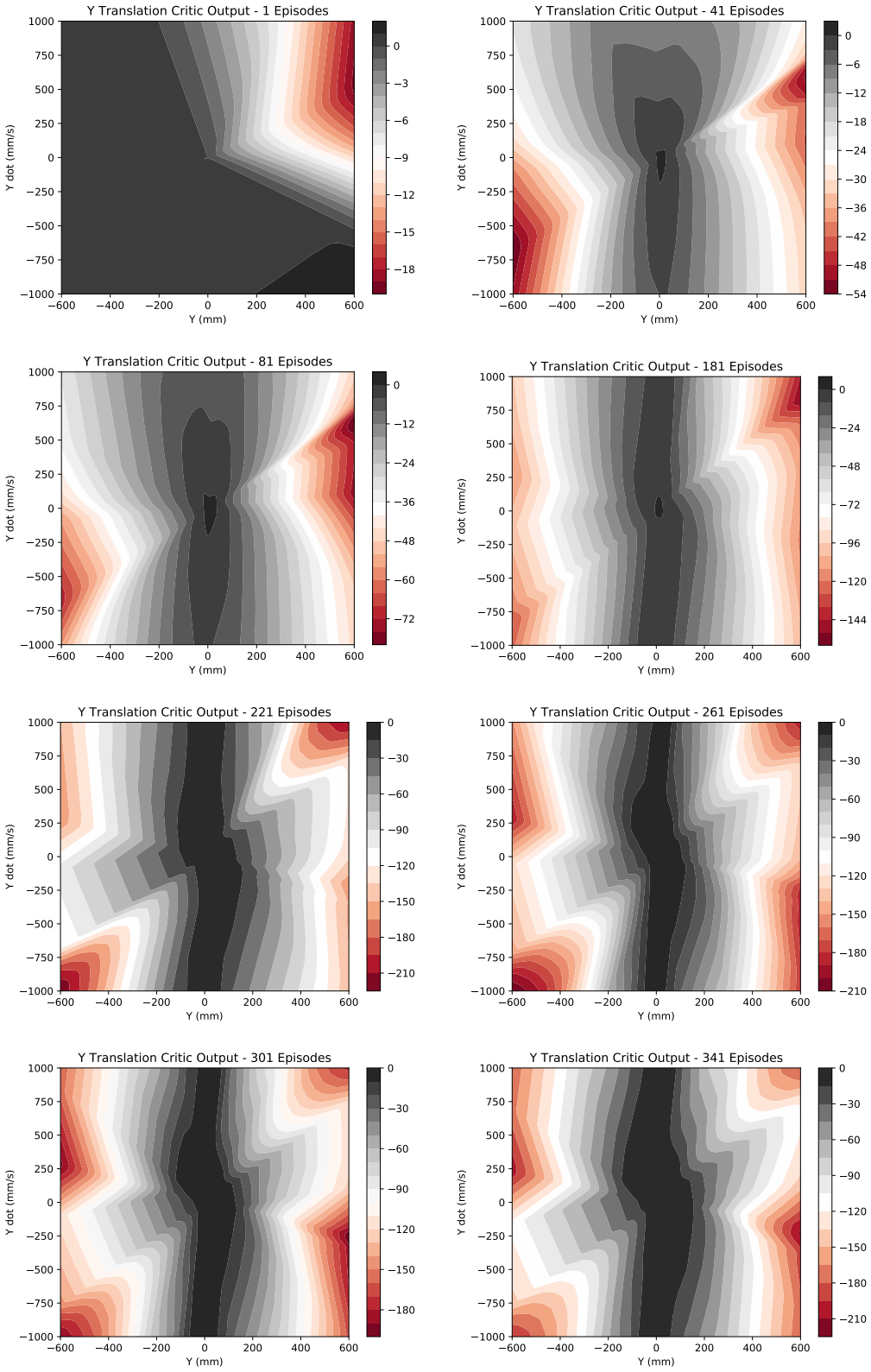


Figure 5.21: Y Translation Critic Output Progression

Rotation Network

The angle network receives the robot's angle error as $\cos(\theta - \theta_{desired})$ and $\sin(\theta - \theta_{desired})$ as well as the rotational velocity $\dot{\theta}$ and outputs u_θ. The reward is calculated as shown in Equation 5.25 where the norm() function normalizes the angle to between $-\pi$ and $+\pi$.

$$r = -1.0(\text{norm}(\theta) - \text{norm}(\theta_{desired}))^2 - 0.1\dot{\theta}^2 - 0.001u_\theta^2 \tag{5.25}$$

Figures 5.22 and 5.23 show test reward progress. The rotation network learns faster than either the x or y actors, achieving a reasonably good policy after only 41 episodes of training, due to the cyclical nature of rotation. Even if the actor decides to constantly spin the robot in one direction, it passes the set point with each revolution, obtaining useful experiences near the desired angle. However, in the x or y translation case, a constant movement in one direction leads the robot away from the set point and eventually gets it stuck at a wall instead of gaining effective experiences.

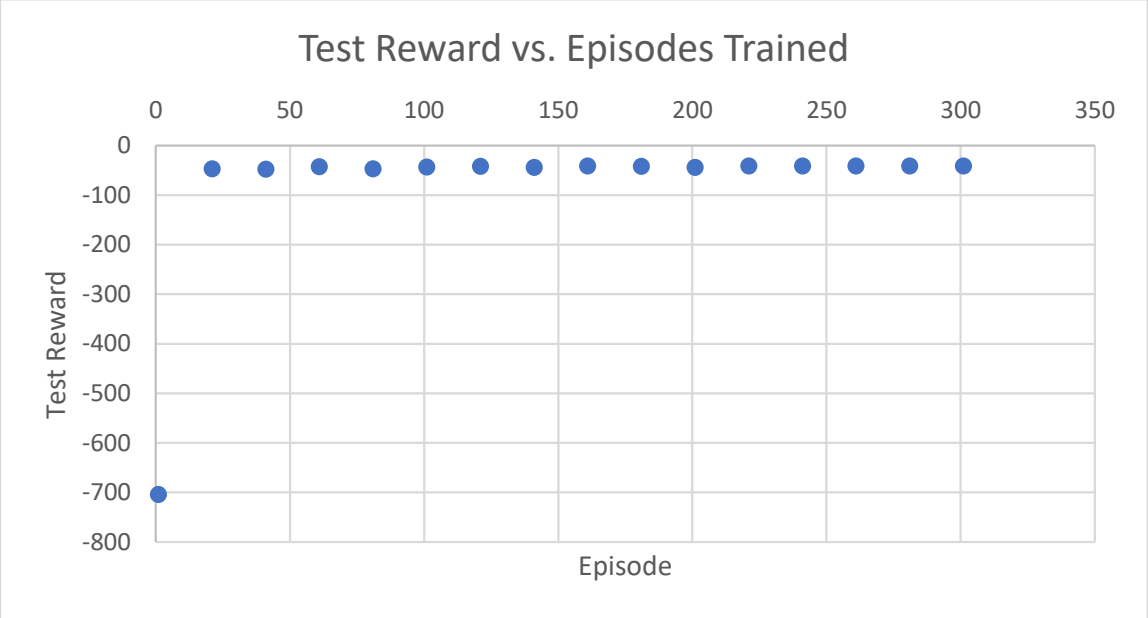


Figure 5.22: Angle Test Reward

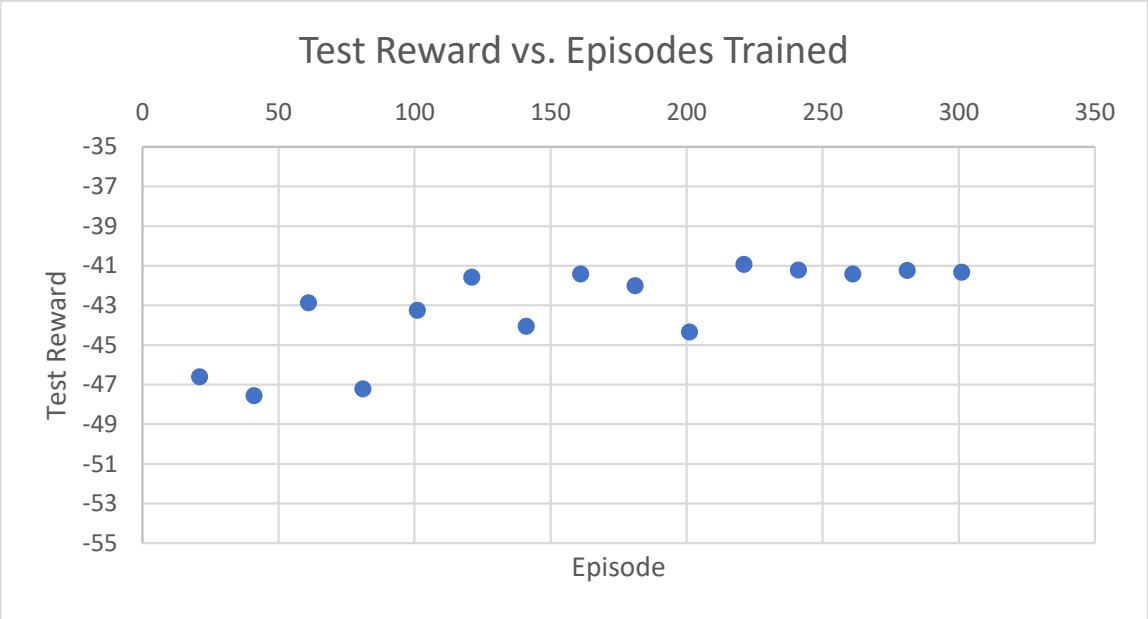


Figure 5.23: Angle Test Reward Zoomed

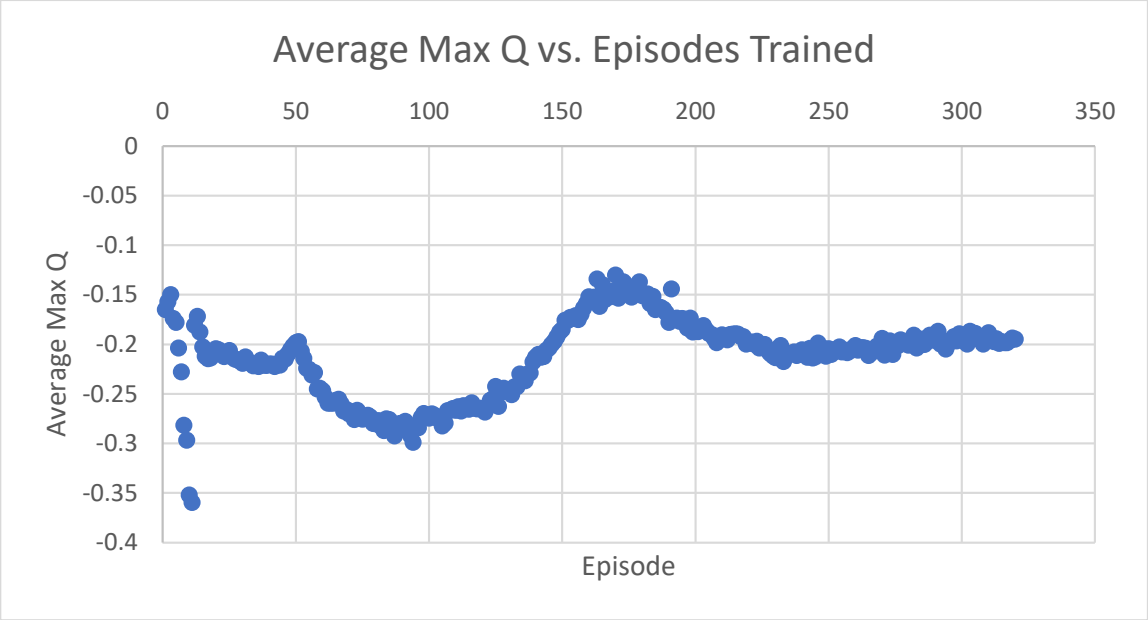


Figure 5.24: Angle Network Average Max Q

As seen in Figure 5.25, the rotation actor monotonically brings the robot to the desired angle with no overshoot and reduces the action to 0 afterwards. Appendix H

provides additional plots for different numbers of episodes trained.

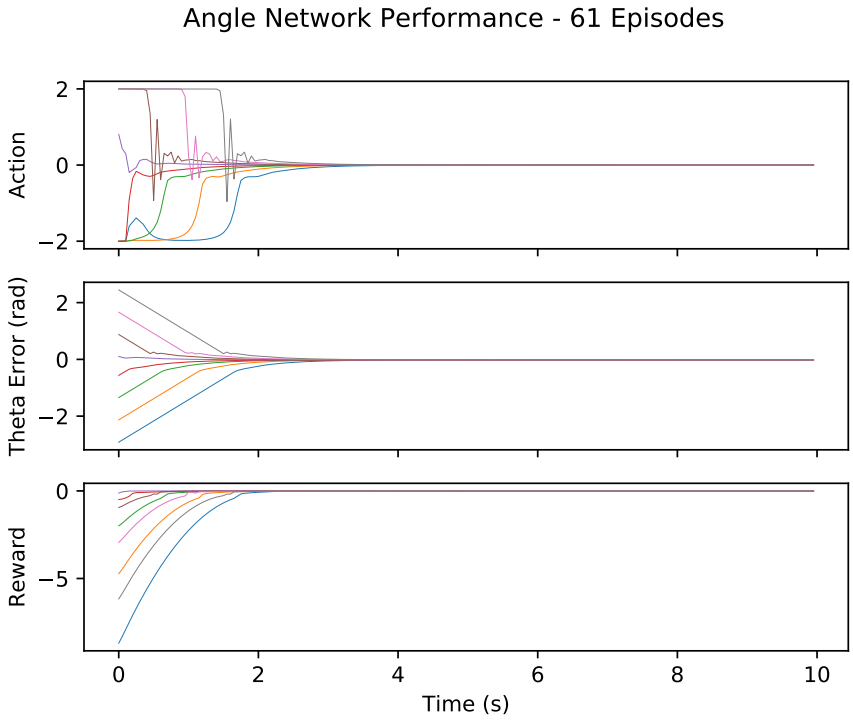


Figure 5.25: Angle Network Performance – 61 Episodes

Figures 5.26 and 5.27 contain contour plots of the actor and critic outputs. As expected, the plots are periodic with respect to θ. The actor contours take on a vaguely sinusoidal shape with only a thin sliver of area producing low action values, much like the other two networks. The critic network produces high Q values for states near the desired angle and low angular velocity.

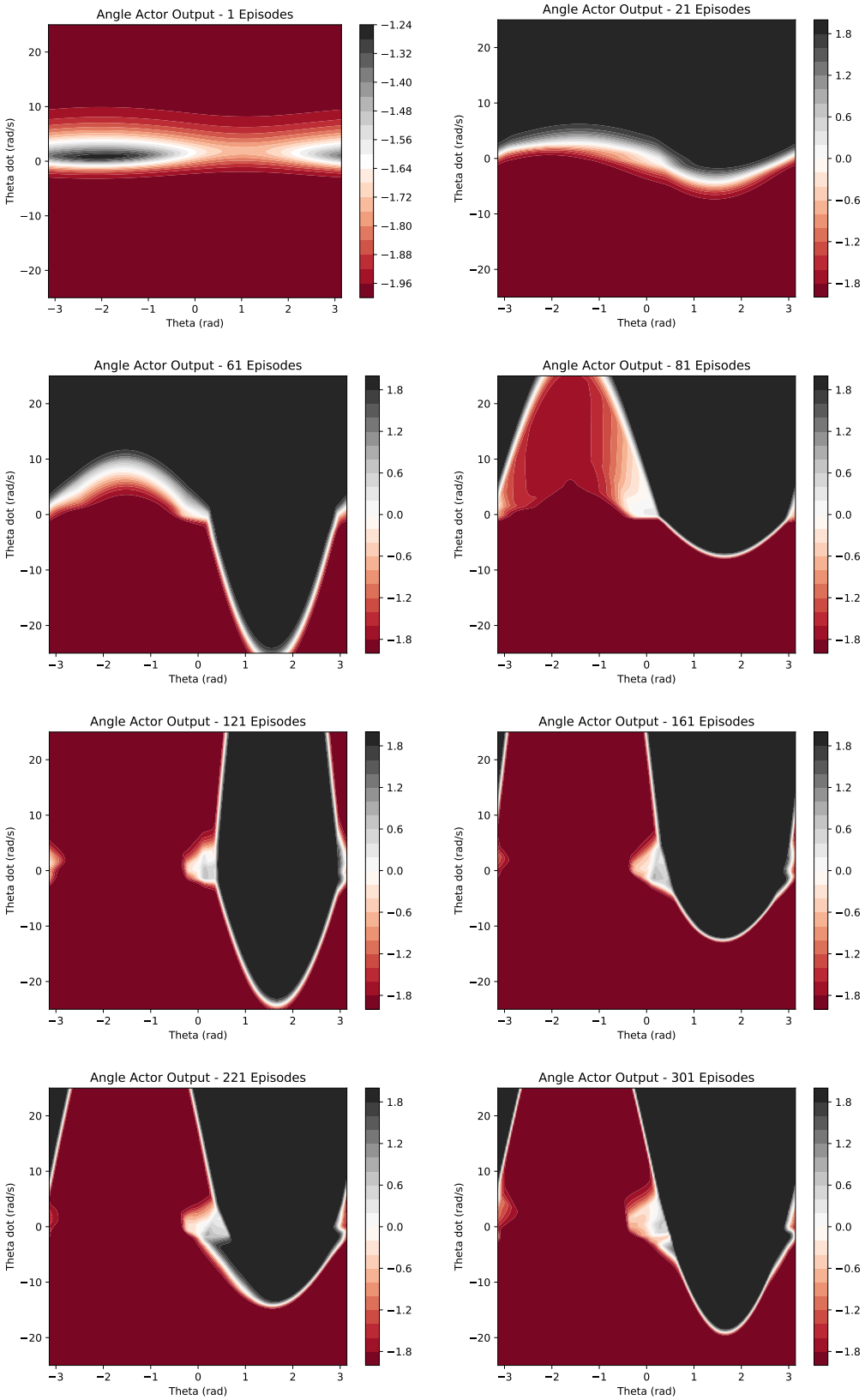


Figure 5.26: Angle Actor Output Progression

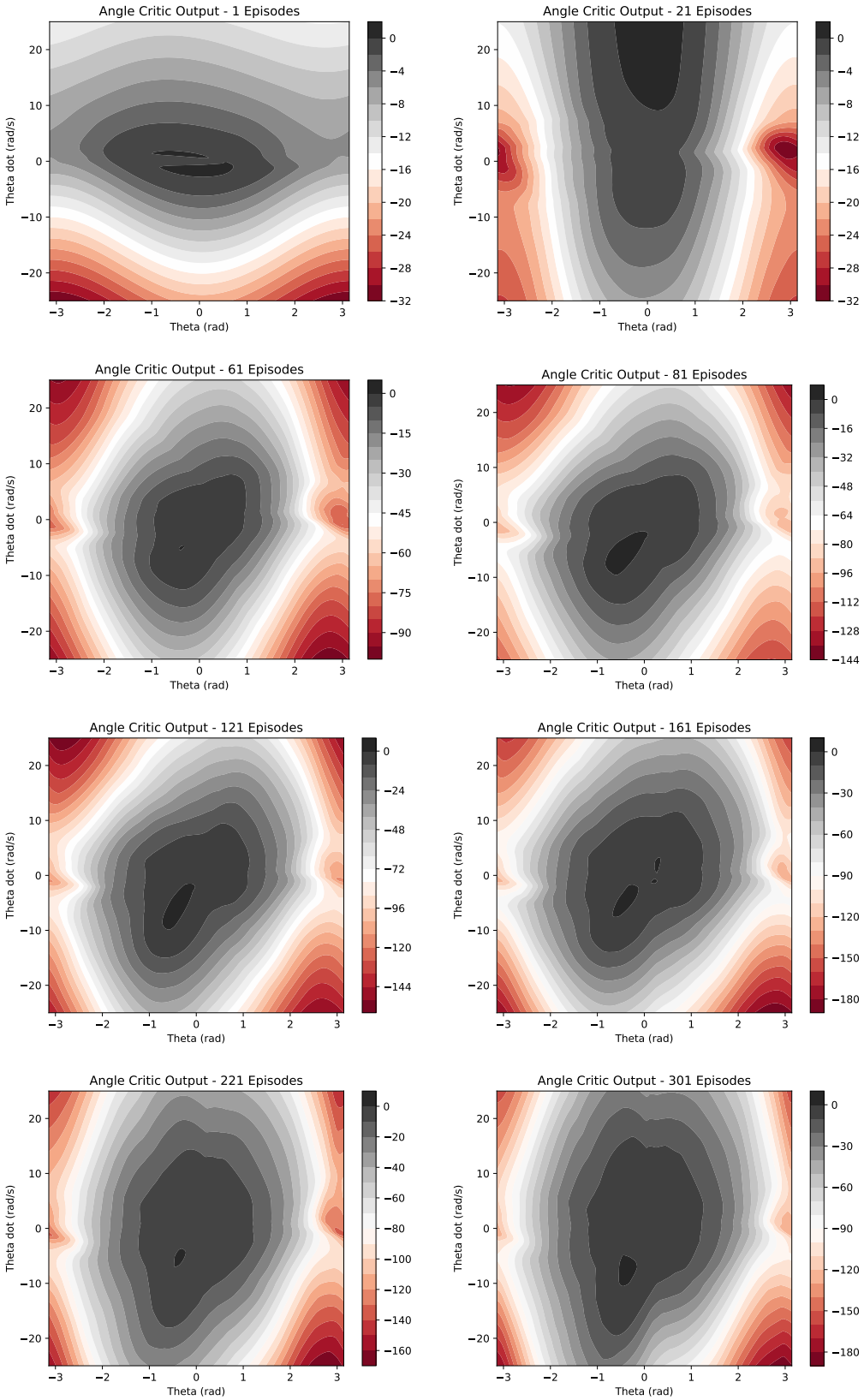


Figure 5.27: Angle Critic Output Progression

5.4.1 Three Actors Combined

After the three networks were trained, the best iteration of each (determined as rapid settling time, low or no overshoot, and decaying action value) was combined. At each time step, each actor produces its respective control component which merge together using Equations 5.20 through 5.23 as described previously. Figure 5.28 displays the response to eight varied initial states. Although the robot reaches the desired x, y, and θ position, it takes longer than in the individual tests above since the robot's movement is now divided amongst the three directions x, y, and θ. The actions still decay to 0.

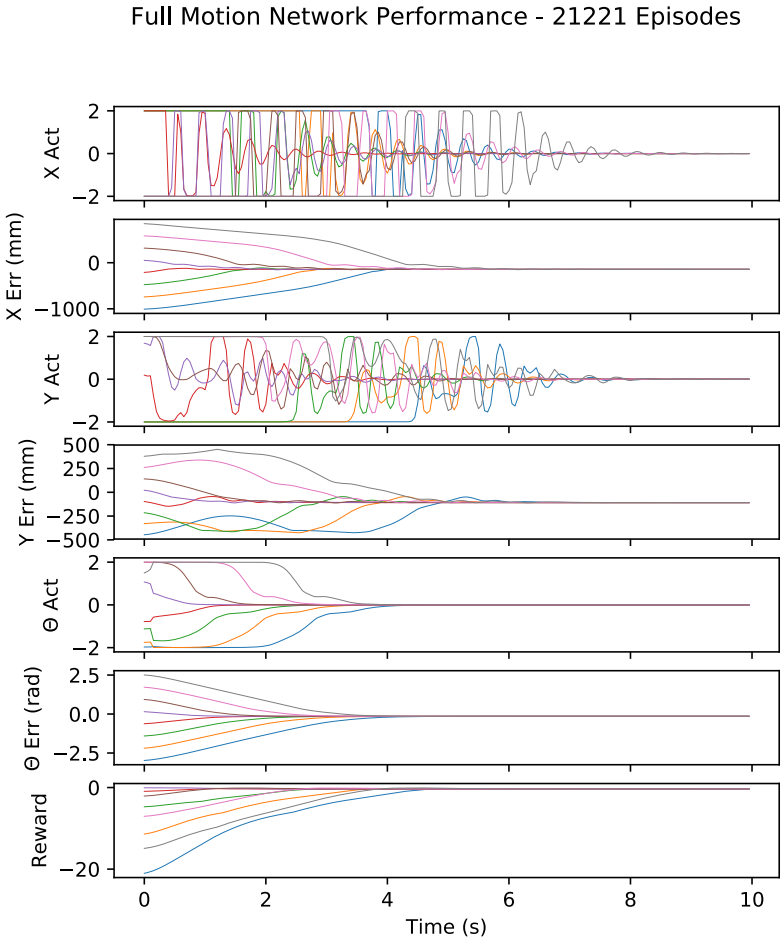


Figure 5.28: Combined Response

5.4.2 Single Actor Network

Since single actor approach required reduced learning rates to prevent training instability, the network trained for substantially longer before achieving reasonable performance. Figures 5.29 and 5.30 show that the learning "jump" lasts nearly 1,000 episodes versus 20-60 episodes in the three actor case. Notably, Figure 5.30 shows that, on average, the test reward continues to very slightly increase with further training at a rate of 1.6 reward per 1,000 additional episodes trained. In Figure 5.31, the average max Q overshoots the ideal maximum of 0 but settles to negative values with further training.

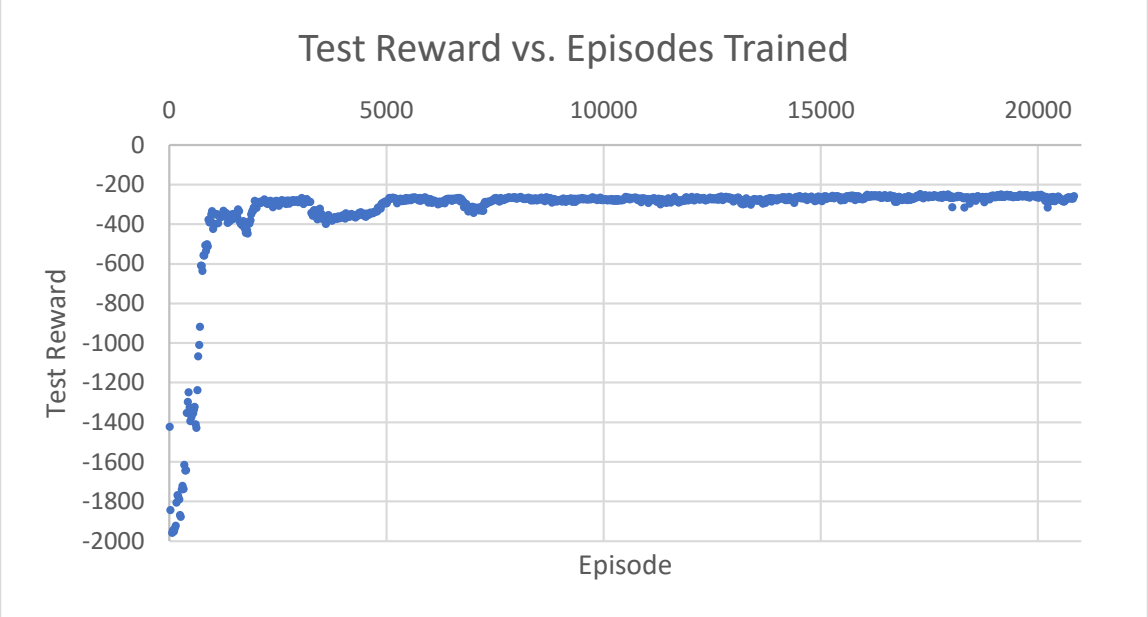


Figure 5.29: Training and Test Reward

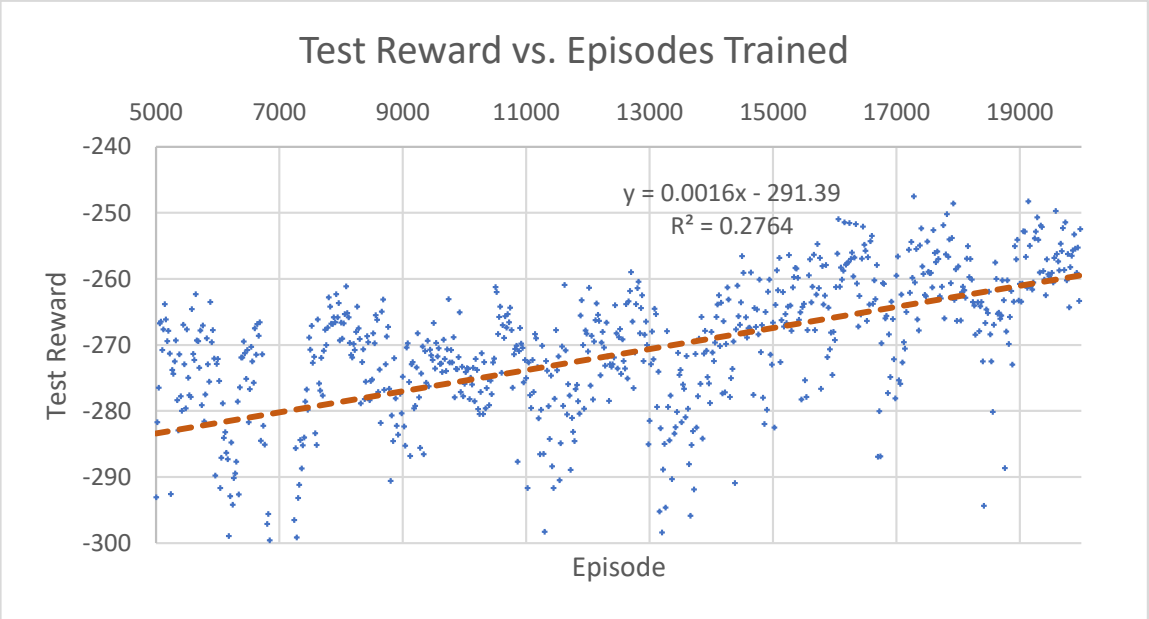


Figure 5.30: Training and Test Reward Zoomed

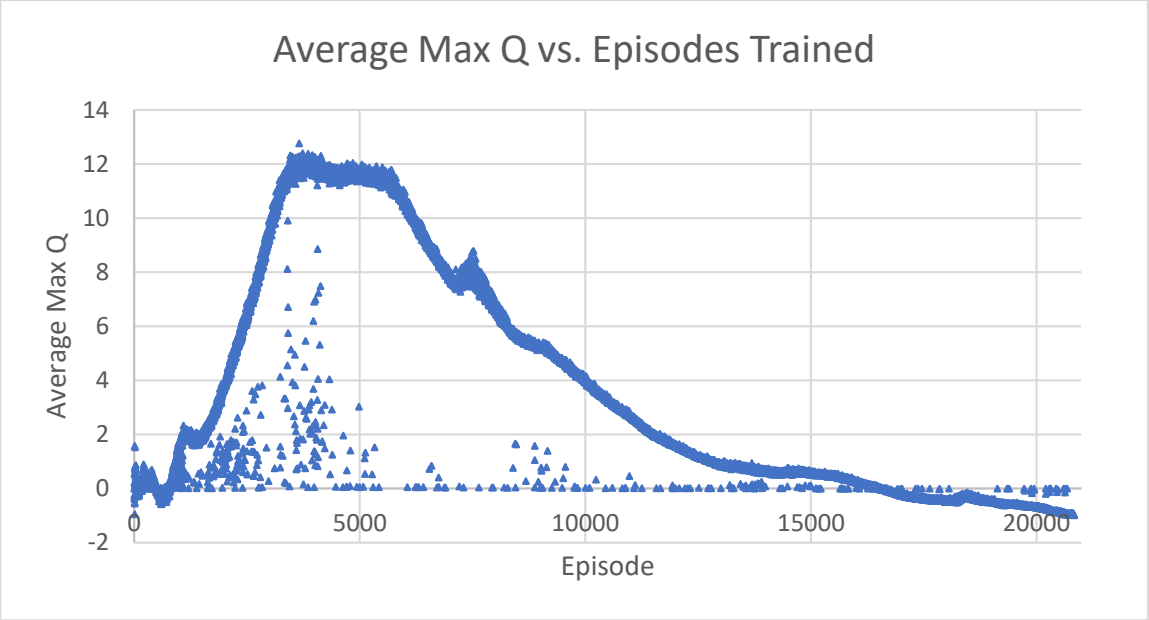


Figure 5.31: Average Max Q

Figure 5.32 shows the single actor response to eight various initial x, y, and θ values. Appendix I contains additional plots for other episodes trained. Although the

actor does successfully bring the robot to the set point, it fails to reduce the action to 0. Instead, the actions exhibit limit cycles, oscillating to keep the net velocity at 0. In this respect, the three actors approach provides the distinct advantage.

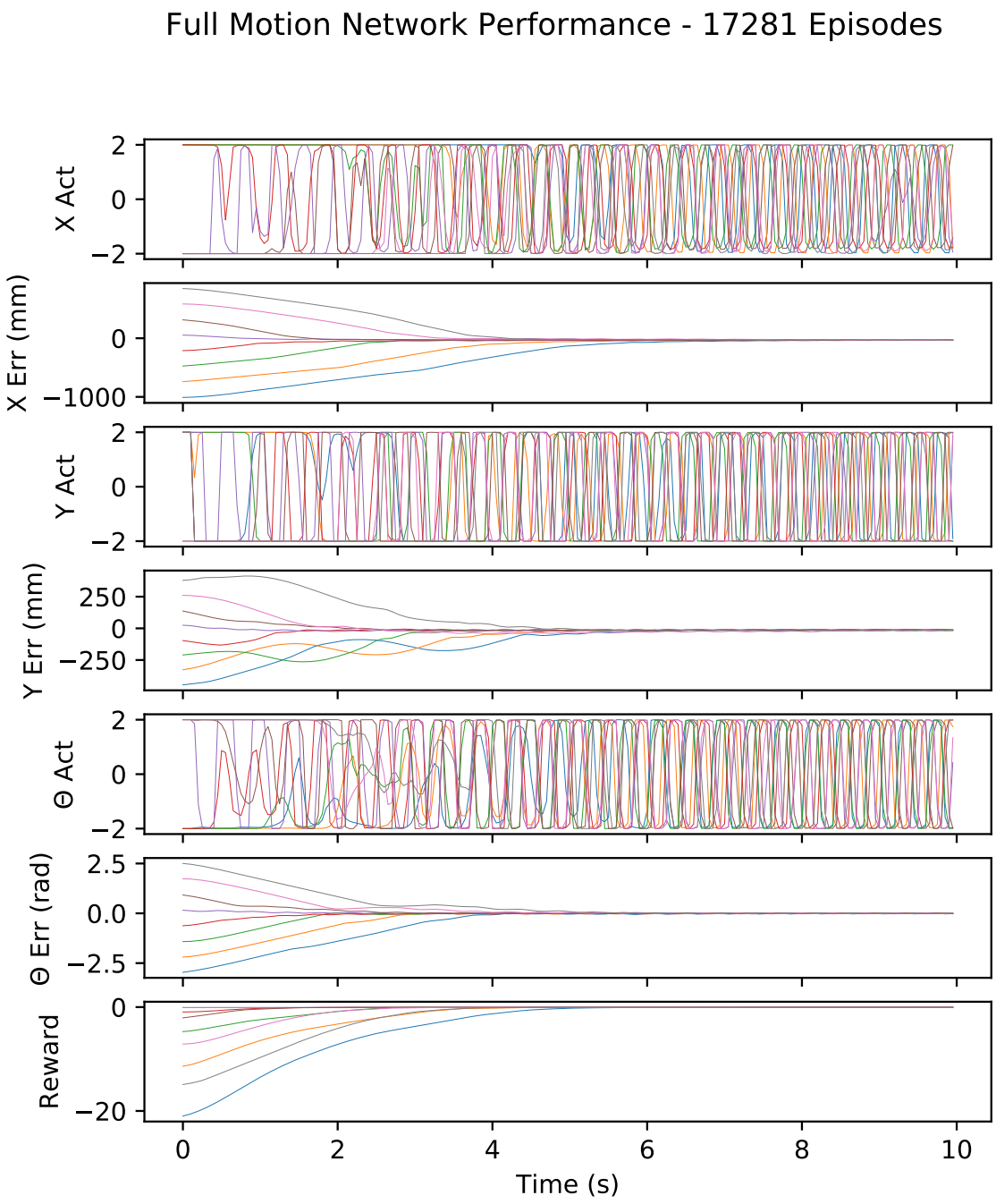


Figure 5.32: All Network Performance – 17281 Episodes

5.5 Conclusion

The deep deterministic policy gradient algorithm successfully solved position and orientation control of the robot using two different approaches. The three actors approach proved to be easier to tune, faster to converge, and better in performance versus the single actor case. Therefore, where possible, dividing a system by its orthogonal controls improves training convergence and hyper-parameter tuning and also allows partial retraining, a major advantage when obtaining experiences is costly. However, designers should consider the single actor method when processing speed is the limiting factor.

5.6 Future Work

The process of developing reward assignments for the environment could be improved by taking a more systemic approach than guess-and-check. Despite successfully achieving the goal of moving the robot to the desired position, it is unclear if the chosen reward calculations were anywhere near optimal.

Although DDPG does successfully solve varied environments, it still depends on proper reward assignment to achieve the desired policy. A heuristic algorithm for automatically scaling rewards would improve on the theme of generalization.

The simulation of the robot only accounts for some basic kinematics, but future designs could incorporate a much greater range of factors. The closer the simulation to reality, the easier it is to train the robot in simulation and expect similar performance when the learned network weights and biases are transferred to the real robot.

Finally, training the network directly on the real robot would allow complete omission of the simulation altogether at the expense of time and effort. However, the resulting policy would most accurately conform to the conditions and environments

of the application.

BIBLIOGRAPHY

[1] GARTT 2 Packs YPG 1800mAh 14.8V 70C 4S LiPo Battery. https://www.amazon.com/gp/product/B01N64JI77/ref=oh_aui_detailpage_o03_s00?ie=UTF8&psc=1. [Online; accessed June 1, 2018].

[2] Number of Possible Go Games. https://senseis.xmp.net/?NumberOfPossibleGoGames, 2018. [Online; accessed May 31, 2018].

[3] Adafruit. Adafruit 9-DOF IMU Breakout. https://www.adafruit.com/product/1714, 2014. [Online; accessed May 15, 2018].

[4] AliExpress. DC-DC CC CV Buck Converter. https://www.aliexpress.com/item/DC-DC-CC-CV-Buck-Converter-Volt-Step-Down-12V-19V-24V-Car-Laptop-Power-Supply/32822603345.html?spm=2114.search0104.3.135.191f73dfqonWDU&ws_ab_test=searchweb0_0,searchweb201602_2_10152_10151_10065_10344_10130_10068_10324_10547_10342_10325_10546_10343_10340_10548_10341_10545_10696_10084_10083_10618_10307_10059_308_100031_10103_10624_10623_10622_10621_10620,searchweb201603_32,ppcSwitch_5&algo_expid=30a33cc9-3acf-4021-b514-6ba550085dd9-19&algo_pvid=30a33cc9-3acf-4021-b514-6ba550085dd9&priceBeautifyAB=0, 2018. [Online; accessed May 11, 2018].

[5] American Go Association. A Brief History of Go. http://www.usgo.org/brief-history-go. [Online; accessed May 31, 2018].

[6] L. Armesto. How to generate a PPM signal with Arduino to control a servo?

http://robotica.webs.upv.es/en/how-to-generate-a-ppm-signal-with-arduino-to-control-a-servo/, 2015. [Online; accessed May 14, 2018].

[7] Autodesk. EAGLE version 7.4. http://eagle.autodesk.com/eagle/software-versions/5, 2015. [Online; accessed June 1, 2018].

[8] J. Bickford. Mechanisms for Intermittent Motion. http://ebooks.library.cornell.edu/k/kmoddl/pdf/002_010.pdf, 1972. [Online; accessed June 1, 2018].

[9] British Go Association. A Comparison of Chess and Go. https://www.britgo.org/learners/chessgo.html. [Online; accessed May 31, 2018].

[10] British Go Association. AlphaGo Match against Fan Hui. https://www.britgo.org/deepmind2016, Aug 2017. [Online; accessed May 31, 2018].

[11] British Go Association. History of Go-playing Programs. https://www.britgo.org/computergo/history, Jan 2018. [Online; accessed May 31, 2018].

[12] Cal Poly. Roborodentia - Cal Poly, San Luis Obispo. http://roborodentia.calpoly.edu/, 2018. [Online; accessed May 11, 2018].

[13] Dassault Systemes. Solidworks. https://www.solidworks.com, 2018. [Online; accessed June 1, 2018].

[14] J. Davies. 3D Modeling CAD Software. https://www.3dhubs.com/knowledge-base/3d-modeling-cad-software. [Online; accessed June 1, 2018].

[15] Digikey. STMicroelectronics L298N. https://www.digikey.com/product-detail/en/stmicroelectronics/L298N/497-1395-5-ND/585918, 2018. [Online; accessed June 1, 2018].

[16] Digikey. Texas Instruments TCA9554ADWR. https://www.digikey.com/product-detail/en/texas-instruments/TCA9554ADWR/296-45456-1-ND/6817816, 2018. [Online; accessed June 1, 2018].

[17] J. Duchi, E. Hazan, and Y. Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, August 2011.

[18] P. Emami. Deep Deterministic Policy Gradients in TensorFlow. https://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html, 2016. [Online; accessed May 27, 2018].

[19] R. P. Foundation. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Online; accessed June 1, 2018].

[20] I. Free Software Foundation. GCC, the GNU Compiler Collection. https://gcc.gnu.org, 2018. [Online; accessed June 1, 2018].

[21] W.-T. Fu and J. Anderson. Solving the Credit Assignment Problem: Interaction of Explicit and Implicit Learning with Internal and External State Information, Nov 2008. [Online; accessed May 31, 2018].

[22] Hasbro. Nerf Rival. https://www.hasbro.com/en-us/brands/nerfrival, 2018. [Online; accessed June 1, 2018].

[23] S. Huang. Introduction to Various Reinforcement Learning Algorithms. Part I (Q-Learning, SARSA, DQN, DDPG). https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287, 2018. [Online; accessed May 27, 2018].

[24] B. E. Ilon. Wheels for a course stable selfpropelling vehicle movable in any desired direction on the ground or some other base, Apr 1975.

[25] Intel. Intel Core i5-4670K Processor. https://ark.intel.com/products/75048/Intel-Core-i5-4670K-Processor-6M-Cache-up-to-3_80-GHz. [Online; accessed June 9, 2018].

[26] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ArXiv e-prints*, Feb. 2015.

[27] A. Karpathy. Deep Reinforcement Learning: Pong from Pixels. http://karpathy.github.io/2016/05/31/rl/, 2016. [Online; accessed May 30, 2018].

[28] R. R. Labbe. Kalman and Bayesian Filters in Python, Sep 2017.

[29] Y. Li, S. Dai, Y. Zheng, F. Tian, and X. Yan. Modeling and Kinematics Simulation of a Mecanum Wheel Platform in RecurDyn. https://www.hindawi.com/journals/jr/2018/9373580/, Jan 2018. [Online; accessed June 1, 2018].

[30] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *ArXiv e-prints*, Sept. 2015.

[31] E. N. Lorenz. Deterministic Nonperiodic Flow. *Journal of Atmospheric Sciences*, 20:130–148, Mar. 1963.

[32] MakerGear. MakerGear M2 3D Printer. `https://www.makergear.com/products/m2`, 2018. [Online; accessed May 25, 2018].

[33] MakerGeeks. ABS FILAMENT 1.75MM. `https://www.makergeeks.com/collections/abs-filament-1-75mm`. [Online; accessed June 1, 2018].

[34] T. Matiisen. DEMYSTIFYING DEEP REINFORCEMENT LEARNING. `http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/`, 2015. [Online; accessed May 27, 2018].

[35] J. McCullock. A Painless Q-Learning Tutorial. `http://mnemstudio.org/path-finding-q-learning-tutorial.htm`. [Online; accessed June 1, 2018].

[36] MechaMan. Working with L298N DC Motor Driver. `http://fritzing.org/projects/working-with-l298n-dc-motor-driver`, 2018. [Online; accessed May 13, 2018].

[37] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, Feb 2015.

[38] R. Negenborn. *Robot Localization and Kalman Filters*. PhD thesis, Institute of Information and Computing Sciences, Sep 2003.

[39] M. Nevada. FAQ: Hard & Soft Iron Correction for Magnetometer Measurements. https://ez.analog.com/docs/DOC-2544, 2014. [Online; accessed May 11, 2018].

[40] D. H. Nguyen and B. Widrow. Neural Networks for Self-Learning Control Systems. *IEEE Control Systems Magazine*, Apr 1990.

[41] Nvidia. GeForce GTX 980 Ti. https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980-ti, 2018. [Online; accessed June 9, 2018].

[42] OpenAI. DDPG Noise. 2017. [Online; accessed June 1, 2018].

[43] OpenAI. Pendulum v0. https://github.com/openai/gym/wiki/Pendulum-v0, 2018. [Online; accessed June 8, 2018].

[44] OpenAI Gym. OpenAI Gym. https://gym.openai.com/. [Online; accessed May 31, 2018].

[45] PCBWay. China PCB Prototype. https://www.pcbway.com, 2018. [Online; accessed June 1, 2018].

[46] Pololu. 70:1 Metal Gearmotor 37Dx70L mm with 64 CPR Encoder. https://www.pololu.com/product/2825. [Online; accessed June 1, 2018].

[47] Python Software Foundation. Python 3.6.0. https://www.python.org/downloads/release/python-360/, 2016. [Online; accessed June 1, 2018].

[48] M. Z. B. A. Rahman. OMNI DIRECTIONAL CONTROL ALGORITHM FOR MECANUM WHEEL. http://eprints.utem.edu.my/16543/1/Omni%20Directional%20Control%

20Algorithm%20For%20Mecanum%20Wheel%2024%20Pages.pdf, Jun 2014. [Online; accessed June 2, 2018].

[49] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for Activation Functions. *ArXiv e-prints*, Oct. 2017.

[50] B. Redwood. How does part orientation affect a 3D Print? https://www.3dhubs.com/knowledge-base/how-does-part-orientation-affect-3d-print. [Online; accessed June 1, 2018].

[51] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. Jan. 2014.

[52] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, H. Aja, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, S. Laurent, G. v. d. Driessche, T. Graepel, and D. Hassabis. Mastering the Game of Go without Human Knowledge, 2017.

[53] STMicroelectronics. STM32CubeMX. http://www.st.com/en/development-tools/stm32cubemx.html. [Online; accessed June 1, 2018].

[54] STMicroelectronics. STM32F446RE. http://www.st.com/en/microcontrollers/stm32f446re.html. [Online; accessed June 1, 2018].

[55] STMicroelectronics. LSM303DLHC Datasheet. http://www.st.com/resource/en/datasheet/DM00027543.pdf, 2013. [Online; accessed May 15, 2018].

[56] STMicroelectronics. STM32 Nucleo-64 User Manual. http://www.st.com/content/ccc/resource/technical/document/user_

manual/98/2e/fa/4b/e0/82/43/b7/DM00105823.pdf/files/DM00105823.pdf/jcr:content/translations/en.DM00105823.pdf, 2017. [Online; accessed May 11, 2018].

[57] STMicroelectronics. VL53L0X. http://www.st.com/en/imaging-and-photonics-solutions/vl53l0x.html, 2018. [Online; accessed May 15, 2018].

[58] STMicroelectronics. VL53L0X Datasheet. http://www.st.com/content/ccc/resource/technical/document/datasheet/group3/b2/1e/33/77/c6/92/47/6b/DM00279086/files/DM00279086.pdf/jcr:content/translations/en.DM00279086.pdf, 2018. [Online; accessed May 12, 2018].

[59] R. Sutton, D. McAllester, S. Singh, and Y. Mansour. *Policy Gradient Methods for Reinforcement Learning with Function Approximation*, pages 1057–1063. 1999.

[60] R. S. Sutton and A. G. Barton. *REINFORCEMENT LEARNING: an introduction*. MIT PRESS, 2018.

[61] D. Tedaldi, A. Pretto, and E. Menegatti. A robust and easy to implement method for imu calibration without external equipments. *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[62] TensorFlow. TensorFlow. https://www.tensorflow.org/. [Online; accessed May 31, 2018].

[63] TensorFlow. tf.train.AdamOptimizer. https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer#apply_gradients, May 2018. [Online; accessed June 1, 2018].

[64] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 2012.

[65] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Phys. Rev.*, 36:823–841, Sep 1930.

[66] F. Varesano. FreeIMU Magnetometer and Accelerometer Calibration GUI. `http://www.varesano.net/blog/fabio/freeimu-magnetometer-and-accelerometer-calibration-gui-alpha-version-out`, 2012. [Online; accessed May 11, 2018].

[67] Windows. Windows 10 Pro. `https://www.microsoft.com/en-us/p/windows-10-pro/df77x4d43rkt/48DN`, 2018. [Online; accessed June 9, 2018].

[68] F. Yu. Deep Q Network vs Policy Gradients - An Experiment on VizDoom with Keras. `https://flyyufelix.github.io/2017/10/12/dqn-vs-pg.html`, 2017. [Online; accessed June 9, 2018].

[69] J.-W. Zuyderduyn. Boss/Base Loft feature. `https://learnsolidworks.com/solidworks_features/bossbase-loft-feature`, 2016. [Online; accessed June 1, 2018].

APPENDICES

Appendix A

MECHANICAL PARTS – BILL OF MATERIALS

Table A.1: Mechanical Bill of Materials

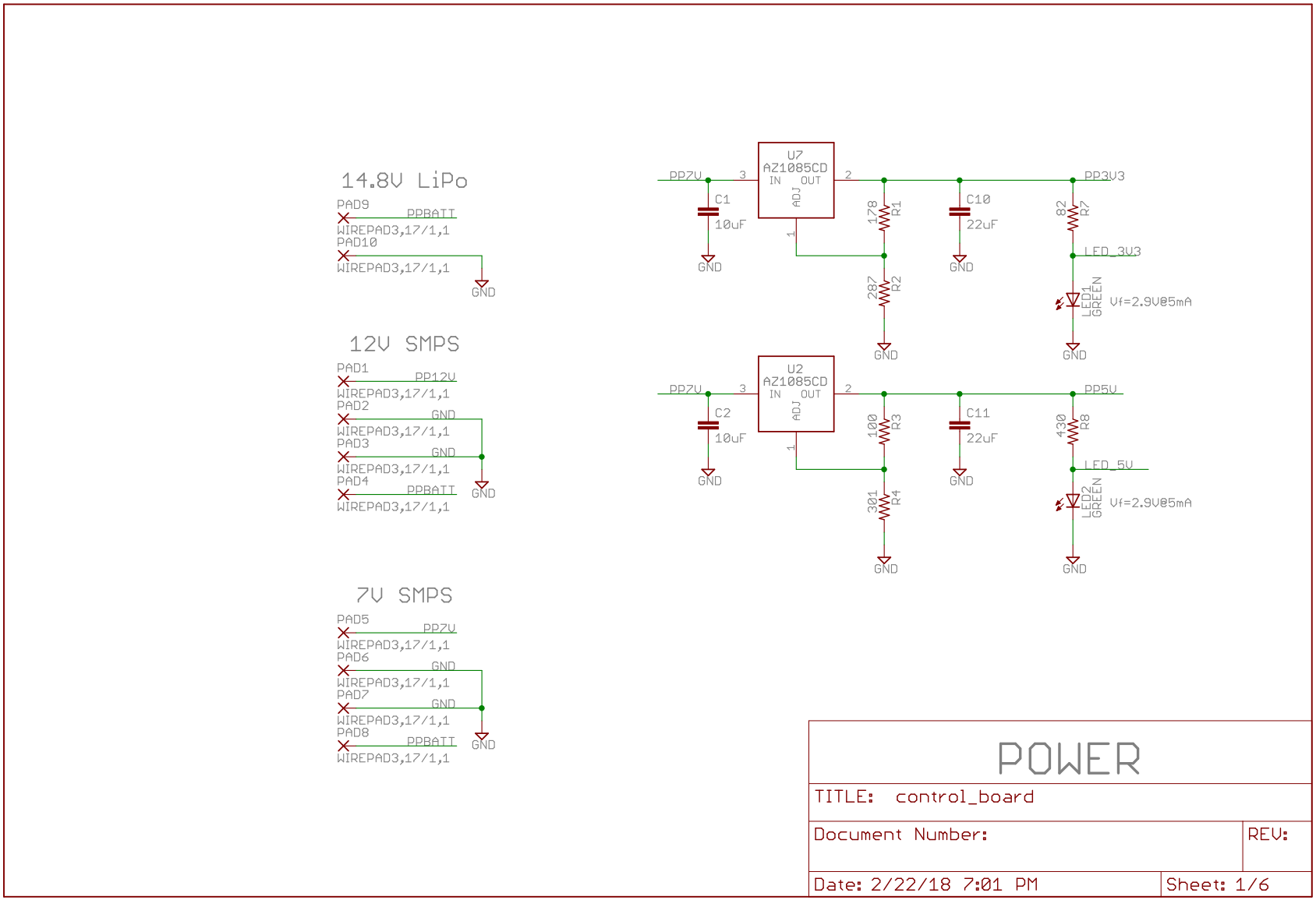
ITEM NO.	PART FILE NAME	QTY.
1.	battery	1
2.	battery_mount	1
3.	blower	1
4.	bottom_plate	1
5.	buck_converter	2
6.	button	1
7.	button_mount	1
8.	control_brd	1
9.	dual_flywheel_cage_fixed	1
10.	electrical_mount	1
11.	electrical_mount_bracket_angled	2
12.	hopper_bar	2
13.	hopper_connector	2
14.	imu	1
15.	launcher_motor_small	2
16.	m2-12	2
17.	m2-8	12
18.	m2-nut	14

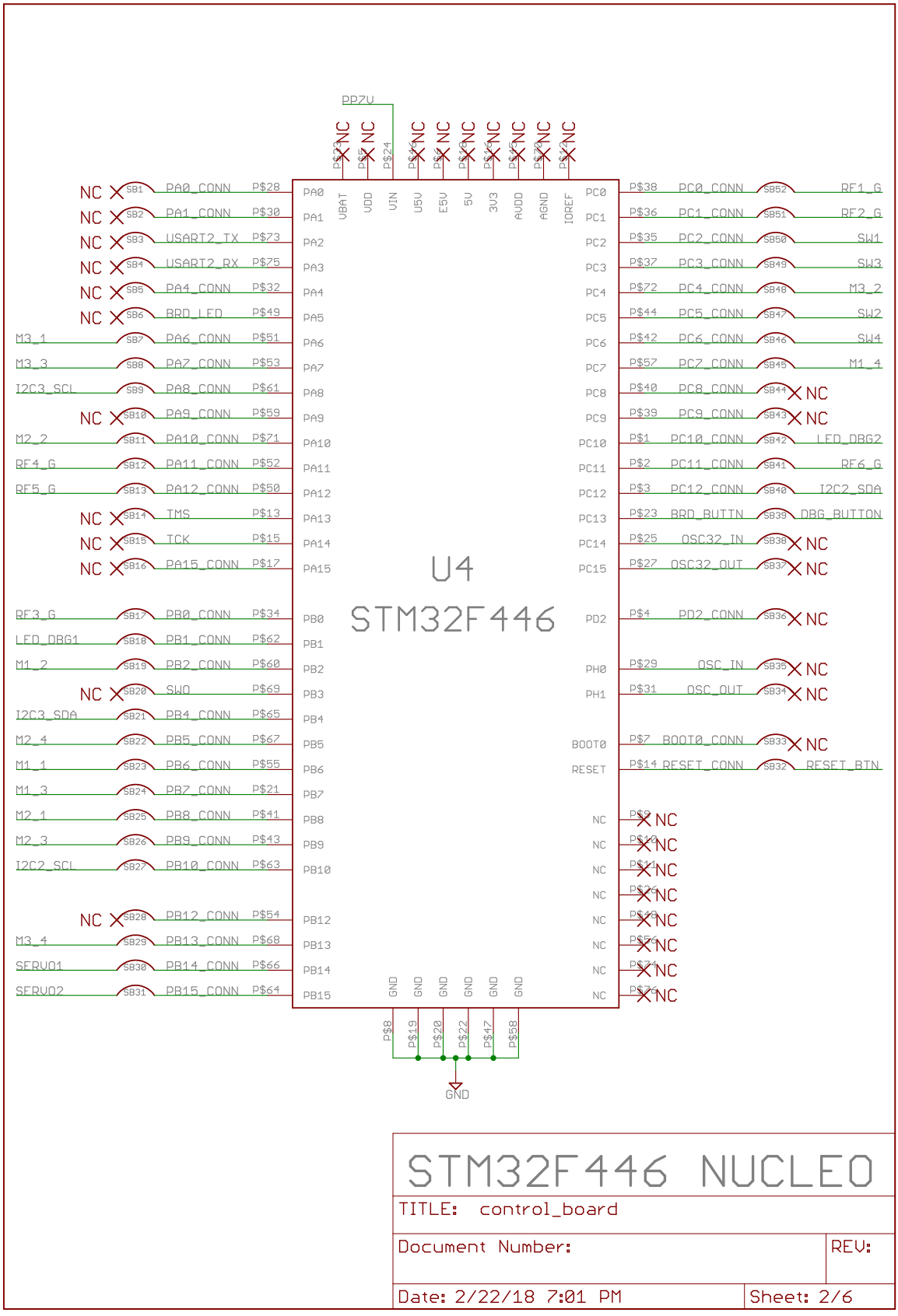
ITEM NO.	PART FILE NAME	QTY.
19.	m3-12	86
20.	m3-20	2
21.	m3-25	4
22.	m3-30	16
23.	m3-35	4
24.	m3-40	5
25.	m3-45	12
26.	m3-nut	115
27.	m3-standoff-5	4
28.	m4-12	8
29.	m4-nut	8
30.	mecanum_coupler	4
31.	mecanum_wheel	4
32.	motor_clamp	2
33.	motor_driver	3
34.	NUCLEO_F446RE	1
35.	nylon_standoff	36
36.	pusher_mount	1
37.	rangefinder	4
38.	rangefinder_mount	4
39.	Raspberry Pi 3 Light Version	1
40.	servo	1
41.	servo_hopper	1
42.	servo_hopper_gate	1
43.	servo_hopper_mount_base_mini	1

ITEM NO.	PART FILE NAME	QTY.
44.	servo_hopper_mount_base_right_mini	1
45.	servo_hopper_mount_bracket	2
46.	servo_hopper_servo_mount	1
47.	servo_horn	1
48.	switch	1
49.	switch_mount	1
50.	venturi_loader_fixed	1
51.	wheel_grip_small	2
52.	wheel_motor	4
53.	wheel_motor_bracket	4
	TOTAL	394

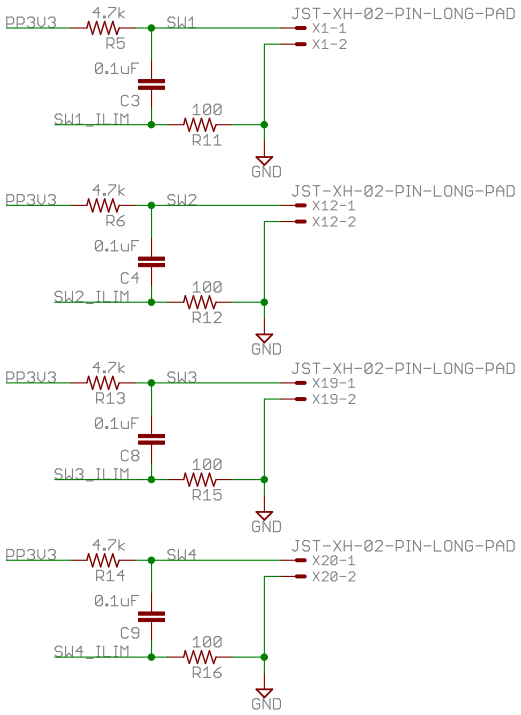
Appendix B

INTERCONNECT PCB SCHEMATIC

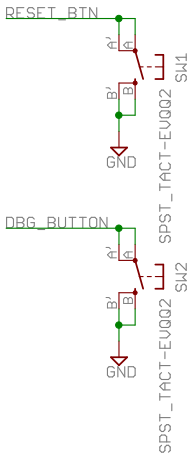

14.8V LiPo
PAD9
PPBATT
WIREPAD3,17/1,1
PAD10
WIREPAD3,17/1,1
GND
12V SMPS
PAD1
PP12V
WIREPAD3,17/1,1
PAD2
GND
WIREPAD3,17/1,1
PAD3
GND
WIREPAD3,17/1,1
PAD4
PPBATT
WIREPAD3,17/1,1
7V SMPS
PAD5
PP7V
WIREPAD3,17/1,1
PAD6
GND
WIREPAD3,17/1,1
PAD7
GND
WIREPAD3,17/1,1
PAD8
PPBATT
WIREPAD3,17/1,1
U7
AZ1085CD
IN
OUT
ADJ
PP7V
C1
10uF
GND
178
R1
287
R2
C10
22uF
82
R7
PP3V3
LED_3V3
LED1
GREEN
Vf=2.9V@5mA
U2
AZ1085CD
C2
10uF
100
R3
301
R4
C11
22uF
430
R8
PP5V
LED_5V
LED2
GREEN
Vf=2.9V@5mA
POWER
TITLE: control_board
Document Number:
REV:
Date: 2/22/18 7:01 PM
Sheet: 1/6


U4
STM32F446
PA0
PA1
PA2
PA3
PA4
PA5
PA6
PA7
PA8
PA9
PA10
PA11
PA12
PA13
PA14
PA15
PB0
PB1
PB2
PB3
PB4
PB5
PB6
PB7
PB8
PB9
PB10
PB12
PB13
PB14
PB15
PC0
PC1
PC2
PC3
PC4
PC5
PC6
PC7
PC8
PC9
PC10
PC11
PC12
PC13
PC14
PC15
PD2
PH0
PH1
BOOT0
RESET
VBAT
VDD
VIN
U5V
E5V
5V
3V3
AVDD
AGND
IOREF
GND
PA0_CONN
PA1_CONN
USART2_TX
USART2_RX
PA4_CONN
BRD_LED
PA6_CONN
PA7_CONN
PA8_CONN
PA9_CONN
PA10_CONN
PA11_CONN
PA12_CONN
TMS
TCK
PA15_CONN
PB0_CONN
PB1_CONN
PB2_CONN
SWO
PB4_CONN
PB5_CONN
PB6_CONN
PB7_CONN
PB8_CONN
PB9_CONN
PB10_CONN
PB12_CONN
PB13_CONN
PB14_CONN
PB15_CONN
PC0_CONN
PC1_CONN
PC2_CONN
PC3_CONN
PC4_CONN
PC5_CONN
PC6_CONN
PC7_CONN
PC8_CONN
PC9_CONN
PC10_CONN
PC11_CONN
PC12_CONN
BRD_BUTTN
OSC32_IN
OSC32_OUT
PD2_CONN
OSC_IN
OSC_OUT
BOOT0_CONN
RESET_CONN
RF1_G
RF2_G
SW1
SW3
M3_2
SW2
SW4
M1_4
LED_DBG2
RF6_G
I2C2_SDA
DBG_BUTTON
RESET_BTN
RF3_G
LED_DBG1
M1_2
I2C3_SDA
M2_4
M1_1
M1_3
M2_1
M2_3
I2C2_SCL
M3_4
SERVO1
SERVO2
M3_1
M3_3
I2C3_SCL
M2_2
RF4_G
RF5_G
NC
STM32F446 NUCLEO
TITLE: control_board
Document Number:
REV:
Date: 2/22/18 7:01 PM
Sheet: 2/6

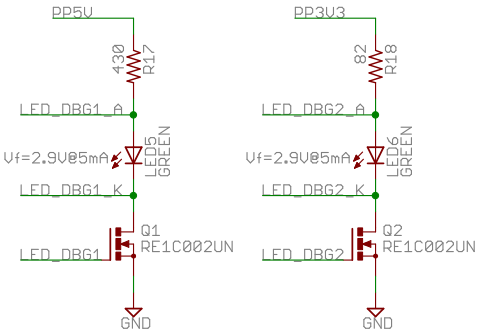
Microswitches



Debug Buttons

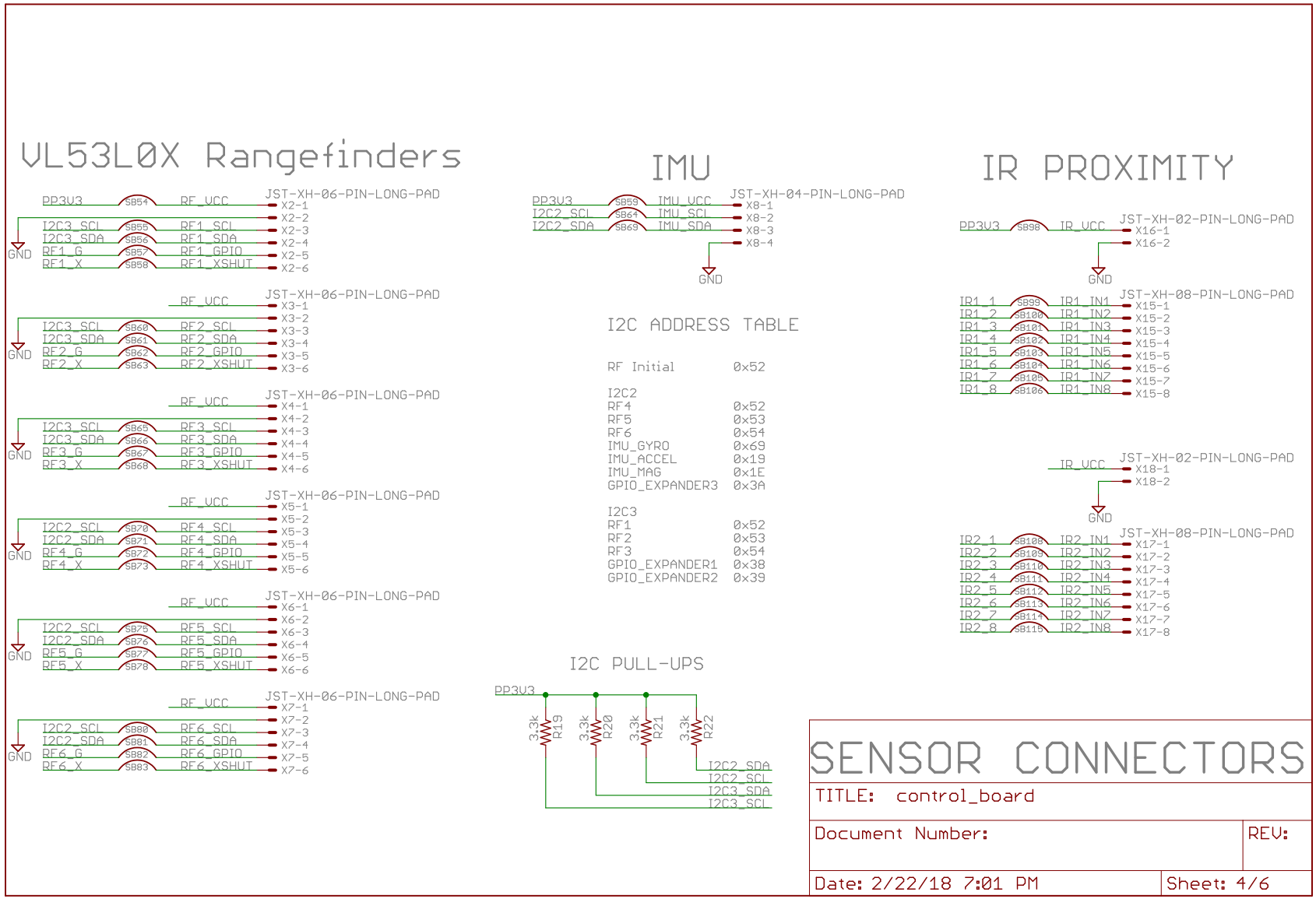


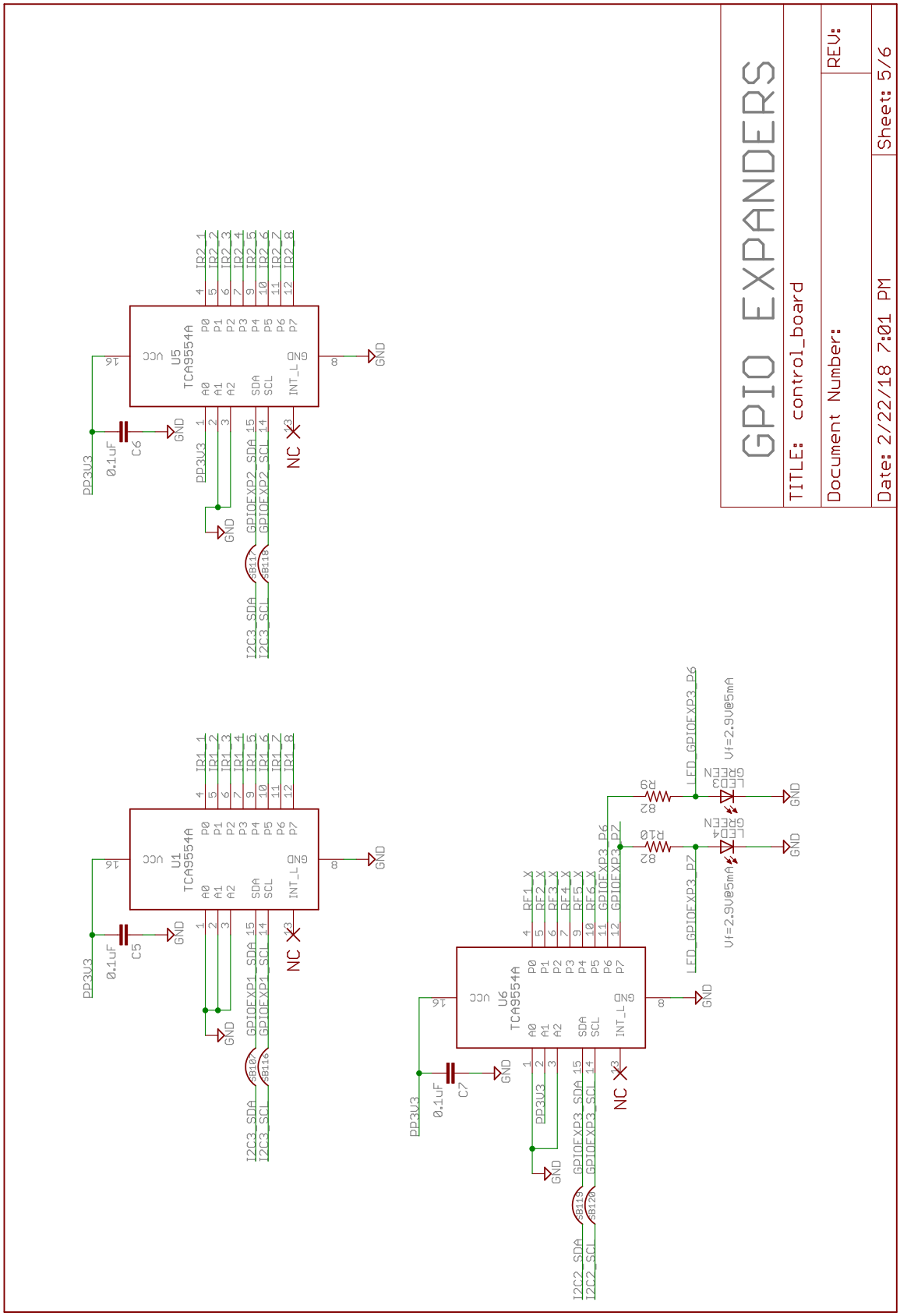
Debug LEDs



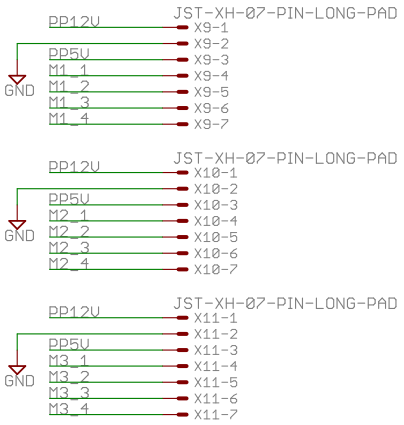
Switches & LEDs

TITLE: control_board	
Document Number:	REV:
Date: 2/22/18 7:01 PM	Sheet: 3/6

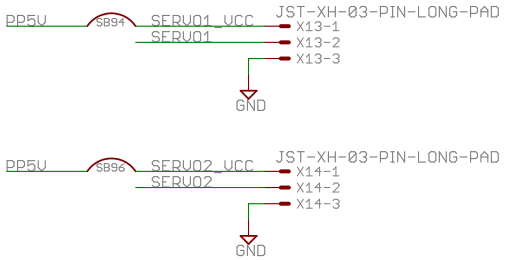

VL53L0X Rangefinders
PP3V3
SB54
RF_VCC
JST-XH-06-PIN-LONG-PAD
X2-1
X2-2
I2C3_SCL
SB55
RF1_SCL
X2-3
I2C3_SDA
SB56
RF1_SDA
X2-4
GND
RF1_G
SB57
RF1_GPIO
X2-5
RF1_X
SB58
RF1_XSHUT
X2-6
RF_VCC
JST-XH-06-PIN-LONG-PAD
X3-1
X3-2
I2C3_SCL
SB60
RF2_SCL
X3-3
I2C3_SDA
SB61
RF2_SDA
X3-4
GND
RF2_G
SB62
RF2_GPIO
X3-5
RF2_X
SB63
RF2_XSHUT
X3-6
RF_VCC
JST-XH-06-PIN-LONG-PAD
X4-1
X4-2
I2C3_SCL
SB65
RF3_SCL
X4-3
I2C3_SDA
SB66
RF3_SDA
X4-4
GND
RF3_G
SB67
RF3_GPIO
X4-5
RF3_X
SB68
RF3_XSHUT
X4-6
RF_VCC
JST-XH-06-PIN-LONG-PAD
X5-1
X5-2
I2C2_SCL
SB70
RF4_SCL
X5-3
I2C2_SDA
SB71
RF4_SDA
X5-4
GND
RF4_G
SB72
RF4_GPIO
X5-5
RF4_X
SB73
RF4_XSHUT
X5-6
RF_VCC
JST-XH-06-PIN-LONG-PAD
X6-1
X6-2
I2C2_SCL
SB75
RF5_SCL
X6-3
I2C2_SDA
SB76
RF5_SDA
X6-4
GND
RF5_G
SB77
RF5_GPIO
X6-5
RF5_X
SB78
RF5_XSHUT
X6-6
RF_VCC
JST-XH-06-PIN-LONG-PAD
X7-1
X7-2
I2C2_SCL
SB80
RF6_SCL
X7-3
I2C2_SDA
SB81
RF6_SDA
X7-4
GND
RF6_G
SB82
RF6_GPIO
X7-5
RF6_X
SB83
RF6_XSHUT
X7-6
IMU
PP3V3
SB59
IMU_VCC
JST-XH-04-PIN-LONG-PAD
X8-1
I2C2_SCL
SB64
IMU_SCL
X8-2
I2C2_SDA
SB69
IMU_SDA
X8-3
X8-4
GND
I2C ADDRESS TABLE
RF Initial 0x52
I2C2
RF4 0x52
RF5 0x53
RF6 0x54
IMU_GYRO 0x69
IMU_ACCEL 0x19
IMU_MAG 0x1E
GPIO_EXPANDER3 0x3A
I2C3
RF1 0x52
RF2 0x53
RF3 0x54
GPIO_EXPANDER1 0x38
GPIO_EXPANDER2 0x39
I2C PULL-UPS
PP3V3
3.3k
R19
3.3k
R20
3.3k
R21
3.3k
R22
I2C2_SDA
I2C2_SCL
I2C3_SDA
I2C3_SCL
IR PROXIMITY
PP3V3
SB98
IR_VCC
JST-XH-02-PIN-LONG-PAD
X16-1
X16-2
GND
IR1_1
SB99
IR1_IN1
JST-XH-08-PIN-LONG-PAD
X15-1
IR1_2
SB100
IR1_IN2
X15-2
IR1_3
SB101
IR1_IN3
X15-3
IR1_4
SB102
IR1_IN4
X15-4
IR1_5
SB103
IR1_IN5
X15-5
IR1_6
SB104
IR1_IN6
X15-6
IR1_7
SB105
IR1_IN7
X15-7
IR1_8
SB106
IR1_IN8
X15-8
IR_VCC
JST-XH-02-PIN-LONG-PAD
X18-1
X18-2
GND
IR2_1
SB108
IR2_IN1
JST-XH-08-PIN-LONG-PAD
X17-1
IR2_2
SB109
IR2_IN2
X17-2
IR2_3
SB110
IR2_IN3
X17-3
IR2_4
SB111
IR2_IN4
X17-4
IR2_5
SB112
IR2_IN5
X17-5
IR2_6
SB113
IR2_IN6
X17-6
IR2_7
SB114
IR2_IN7
X17-7
IR2_8
SB115
IR2_IN8
X17-8
SENSOR CONNECTORS
TITLE: control_board
Document Number:
REV:
Date: 2/22/18 7:01 PM
Sheet: 4/6


GPIO EXPANDERS
TITLE: control_board
Document Number:
REV:
Date: 2/22/18 7:01 PM
Sheet: 5/6
U5
TCA9554A
U1
TCA9554A
U6
TCA9554A
VCC
GND
SDA
SCL
INT_L
A0
A1
A2
P0
P1
P2
P3
P4
P5
P6
P7
3P3V3
0.1uF
C6
C5
C7
NC
I2C3_SDA
I2C3_SCL
I2C2_SDA
I2C2_SCL
GPIOEXP2_SDA
GPIOEXP2_SCL
GPIOEXP1_SDA
GPIOEXP1_SCL
GPIOEXP3_SDA
GPIOEXP3_SCL
IR2_1
IR2_2
IR2_3
IR2_4
IR2_5
IR2_6
IR2_7
IR2_8
IR1_1
IR1_2
IR1_3
IR1_4
IR1_5
IR1_6
IR1_7
IR1_8
RF1_X
RF2_X
RF3_X
RF4_X
RF5_X
RF6_X
GPIOEXP3_P6
GPIOEXP3_P7
LED_GPIOEXP3_P6
LED_GPIOEXP3_P7
R9
R10
82
Vf=2.9V@5mA

MOTOR DRIVERS



SERVOS



MOTOR DRIVERS	
TITLE: control_board	
Document Number:	REV:
Date: 2/22/18 7:01 PM	Sheet: 6/6

Appendix C

INTERCONNECT PCB LAYOUT

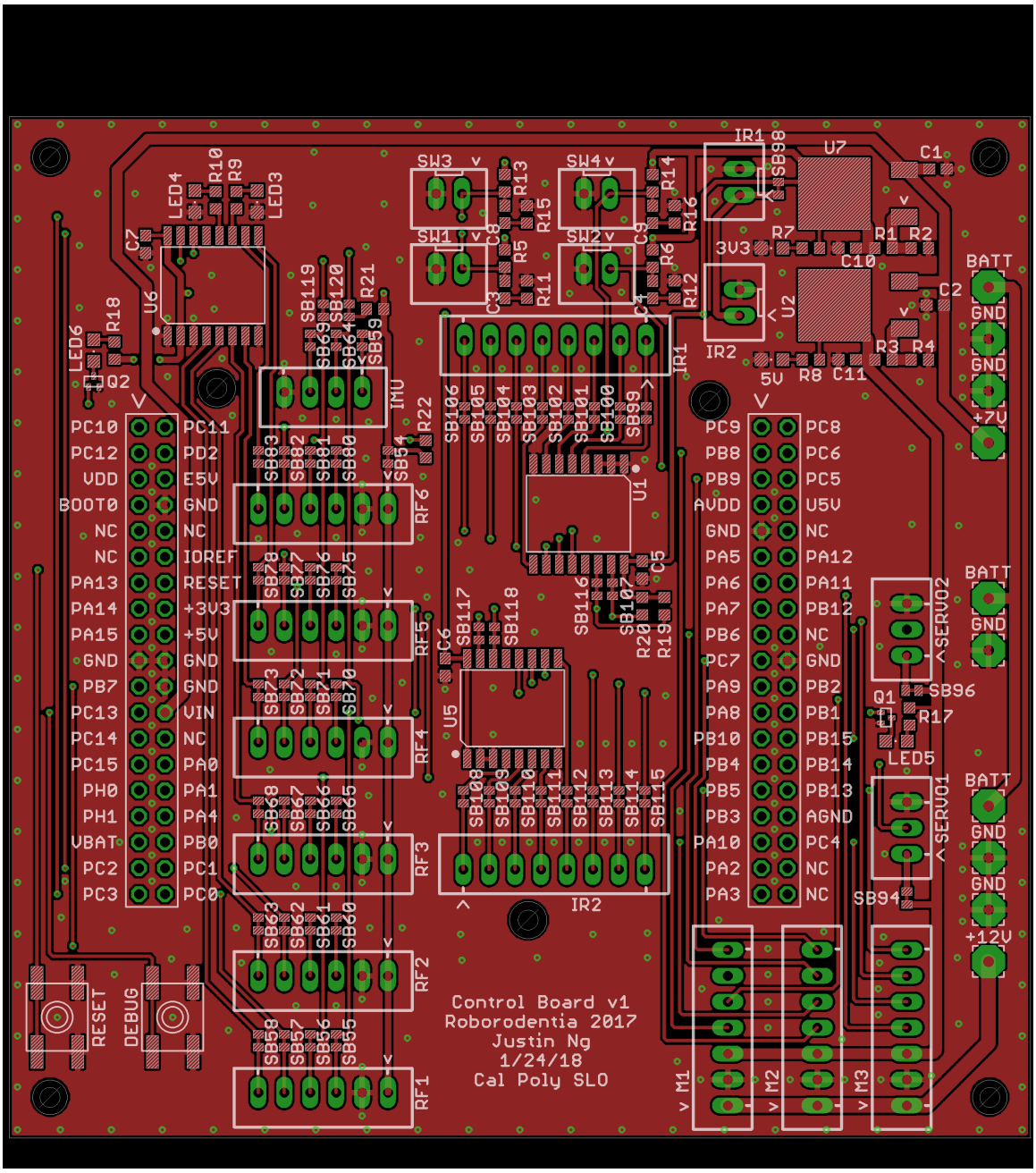


Figure C.1: Interconnect PCB Layout – Top Layer

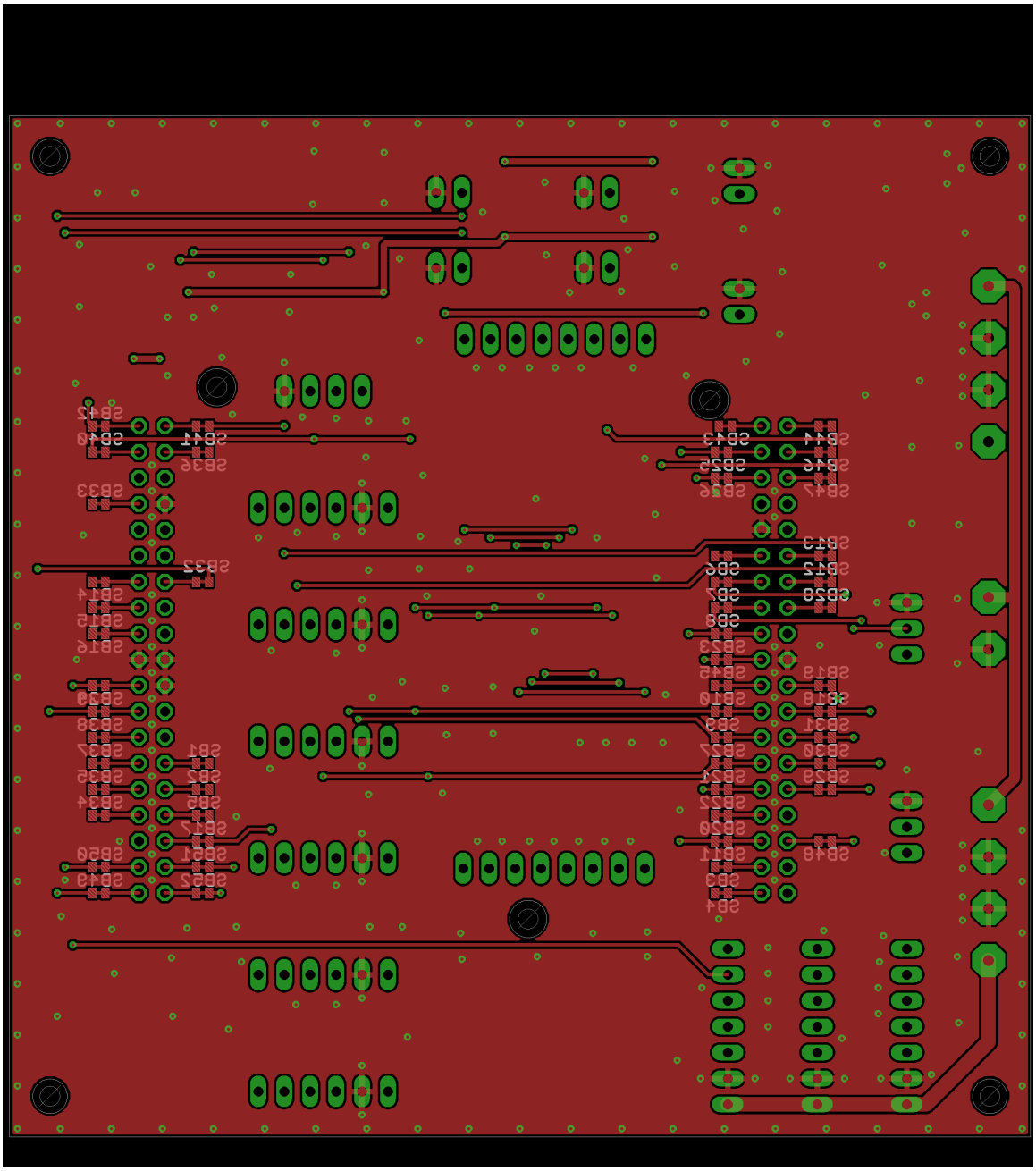


Figure C.2: Interconnect PCB Layout – Bottom Layer

Appendix D

INTERCONNECT PCB BILL OF MATERIALS

Table D.1: Interconnect PCB Bill of Materials

Qty	Value	Parts	Description	DIGIKEY PN
2	AZ1085CD	U2, U7	AZ1085C	AZ1085CD-ADJTRG1DICT-ND
7	0.1uF	C3, C4, C5, C6, C7, C8, C9	CAPACITOR	445-5667-1-ND
2	10uF3	C1, C2	CAPACITOR	490-13248-1-ND
2	22uF	C10, C11	CAPACITOR	490-10476-1-ND
3	TCA9554A	U1, U5, U6	I2C GPIO Expander	296-45456-1-ND
6		X1, X12, X16, X18, X19, X20	JST XH 2 Pin	455-2247-ND
2		X13, X14	JST XH 3 Pin	455-2248-ND
1		X8	JST XH 4 Pin	455-2249-ND
6		X2, X3, X4, X5, X6, X7	JST XH 6 Pin	455-2271-ND
3		X9, X10, X11	JST XH 7 Pin	455-2252-ND
2		X15, X17	JST XH 8 Pin	455-2251-ND
6	GREEN	LED1, LED2, LED3, LED4, LED5, LED6	LED	732-4971-1-ND
2	RE1C002UN	Q1, Q2	Logic-level N-FET	RE1C002UNTCLCT-ND
4	82	R7, R9, R10, R18	RESISTOR	311-82.0HRCT-ND
5	100	R3, R11, R12, R15, R16	RESISTOR	311-100HRCT-ND

Qty	Value	Parts	Description	DIGIKEY PN
1	178	R1	RESISTOR	311-178HRCT-ND
1	287	R2	RESISTOR	311-287HRCT-ND
1	301	R4	RESISTOR	311-301HRCT-ND
2	430	R8, R17	RESISTOR	311-430HRCT-ND
4	3.3k	R19, R20, R21, R22	RESISTOR	311-3.30KHRCT-ND
4	4.7k	R5, R6, R13, R14	RESISTOR	311-4.70KHRCT-ND
2	SPST	SW1, SW2	SMT 6mm switch	P12955SCT-ND
1	STM32F446	U4	NUCLEO-64	497-15882-ND
10		PAD1, PAD2, PAD3, PAD4, PAD5, PAD6, PAD7, PAD8, PAD9, PAD10	Wire Pad	n/a

Qty	Value	Parts	Description	DIGIKEY PN
105		SB1, SB2, SB3, SB4, SB5, SB6, SB7, SB8, SB9, SB10, SB11, SB12, SB13, SB14, SB15, SB16, SB17, SB18, SB19, SB20, SB21, SB22, SB23, SB24, SB25, SB26, SB27, SB28, SB29, SB30, SB31, SB32, SB33, SB34, SB35, SB36, SB37, SB38, SB39, SB40, SB41, SB42, SB43, SB44, SB45, SB46, SB47, SB48, SB49, SB50, SB51, SB52, SB54, SB55, SB56, SB57, SB58, SB59, SB60, SB61, SB62, SB63, SB64, SB65, SB66, SB67, SB68, SB69, SB70, SB71, SB72, SB73, SB75, SB76, SB77, SB78, SB80, SB81, SB82, SB83, SB94, SB96, SB98, SB99, SB100, SB101, SB102, SB103, SB104, SB105, SB106, SB107, SB108, SB109, SB110, SB111, SB112, SB113, SB114, SB115, SB116, SB117, SB118, SB119, SB120	Solder bridge with knife-cuttable pre-bridged connection.	n/a

Appendix E

STM32CUBEMX REPORT

1. Description

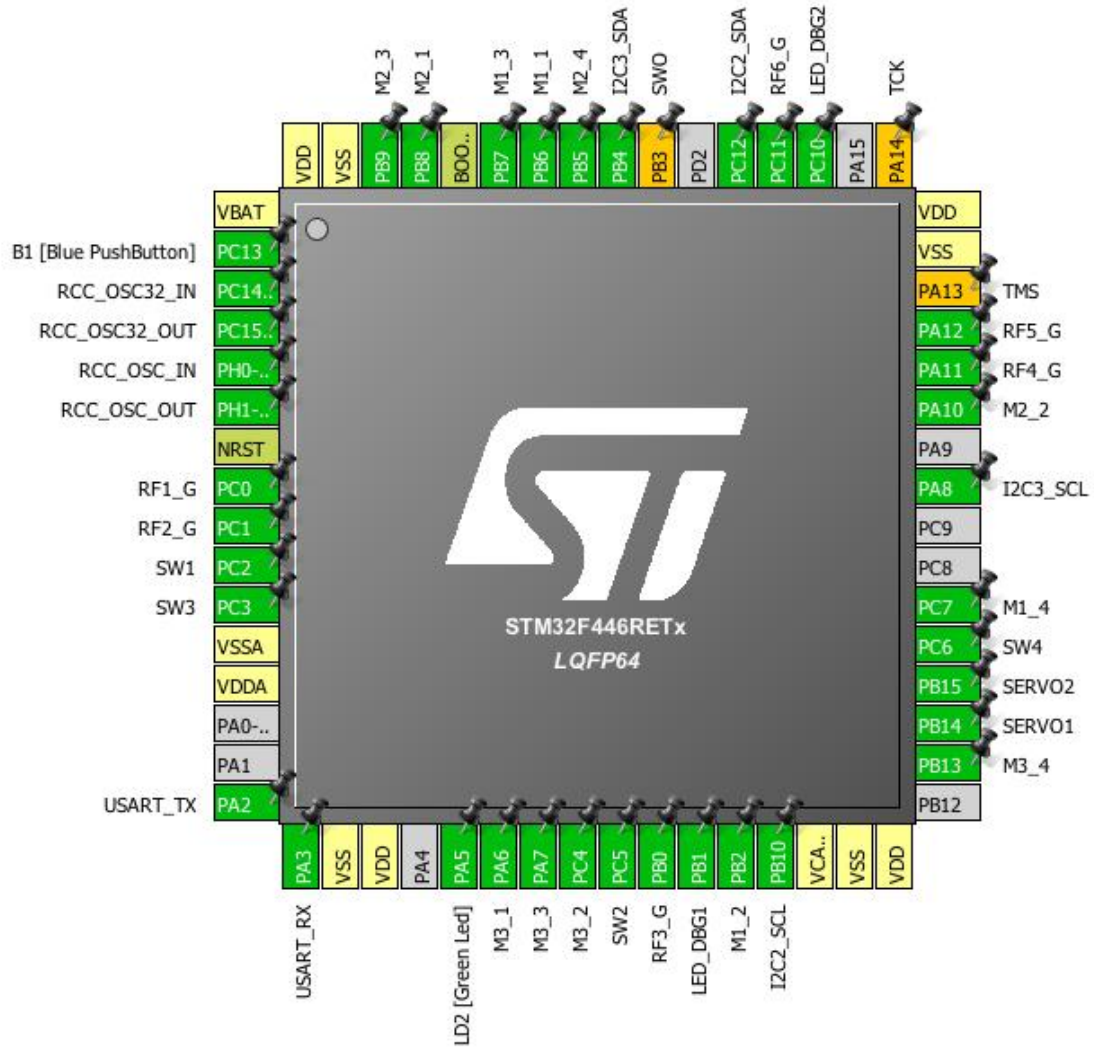
1.1. Project

Project Name	firmware_new
Board Name	NUCLEO-F446RE
Generated with:	STM32CubeMX 4.24.0
Date	05/19/2018

1.2. MCU

MCU Series	STM32F4
MCU Line	STM32F446
MCU name	STM32F446RETx
MCU Package	LQFP64
MCU Pin number	64

2. Pinout Configuration



3. Pins Configuration

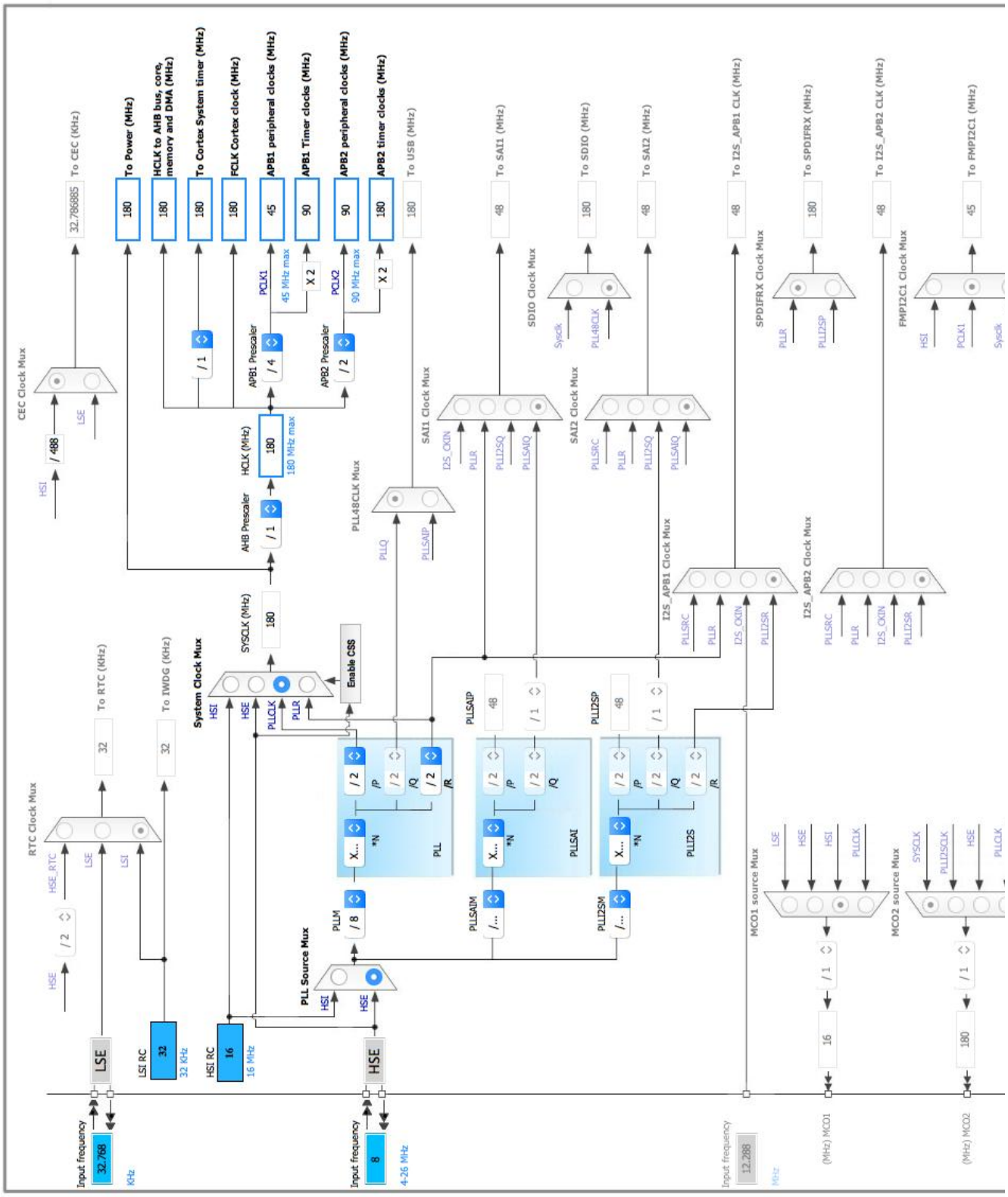
Pin Number LQFP64	Pin Name (function after reset)	Pin Type	Alternate Function(s)	Label
1	VBAT	Power		
2	PC13	I/O	GPIO_EXTI13	B1 [Blue PushButton]
3	PC14-OSC32_IN	I/O	RCC_OSC32_IN	
4	PC15-OSC32_OUT	I/O	RCC_OSC32_OUT	
5	PH0-OSC_IN	I/O	RCC_OSC_IN	
6	PH1-OSC_OUT	I/O	RCC_OSC_OUT	
7	NRST	Reset		
8	PC0	I/O	GPIO_EXTI0	RF1_G
9	PC1	I/O	GPIO_EXTI1	RF2_G
10	PC2 *	I/O	GPIO_Output	SW1
11	PC3 *	I/O	GPIO_Output	SW3
12	VSSA	Power		
13	VDDA	Power		
16	PA2	I/O	USART2_TX	USART_TX
17	PA3	I/O	USART2_RX	USART_RX
18	VSS	Power		
19	VDD	Power		
21	PA5 *	I/O	GPIO_Output	LD2 [Green Led]
22	PA6	I/O	TIM3_CH1	M3_1
23	PA7	I/O	TIM3_CH2	M3_3
24	PC4 *	I/O	GPIO_Output	M3_2
25	PC5 *	I/O	GPIO_Output	SW2
26	PB0 *	I/O	GPIO_Input	RF3_G
27	PB1 *	I/O	GPIO_Output	LED_DBG1
28	PB2 *	I/O	GPIO_Output	M1_2
29	PB10	I/O	I2C2_SCL	
30	VCAP_1	Power		
31	VSS	Power		
32	VDD	Power		
34	PB13 *	I/O	GPIO_Output	M3_4
35	PB14	I/O	TIM12_CH1	SERVO1
36	PB15	I/O	TIM12_CH2	SERVO2
37	PC6 *	I/O	GPIO_Output	SW4
38	PC7 *	I/O	GPIO_Output	M1_4
41	PA8	I/O	I2C3_SCL	
43	PA10 *	I/O	GPIO_Output	M2_2

Pin Number LQFP64	Pin Name (function after reset)	Pin Type	Alternate Function(s)	Label
44	PA11	I/O	GPIO_EXTI11	RF4_G
45	PA12	I/O	GPIO_EXTI12	RF5_G
46	PA13 **	I/O	SYS_JTMS-SWDIO	TMS
47	VSS	Power		
48	VDD	Power		
49	PA14 **	I/O	SYS_JTCK-SWCLK	TCK
51	PC10 *	I/O	GPIO_Output	LED_DBG2
52	PC11 *	I/O	GPIO_Input	RF6_G
53	PC12	I/O	I2C2_SDA	
55	PB3 **	I/O	SYS_JTDO-SWO	SWO
56	PB4	I/O	I2C3_SDA	
57	PB5 *	I/O	GPIO_Output	M2_4
58	PB6	I/O	TIM4_CH1	M1_1
59	PB7	I/O	TIM4_CH2	M1_3
60	BOOT0	Boot		
61	PB8	I/O	TIM4_CH3	M2_1
62	PB9	I/O	TIM4_CH4	M2_3
63	VSS	Power		
64	VDD	Power		

* The pin is affected with an I/O function

** The pin is affected with a peripheral function but no peripheral mode is activated

4. Clock Tree Configuration



5. IPs and Middleware Configuration

5.1. I2C2

I2C: I2C

5.1.1. Parameter Settings:

Master Features:

I2C Speed Mode	**Fast Mode ***
I2C Clock Speed (Hz)	400000
Fast Mode Duty Cycle	Duty cycle Tlow/Thigh = 2

Slave Features:

Clock No Stretch Mode	Disabled
Primary Address Length selection	7-bit
Dual Address Acknowledged	Disabled
Primary slave address	0
General Call address detection	Disabled

5.2. I2C3

I2C: I2C

5.2.1. Parameter Settings:

Master Features:

I2C Speed Mode	**Fast Mode ***
I2C Clock Speed (Hz)	400000
Fast Mode Duty Cycle	Duty cycle Tlow/Thigh = 2

Slave Features:

Clock No Stretch Mode	Disabled
Primary Address Length selection	7-bit
Dual Address Acknowledged	Disabled
Primary slave address	0
General Call address detection	Disabled

5.3. RCC

High Speed Clock (HSE): Crystal/Ceramic Resonator
Low Speed Clock (LSE) : Crystal/Ceramic Resonator

5.3.1. Parameter Settings:

System Parameters:	
VDD voltage (V)	3.3
Instruction Cache	Enabled
Prefetch Buffer	Enabled
Data Cache	Enabled
Flash Latency(WS)	5 WS (6 CPU cycle)
RCC Parameters:	
HSI Calibration Value	16
TIM Prescaler Selection	Disabled
HSE Startup Timout Value (ms)	100
LSE Startup Timout Value (ms)	5000
Power Parameters:	
Power Regulatror Voltage Scale	Power Regulator Voltage Scale 1
Power Over Drive	Enabled

5.4. SYS

Timebase Source: SysTick

5.5. TIM3

Clock Source : Internal Clock
Channel1: PWM Generation CH1
Channel2: PWM Generation CH2

5.5.1. Parameter Settings:

Counter Settings:	
Prescaler (PSC - 16 bits value)	**1 ***
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits value)	**2047 ***
Internal Clock Division (CKD)	No Division

Trigger Output (TRGO) Parameters:

Master/Slave Mode	Disable (no sync between this TIM (Master) and its Slaves
Trigger Event Selection	Reset (UG bit from TIMx_EGR)

PWM Generation Channel 1:

Mode	PWM mode 1
Pulse (16 bits value)	0
Fast Mode	Disable
CH Polarity	High

PWM Generation Channel 2:

Mode	PWM mode 1
Pulse (16 bits value)	0
Fast Mode	Disable
CH Polarity	High

5.6. TIM4

mode: Clock Source
Channel1: PWM Generation CH1
Channel2: PWM Generation CH2
Channel3: PWM Generation CH3
Channel4: PWM Generation CH4

5.6.1. Parameter Settings:

Counter Settings:

Prescaler (PSC - 16 bits value)	**1 ***
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits value)	**2047 ***
Internal Clock Division (CKD)	No Division

Trigger Output (TRGO) Parameters:

Master/Slave Mode	Disable (no sync between this TIM (Master) and its Slaves
Trigger Event Selection	Reset (UG bit from TIMx_EGR)

PWM Generation Channel 1:

Mode	PWM mode 1
Pulse (16 bits value)	0
Fast Mode	Disable
CH Polarity	High

PWM Generation Channel 2:

Mode	PWM mode 1

Pulse (16 bits value)	0
Fast Mode	Disable
CH Polarity	High

PWM Generation Channel 3:

Mode	PWM mode 1
Pulse (16 bits value)	0
Fast Mode	Disable
CH Polarity	High

PWM Generation Channel 4:

Mode	PWM mode 1
Pulse (16 bits value)	0
Fast Mode	Disable
CH Polarity	High

5.7. TIM5

mode: Clock Source

5.7.1. Parameter Settings:

Counter Settings:

Prescaler (PSC - 16 bits value)	**9000 ***
Counter Mode	Up
Counter Period (AutoReload Register - 32 bits value)	**0xFFFFFFFF ***
Internal Clock Division (CKD)	No Division

Trigger Output (TRGO) Parameters:

Master/Slave Mode	Disable (no sync between this TIM (Master) and its Slaves
Trigger Event Selection	Reset (UG bit from TIMx_EGR)

5.8. TIM12

mode: Clock Source
Channel1: PWM Generation CH1
Channel2: PWM Generation CH2

5.8.1. Parameter Settings:

Counter Settings:

Prescaler (PSC - 16 bits value)	**354 ***
Counter Mode	Up
Counter Period (AutoReload Register - 16 bits value)	**3178 ***
Internal Clock Division (CKD)	No Division

PWM Generation Channel 1:

Mode	PWM mode 1
Pulse (16 bits value)	**63 ***
Fast Mode	Disable
CH Polarity	High

PWM Generation Channel 2:

Mode	PWM mode 1
Pulse (16 bits value)	**63 ***
Fast Mode	Disable
CH Polarity	High

5.9. USART2

Mode: Asynchronous

5.9.1. Parameter Settings:

Basic Parameters:

Baud Rate	**921600 ***
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1

Advanced Parameters:

Data Direction	Receive and Transmit
Over Sampling	16 Samples

*** User modified value**

6. System Configuration

6.1. GPIO configuration

IP	Pin	Signal	GPIO mode	GPIO pull/up pull down	Max Speed	User Label
I2C2	PB10	I2C2_SCL	Alternate Function Open Drain	Pull-up	**Very High** *	
	PC12	I2C2_SDA	Alternate Function Open Drain	Pull-up	**Very High** *	
I2C3	PA8	I2C3_SCL	Alternate Function Open Drain	Pull-up	**Very High** *	
	PB4	I2C3_SDA	Alternate Function Open Drain	Pull-up	**Very High** *	
RCC	PC14-OSC32_IN	RCC_OSC32_IN	n/a	n/a	n/a	
	PC15-OSC32_OUT	RCC_OSC32_OUT	n/a	n/a	n/a	
	PH0-OSC_IN	RCC_OSC_IN	n/a	n/a	n/a	
	PH1-OSC_OUT	RCC_OSC_OUT	n/a	n/a	n/a	
TIM3	PA6	TIM3_CH1	Alternate Function Push Pull	No pull-up and no pull-down	Low	M3_1
	PA7	TIM3_CH2	Alternate Function Push Pull	No pull-up and no pull-down	Low	M3_3
TIM4	PB6	TIM4_CH1	Alternate Function Push Pull	No pull-up and no pull-down	Low	M1_1
	PB7	TIM4_CH2	Alternate Function Push Pull	No pull-up and no pull-down	Low	M1_3
	PB8	TIM4_CH3	Alternate Function Push Pull	No pull-up and no pull-down	Low	M2_1
	PB9	TIM4_CH4	Alternate Function Push Pull	No pull-up and no pull-down	Low	M2_3
TIM12	PB14	TIM12_CH1	Alternate Function Push Pull	No pull-up and no pull-down	Low	SERVO1
	PB15	TIM12_CH2	Alternate Function Push Pull	No pull-up and no pull-down	Low	SERVO2
USART2	PA2	USART2_TX	Alternate Function Push Pull	Pull-up	**Very High** *	USART_TX
	PA3	USART2_RX	Alternate Function Push Pull	Pull-up	**Very High** *	USART_RX
Single Mapped Signals	PA13	SYS_JTMS-SWDIO	n/a	n/a	n/a	TMS
	PA14	SYS_JTCK-SWCLK	n/a	n/a	n/a	TCK
	PB3	SYS_JTDO-	n/a	n/a	n/a	SWO

IP	Pin	Signal	GPIO mode	GPIO pull/up pull down	Max Speed	User Label
		SWO				
GPIO	PC13	GPIO_EXTI13	**External Interrupt Mode with Falling edge trigger detection**	No pull-up and no pull-down	n/a	B1 [Blue PushButton]
	PC0	GPIO_EXTI0	**External Interrupt Mode with Falling edge trigger detection**	**Pull-up ***	n/a	RF1_G
	PC1	GPIO_EXTI1	**External Interrupt Mode with Falling edge trigger detection**	**Pull-up ***	n/a	RF2_G
	PC2	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	SW1
	PC3	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	SW3
	PA5	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	LD2 [Green Led]
	PC4	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	M3_2
	PC5	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	SW2
	PB0	GPIO_Input	Input mode	**Pull-up ***	n/a	RF3_G
	PB1	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	LED_DBG1
	PB2	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	M1_2
	PB13	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	M3_4
	PC6	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	SW4
	PC7	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	M1_4
	PA10	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	M2_2
	PA11	GPIO_EXTI11	**External Interrupt Mode with Falling edge trigger detection**	**Pull-up ***	n/a	RF4_G
	PA12	GPIO_EXTI12	**External Interrupt Mode with Falling edge trigger detection**	**Pull-up ***	n/a	RF5_G
	PC10	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	LED_DBG2
	PC11	GPIO_Input	Input mode	No pull-up and no pull-down	n/a	RF6_G
	PB5	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	M2_4

6.2. DMA configuration

DMA request	Stream	Direction	Priority
I2C2_RX	DMA1_Stream2	Peripheral To Memory	Low
I2C2_TX	DMA1_Stream7	Memory To Peripheral	**High ***
I2C3_RX	DMA1_Stream1	Peripheral To Memory	Low
I2C3_TX	DMA1_Stream4	Memory To Peripheral	**High ***
USART2_RX	DMA1_Stream5	Peripheral To Memory	**Very High ***
USART2_TX	DMA1_Stream6	Memory To Peripheral	**Very High ***

I2C2_RX: DMA1_Stream2 DMA request Settings:

Mode: Normal
Use fifo: Disable
Peripheral Increment: Disable
Memory Increment: **Enable ***
Peripheral Data Width: Byte
Memory Data Width: Byte

I2C2_TX: DMA1_Stream7 DMA request Settings:

Mode: Normal
Use fifo: Disable
Peripheral Increment: Disable
Memory Increment: **Enable ***
Peripheral Data Width: Byte
Memory Data Width: Byte

I2C3_RX: DMA1_Stream1 DMA request Settings:

Mode: Normal
Use fifo: Disable
Peripheral Increment: Disable
Memory Increment: **Enable ***
Peripheral Data Width: Byte
Memory Data Width: Byte

I2C3_TX: DMA1_Stream4 DMA request Settings:

Mode:	Normal
Use fifo:	Disable
Peripheral Increment:	Disable
Memory Increment:	**Enable ***
Peripheral Data Width:	Byte
Memory Data Width:	Byte

USART2_RX: DMA1_Stream5 DMA request Settings:

Mode:	**Circular ***
Use fifo:	Disable
Peripheral Increment:	Disable
Memory Increment:	**Enable ***
Peripheral Data Width:	Byte
Memory Data Width:	Byte

USART2_TX: DMA1_Stream6 DMA request Settings:

Mode:	Normal
Use fifo:	Disable
Peripheral Increment:	Disable
Memory Increment:	**Enable ***
Peripheral Data Width:	Byte
Memory Data Width:	Byte

6.3. NVIC configuration

Interrupt Table	Enable	Preenmption Priority	SubPriority
Non maskable interrupt	true	0	0
Hard fault interrupt	true	0	0
Memory management fault	true	0	0
Pre-fetch fault, memory access fault	true	0	0
Undefined instruction or illegal state	true	0	0
System service call via SWI instruction	true	0	0
Debug monitor	true	0	0
Pendable request for system service	true	0	0
System tick timer	true	0	0
DMA1 stream1 global interrupt	true	0	0
DMA1 stream2 global interrupt	true	0	0
DMA1 stream4 global interrupt	true	0	0
DMA1 stream5 global interrupt	true	0	0
DMA1 stream6 global interrupt	true	0	0
I2C2 event interrupt	true	0	0
I2C2 error interrupt	true	0	0
USART2 global interrupt	true	0	0
DMA1 stream7 global interrupt	true	0	0
I2C3 event interrupt	true	0	0
I2C3 error interrupt	true	0	0
PVD interrupt through EXTI line 16	unused		
Flash global interrupt	unused		
RCC global interrupt	unused		
EXTI line 0 interrupt	unused		
EXTI line 1 interrupt	unused		
TIM3 global interrupt	unused		
TIM4 global interrupt	unused		
EXTI line[15:10] interrupts	unused		
TIM8 break interrupt and TIM12 global interrupt	unused		
TIM5 global interrupt	unused		
FPU global interrupt	unused		

*** User modified value**

7. Power Consumption Calculator report

7.1. Microcontroller Selection

Series	STM32F4
Line	STM32F446
MCU	STM32F446RETx
Datasheet	027107_Rev6

7.2. Parameter Selection

Temperature	25
Vdd	3.3

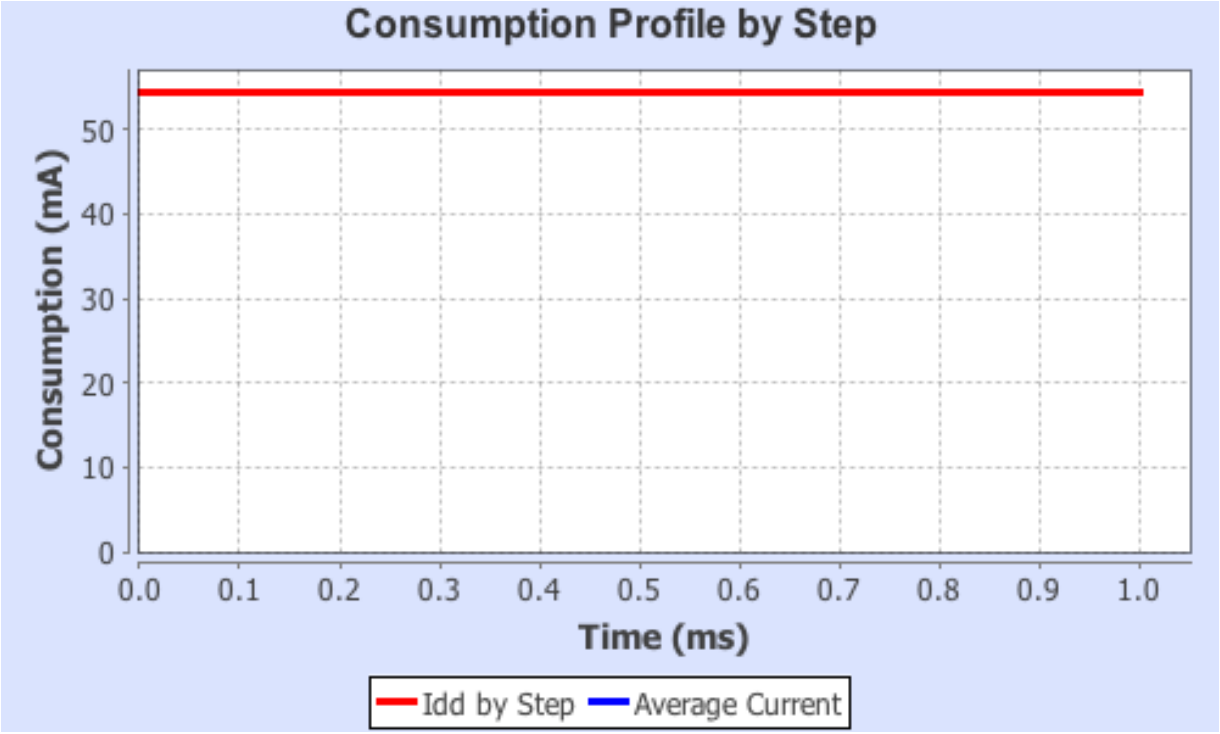
7.3. Sequence

Step	Step1
Mode	RUN
Vdd	3.3
Voltage Source	Vbus
Range	Scale1-High
Fetch Type	RAM/FLASH/REGON/ART/PREFETCH
Clock Configuration	HSE PLL
Clock Source Frequency	4 MHz
CPU Frequency	180 MHz
Peripherals	DMA1 DMA2 GPIOA GPIOB GPIOC I2C2 I2C3 SYS TIM1 TIM2 TIM3 TIM4 TIM12 USART2
Additional Cons.	0 mA
Average Current	54.31 mA
Duration	1 ms
DMIPS	225.0
Ta Max	96.76
Category	In DS Table

7.4. RESULTS

Sequence Time	1 ms	Average Current	54.31 mA
Battery Life	0	Average DMIPS	225.0 DMIPS

7.5. Chart



8. Software Project

8.1. Project Settings

Name	Value
Project Name	firmware_new
Project Folder	/Users/justinng/Documents/Github/roborodentia2017/firmware_new
Toolchain / IDE	Makefile
Firmware Package Name and Version	STM32Cube FW_F4 V1.18.0

8.2. Code Generation Settings

Name	Value
STM32Cube Firmware Library Package	Copy all used libraries into the project folder
Generate peripheral initialization as a pair of '.c/.h' files	Yes
Backup previously generated files when re-generating	No
Delete previously generated files when not re-generated	Yes
Set all free pins as analog (to optimize the power consumption)	No

9. Software Pack Report

Appendix F

X TRANSLATION NETWORK PERFORMANCE

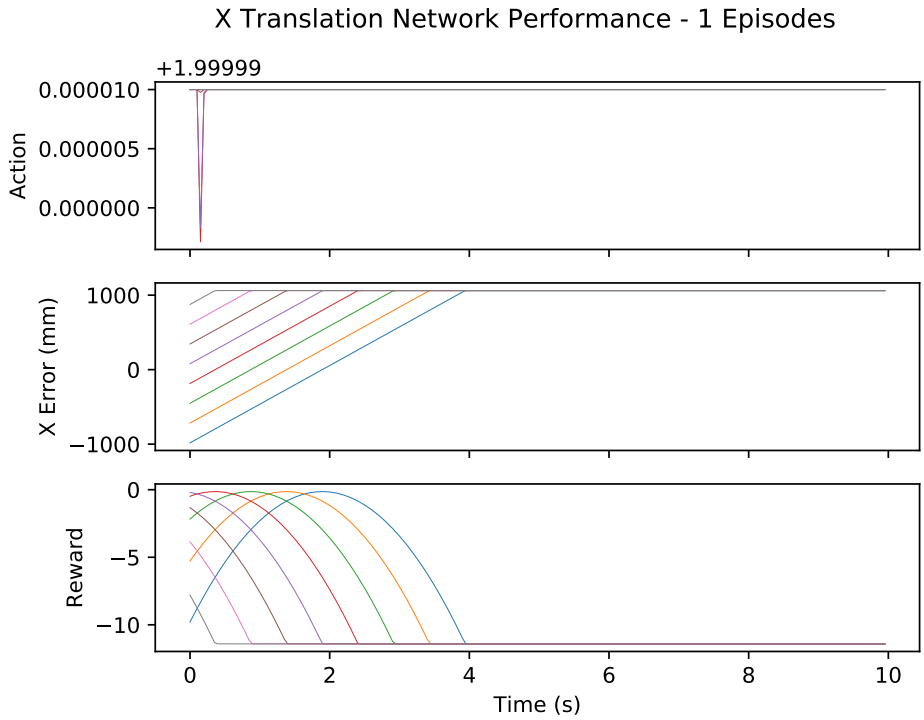


Figure F.1: X Translation Network Performance – 1 Episode

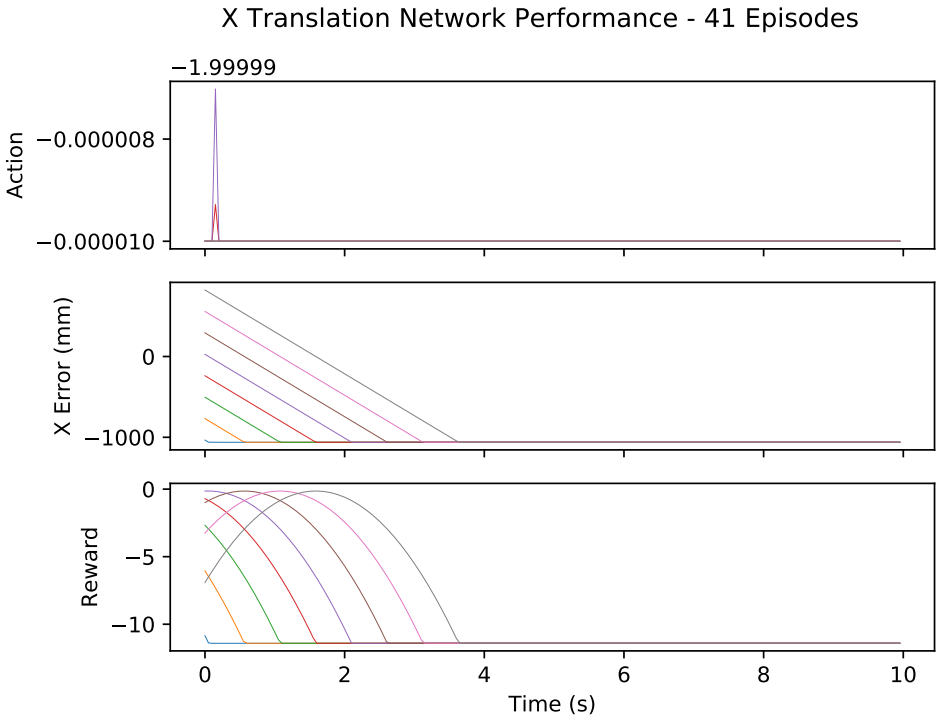


Figure F.2: X Translation Network Performance – 41 Episodes

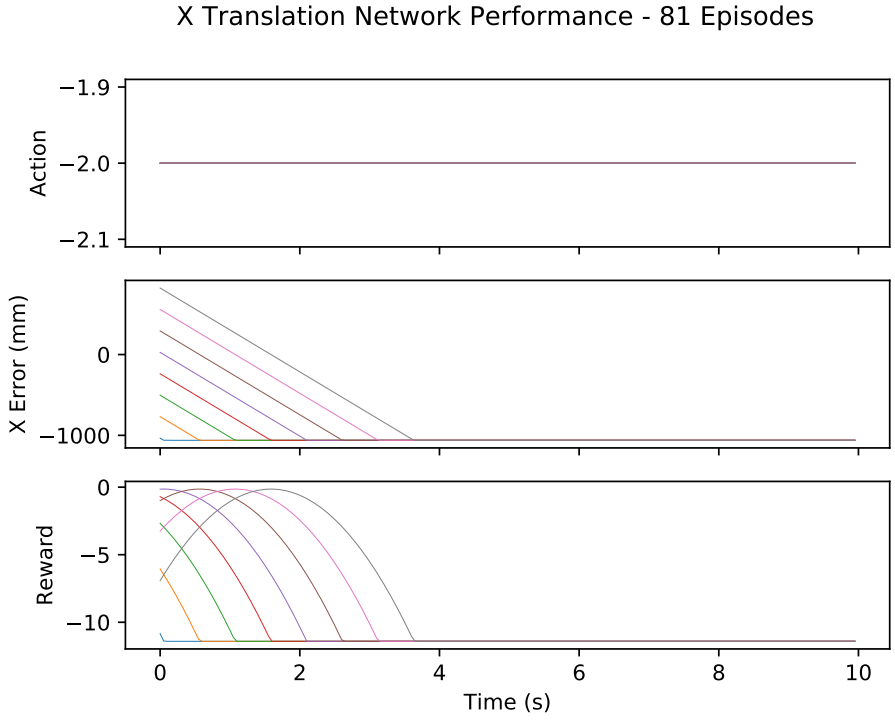


Figure F.3: X Translation Network Performance – 81 Episodes

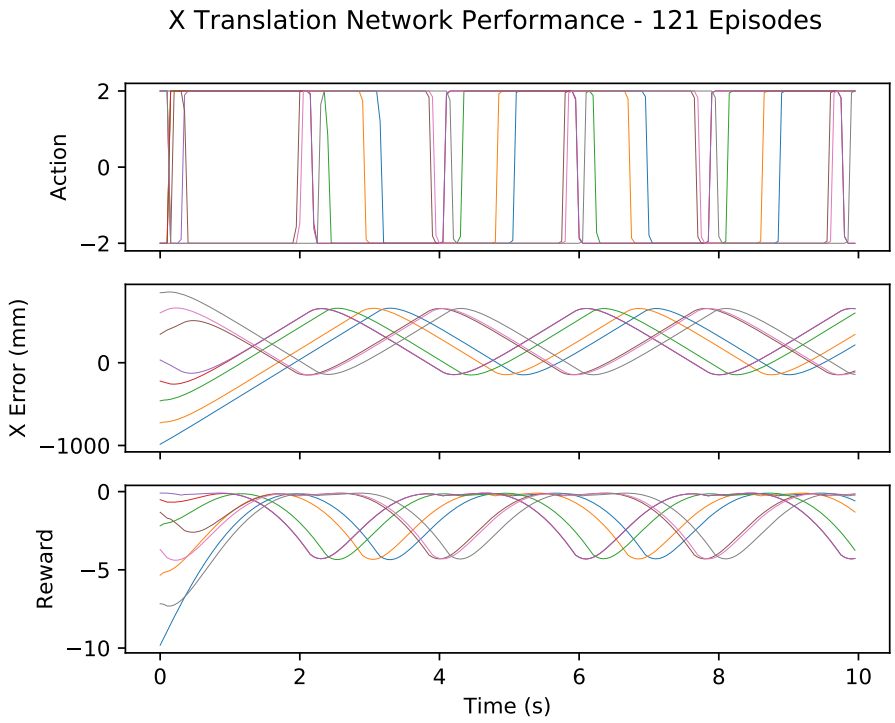


Figure F.4: X Translation Network Performance – 121 Episodes

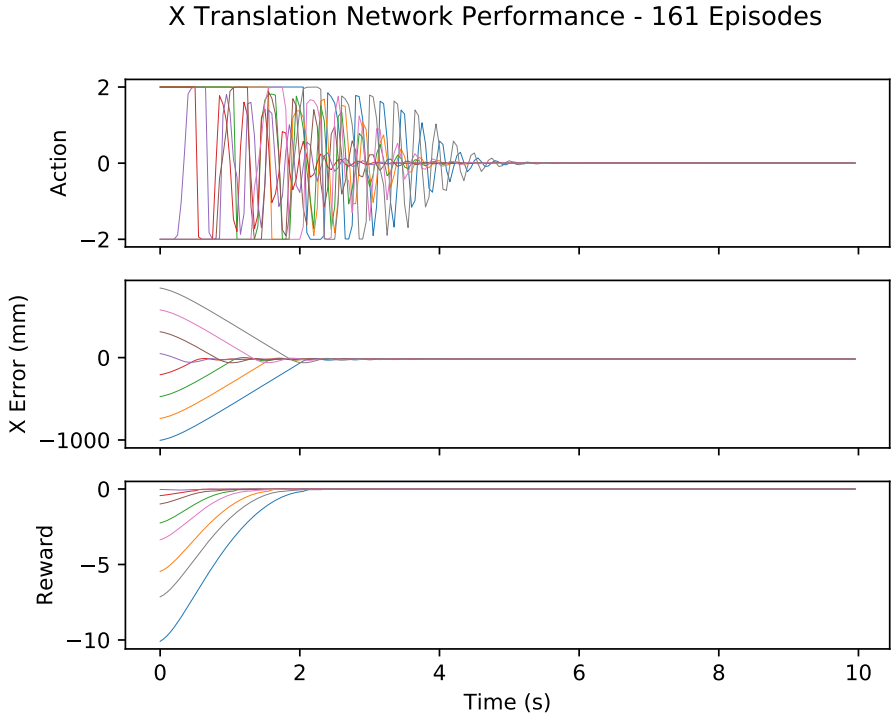


Figure F.5: X Translation Network Performance – 161 Episodes

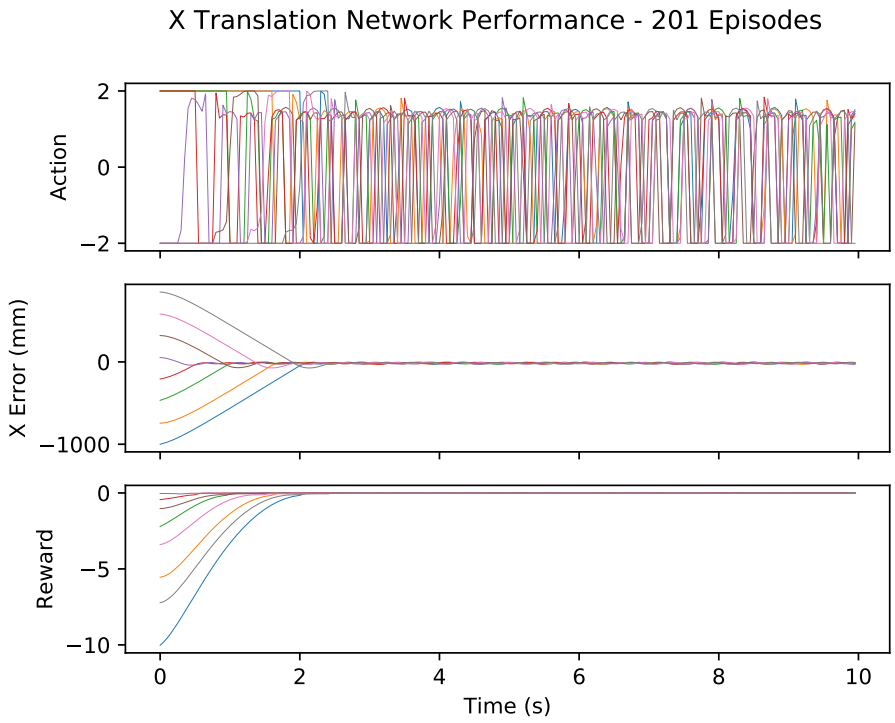


Figure F.6: X Translation Network Performance – 201 Episodes

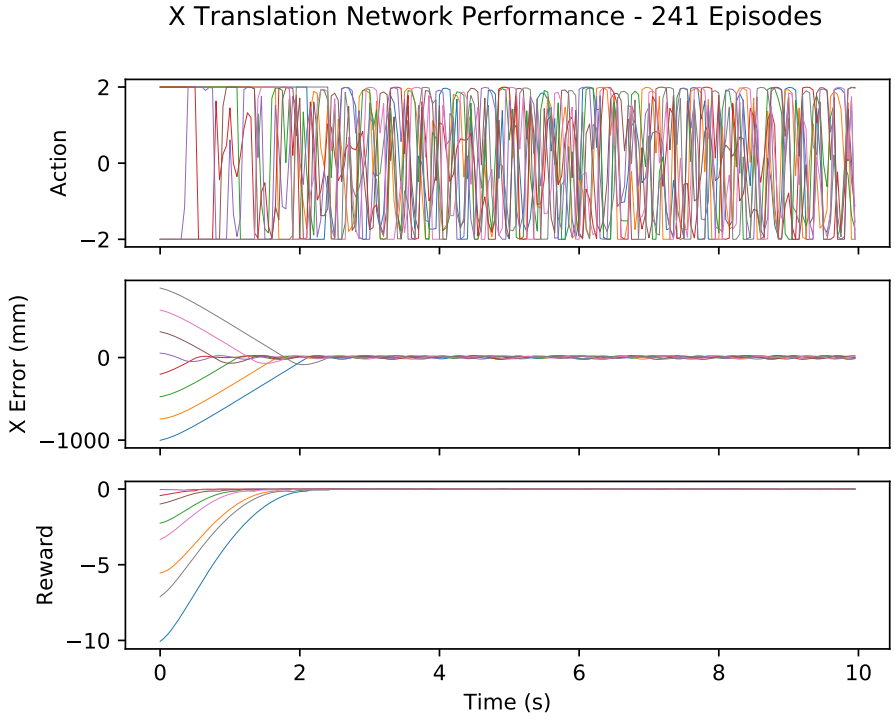


Figure F.7: X Translation Network Performance – 241 Episodes

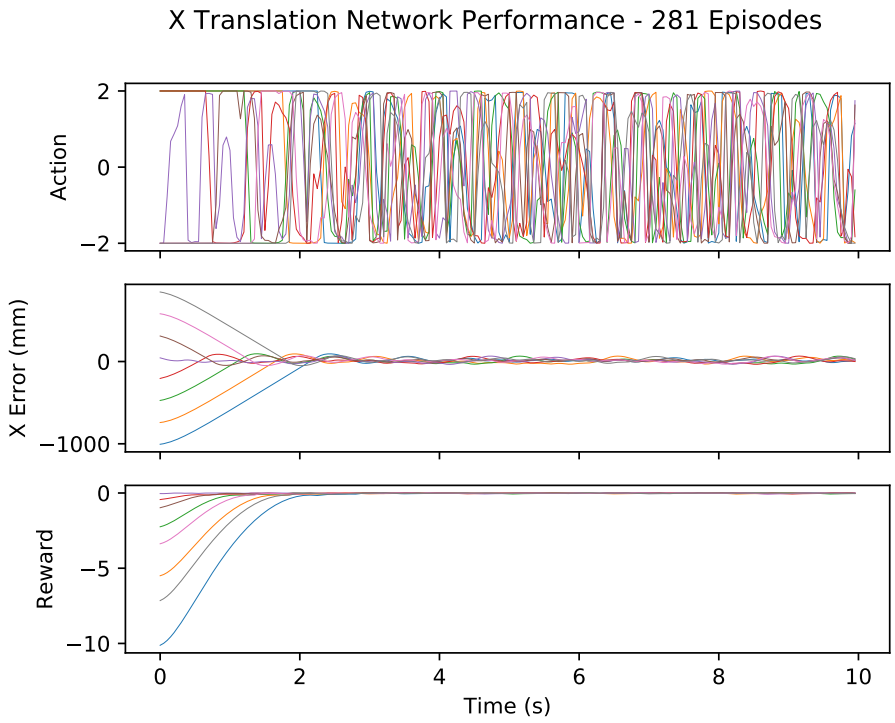


Figure F.8: X Translation Network Performance – 281 Episodes

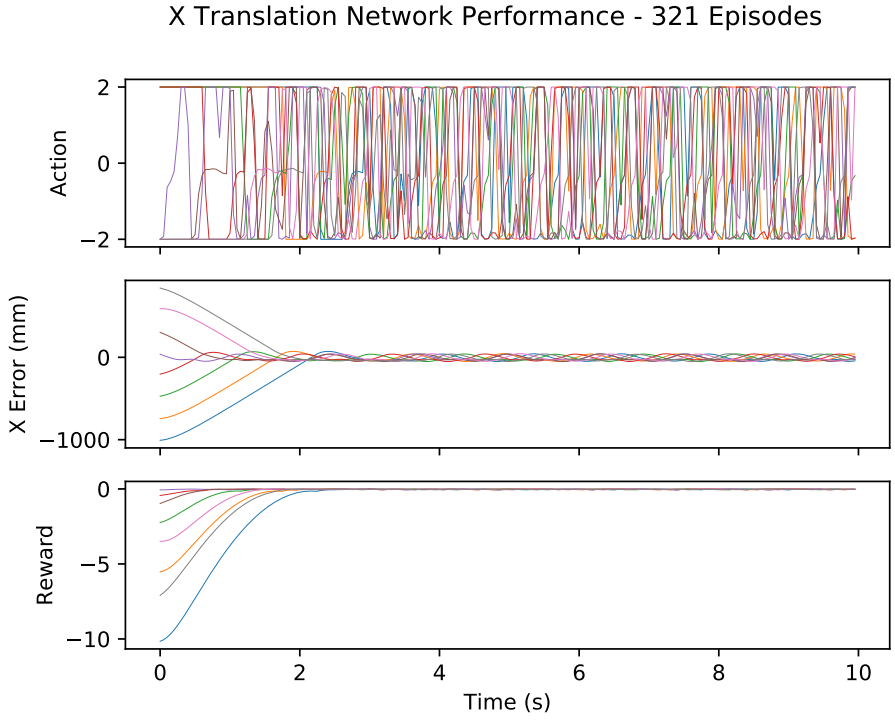


Figure F.9: X Translation Network Performance – 321 Episodes

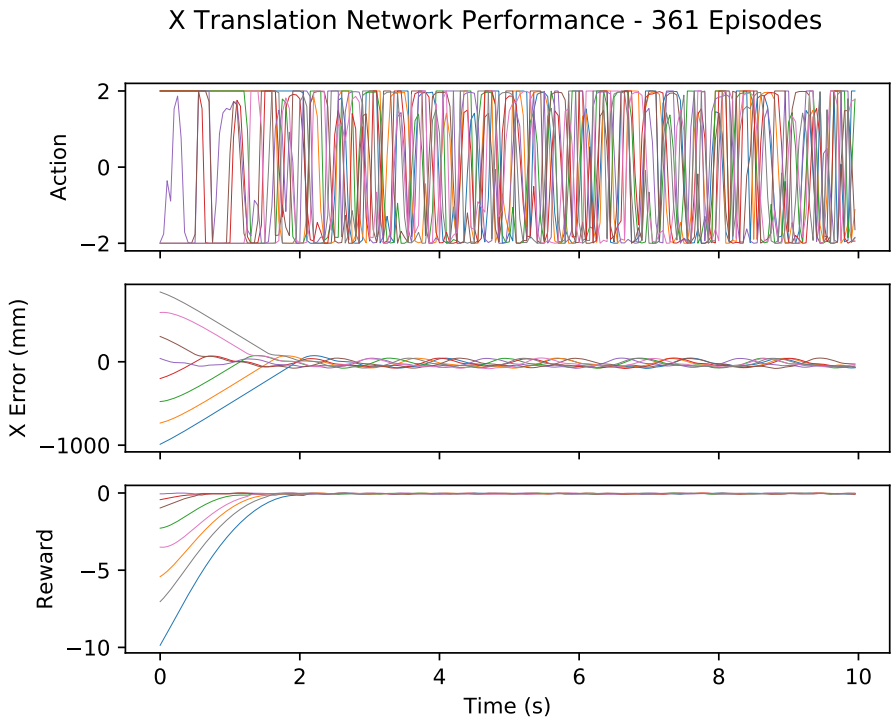


Figure F.10: X Translation Network Performance – 361 Episodes

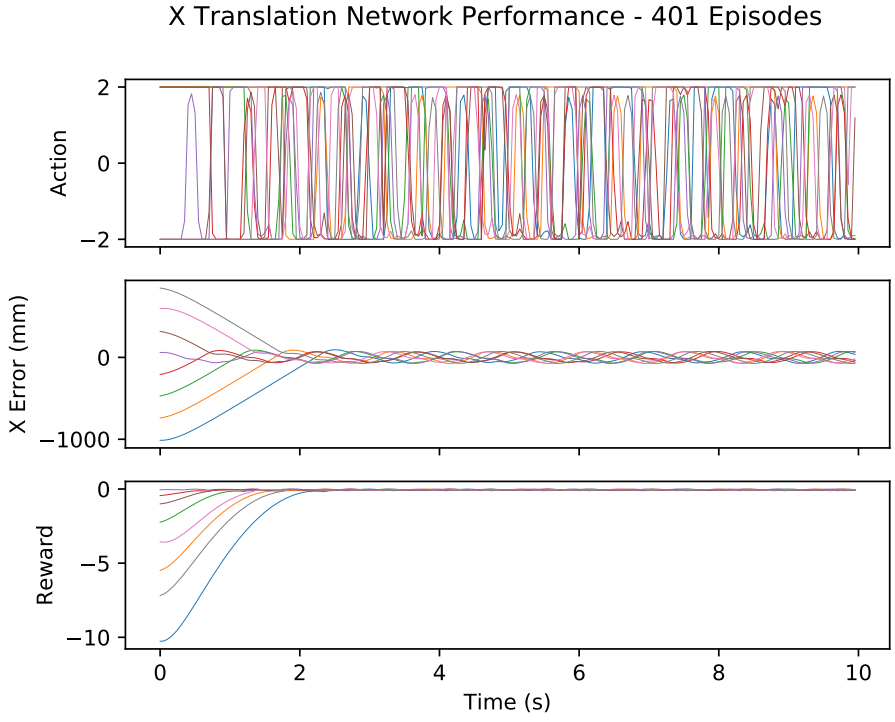


Figure F.11: X Translation Network Performance – 401 Episodes

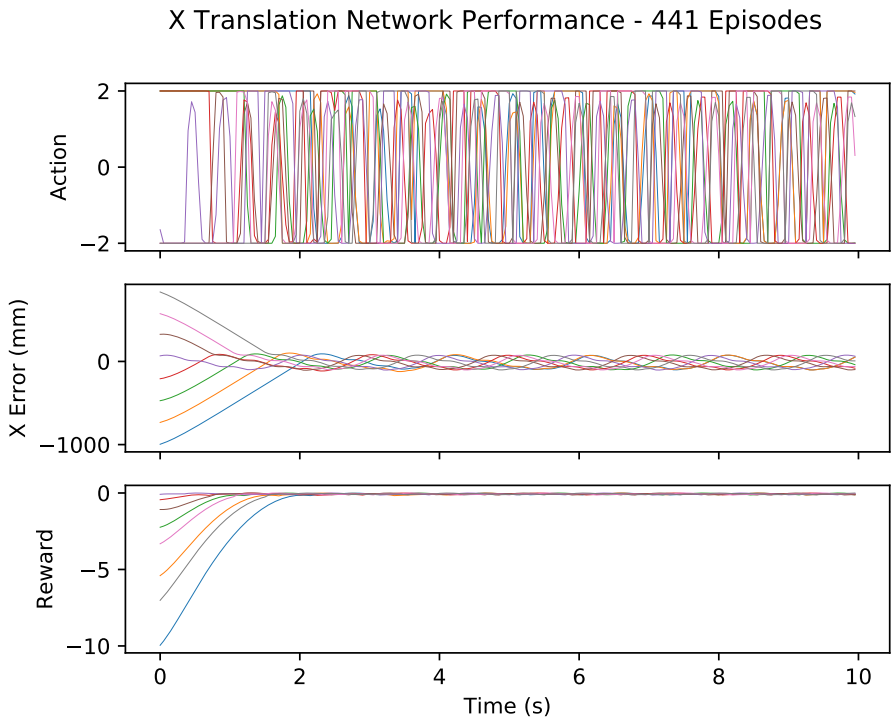


Figure F.12: X Translation Network Performance – 441 Episodes

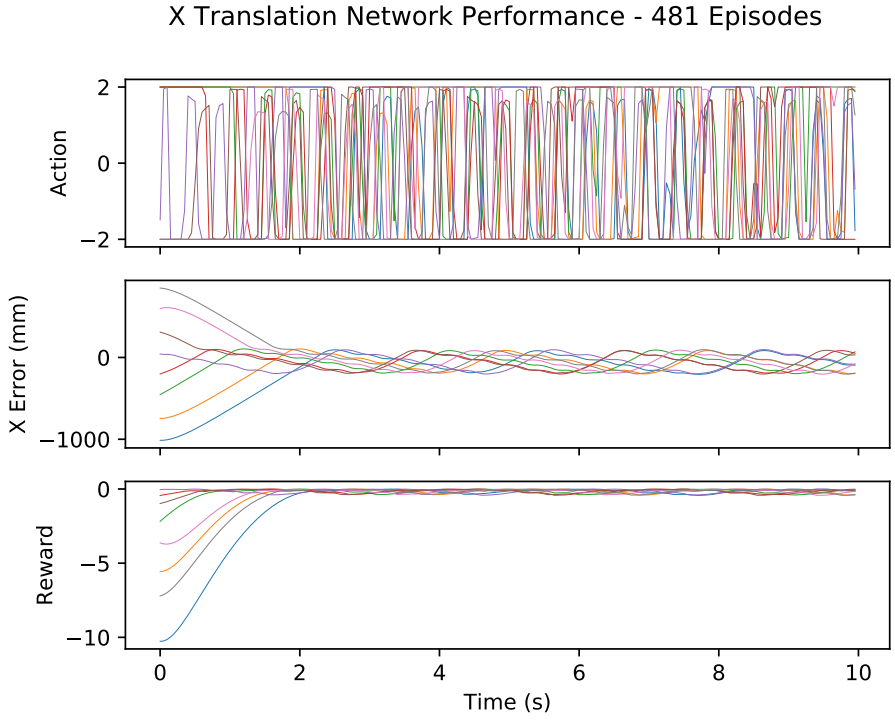


Figure F.13: X Translation Network Performance – 481 Episodes

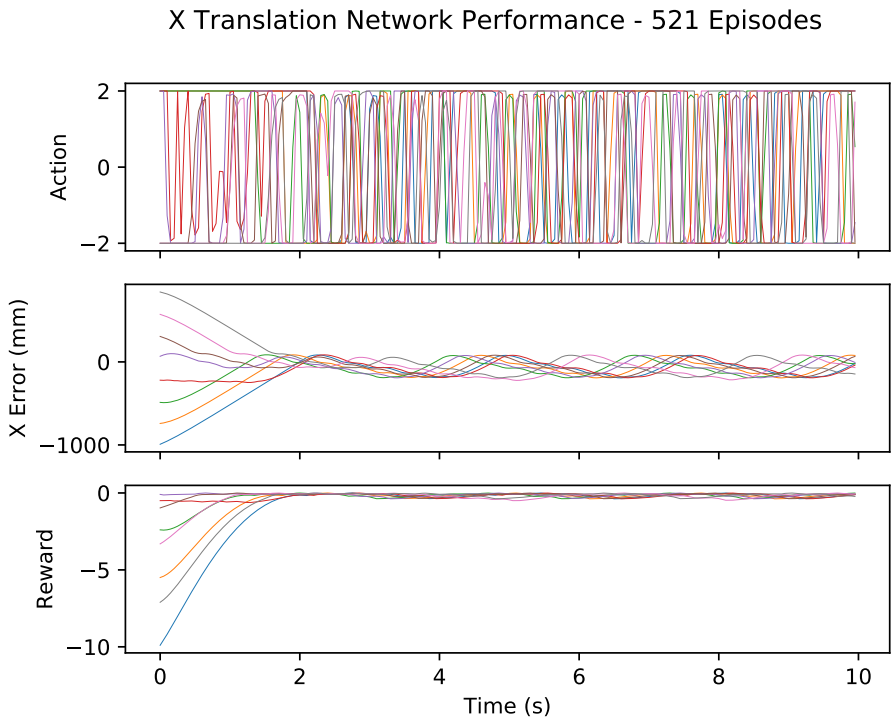


Figure F.14: X Translation Network Performance – 521 Episodes

Appendix G

Y TRANSLATION NETWORK PERFORMANCE

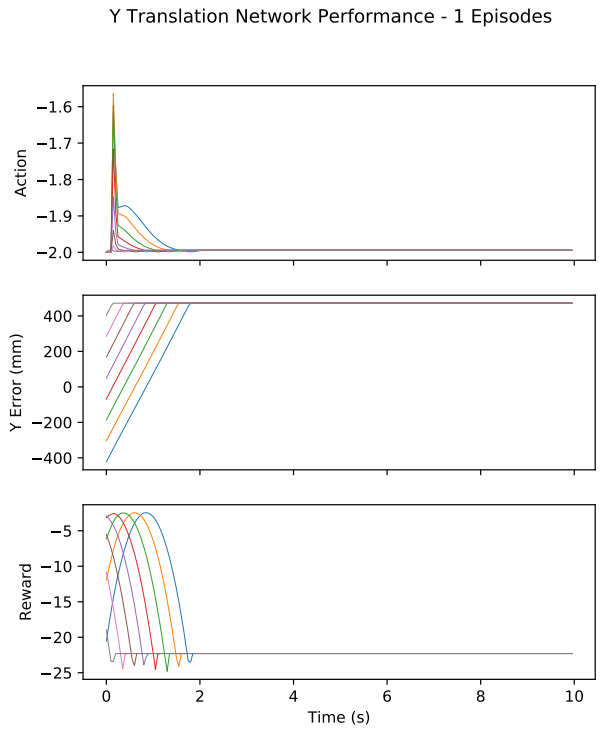


Figure G.1: Y Translation Network Performance – 1 Episode

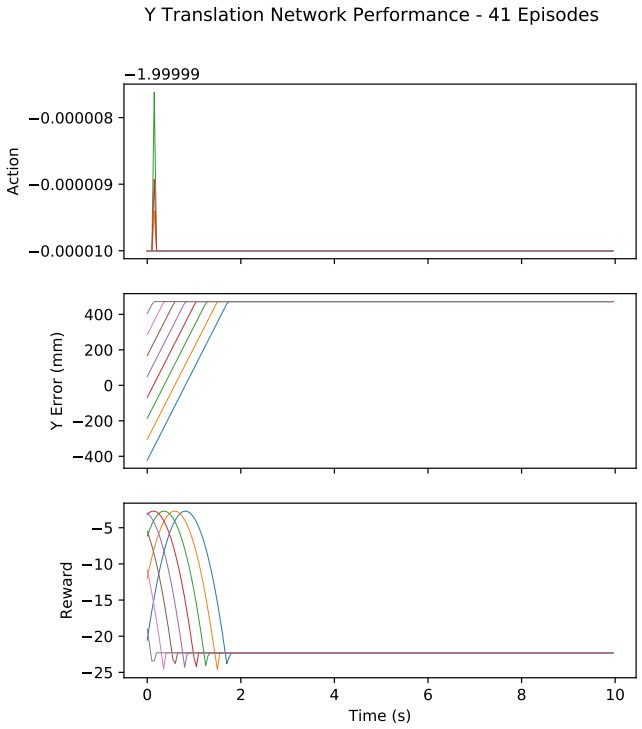


Figure G.2: Y Translation Network Performance – 41 Episodes

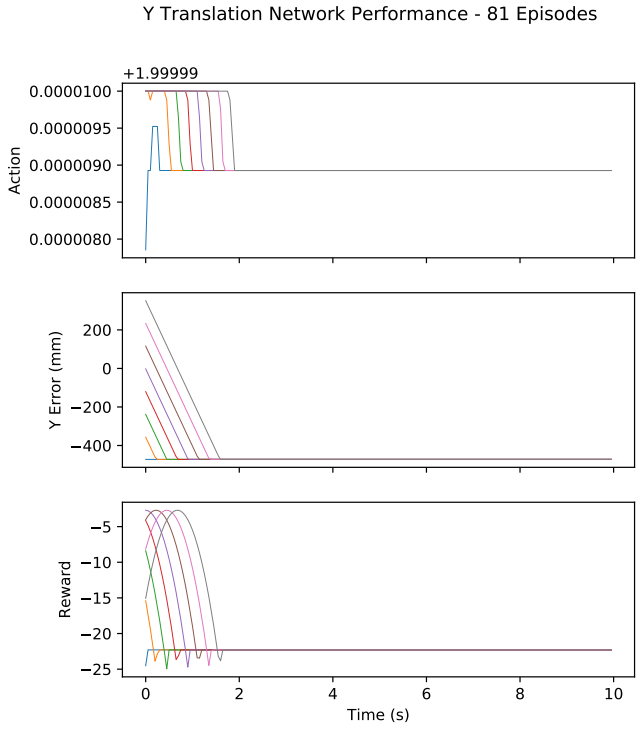


Figure G.3: Y Translation Network Performance – 81 Episodes

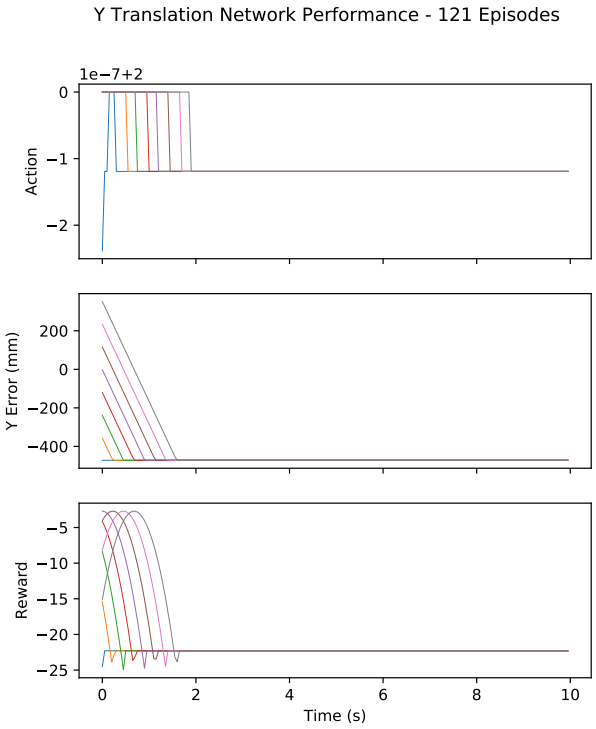


Figure G.4: Y Translation Network Performance – 121 Episodes

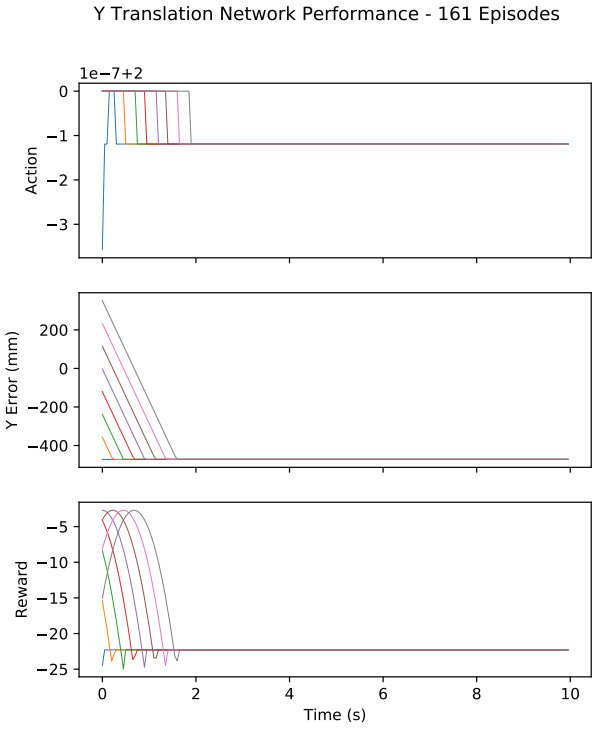


Figure G.5: Y Translation Network Performance – 161 Episodes

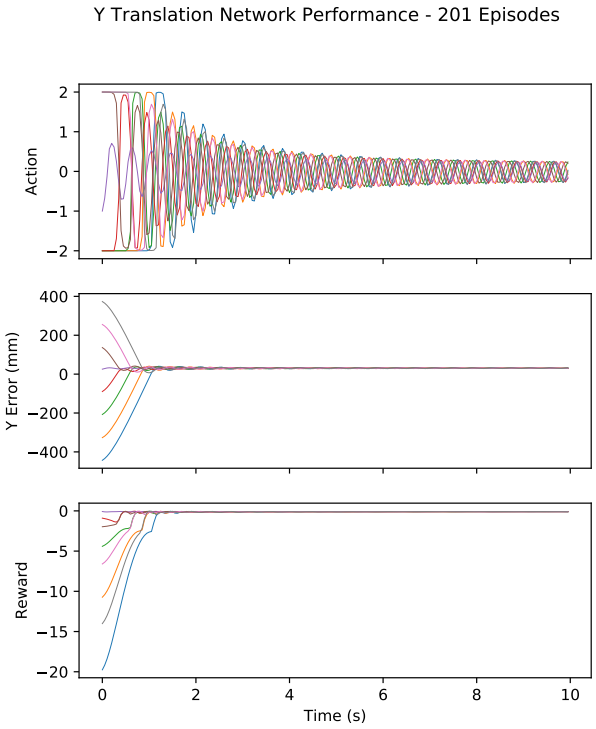


Figure G.6: Y Translation Network Performance – 201 Episodes

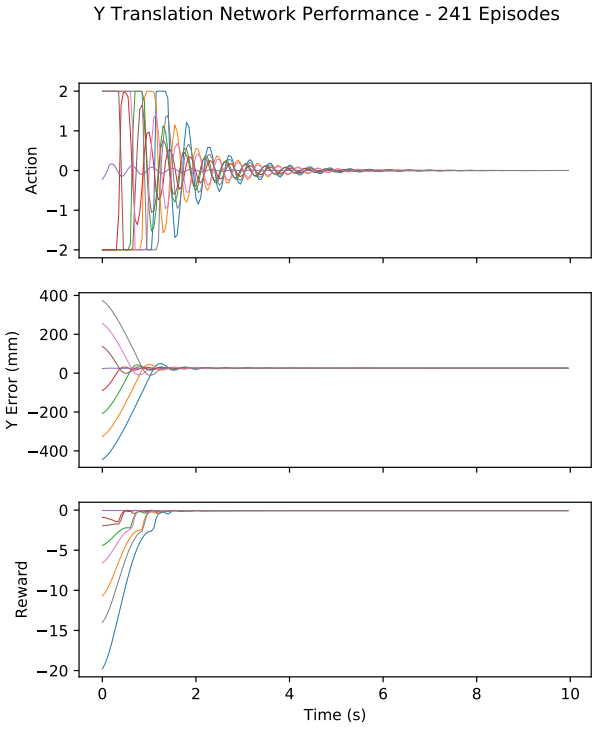


Figure G.7: Y Translation Network Performance – 241 Episodes

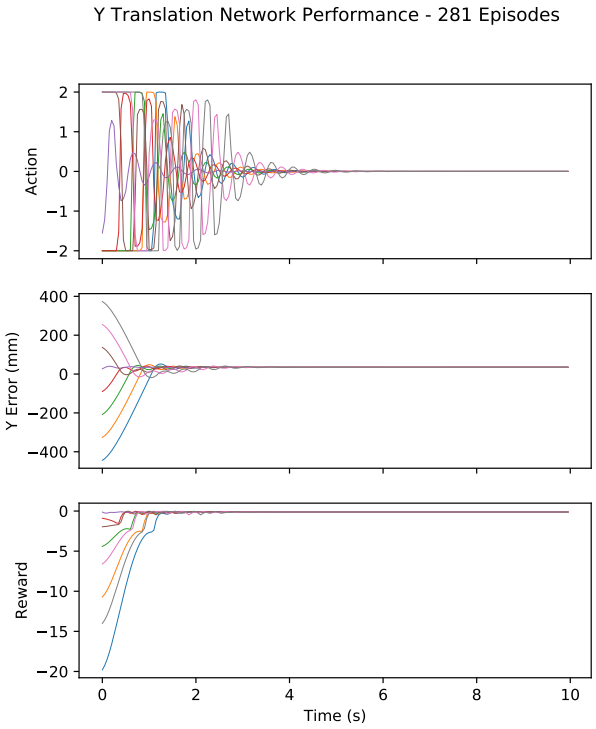


Figure G.8: Y Translation Network Performance – 281 Episodes

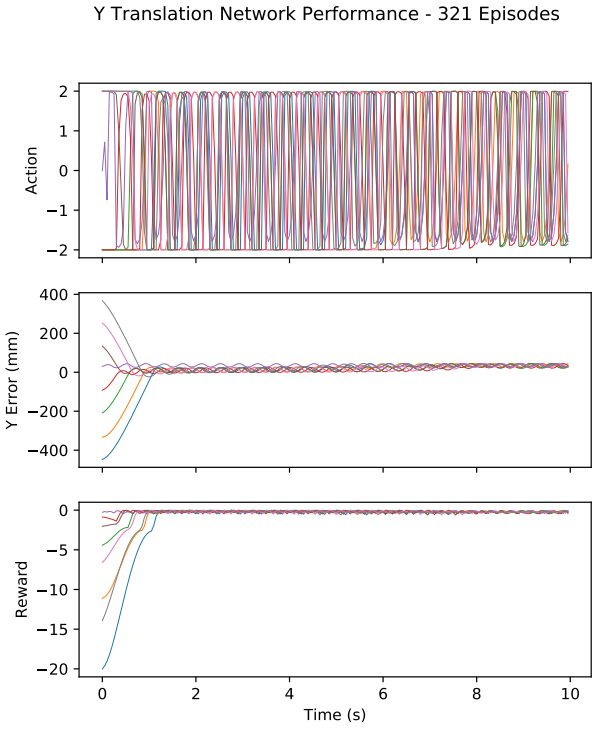


Figure G.9: Y Translation Network Performance – 321 Episodes

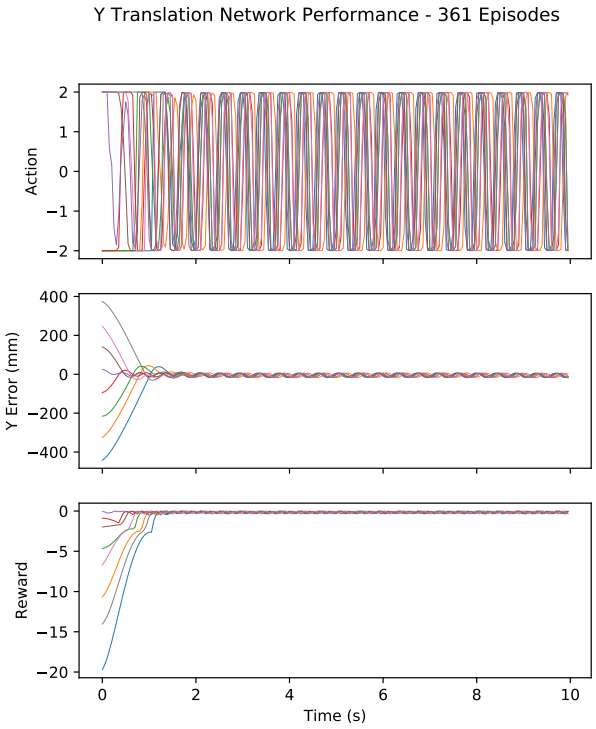


Figure G.10: Y Translation Network Performance – 361 Episodes

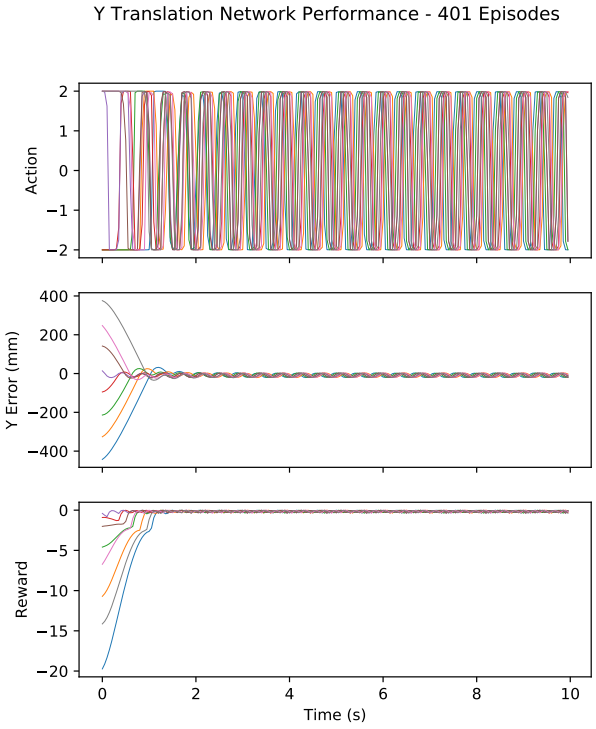


Figure G.11: Y Translation Network Performance – 401 Episodes

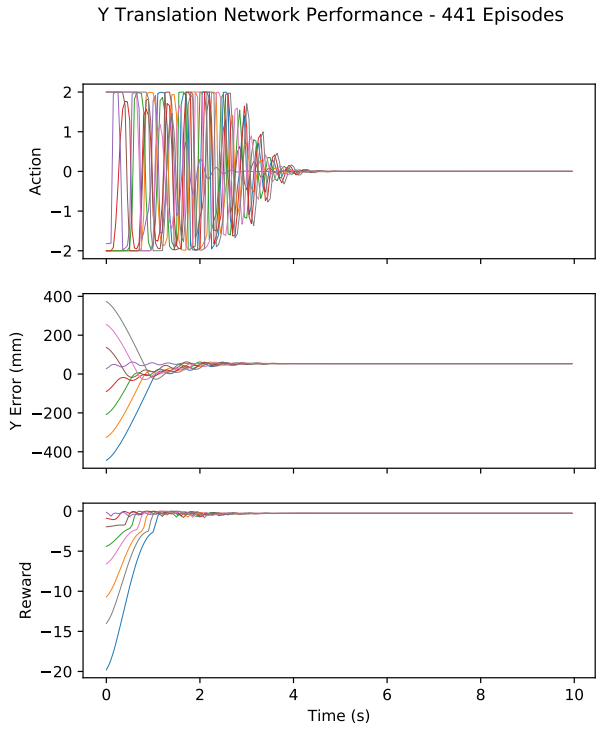


Figure G.12: Y Translation Network Performance – 441 Episodes

Appendix H

ANGLE NETWORK PERFORMANCE

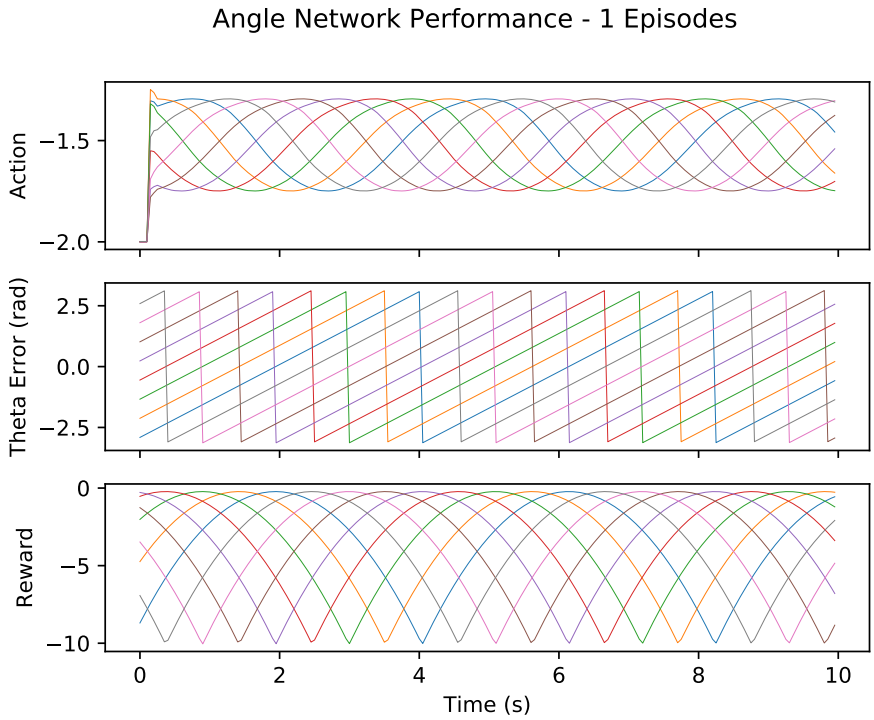


Figure H.1: Angle Network Performance – 1 Episode

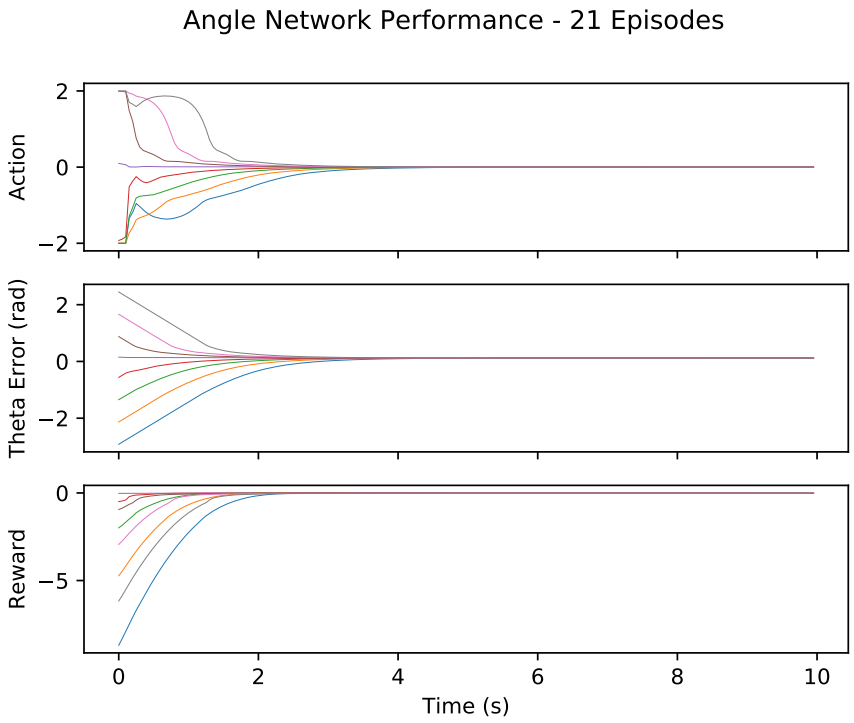


Figure H.2: Angle Network Performance – 21 Episodes

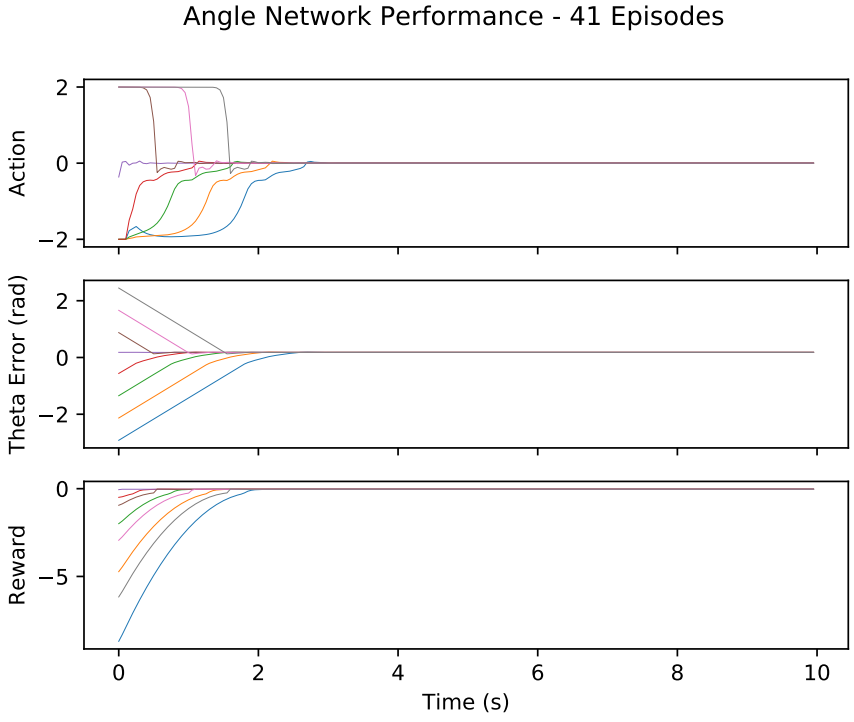


Figure H.3: Angle Network Performance – 41 Episodes

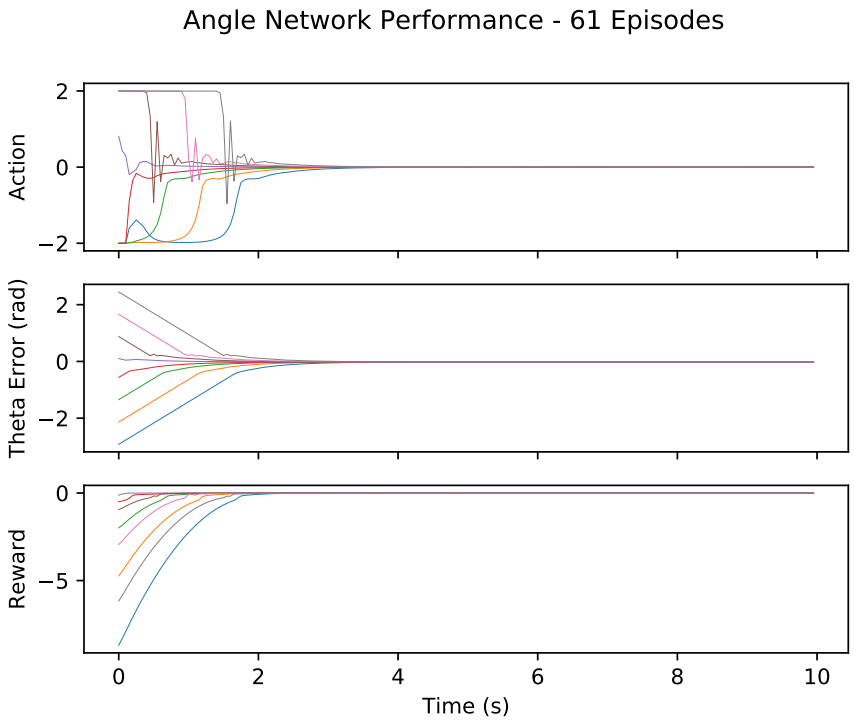


Figure H.4: Angle Network Performance – 61 Episodes

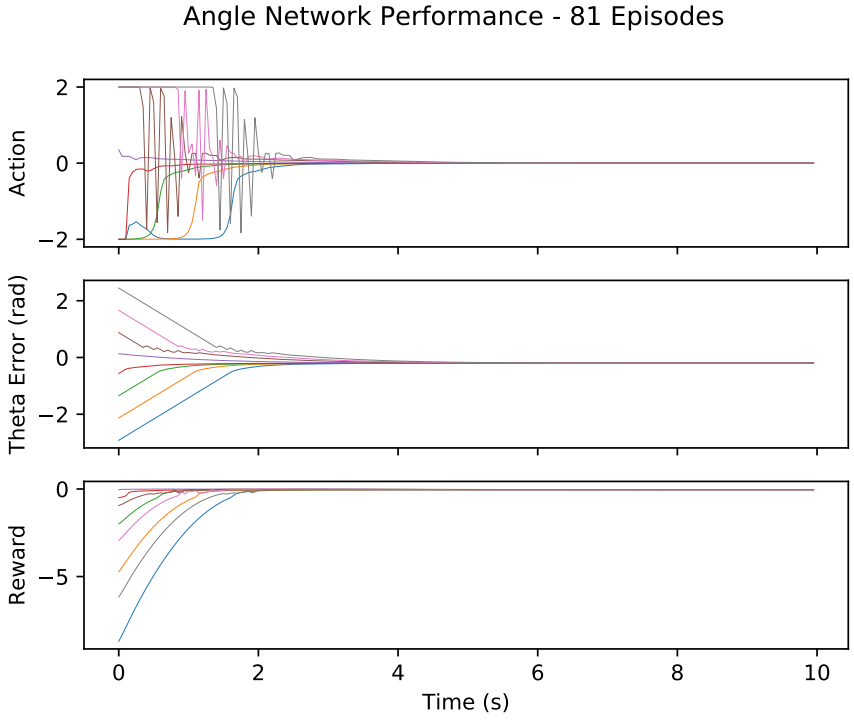


Figure H.5: Angle Network Performance – 81 Episodes

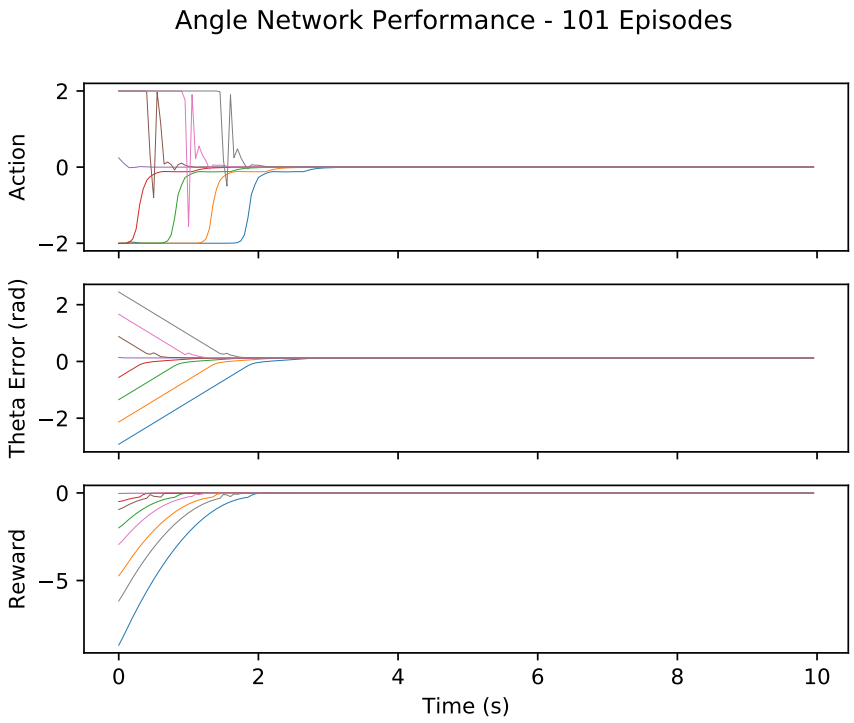


Figure H.6: Angle Network Performance – 101 Episodes

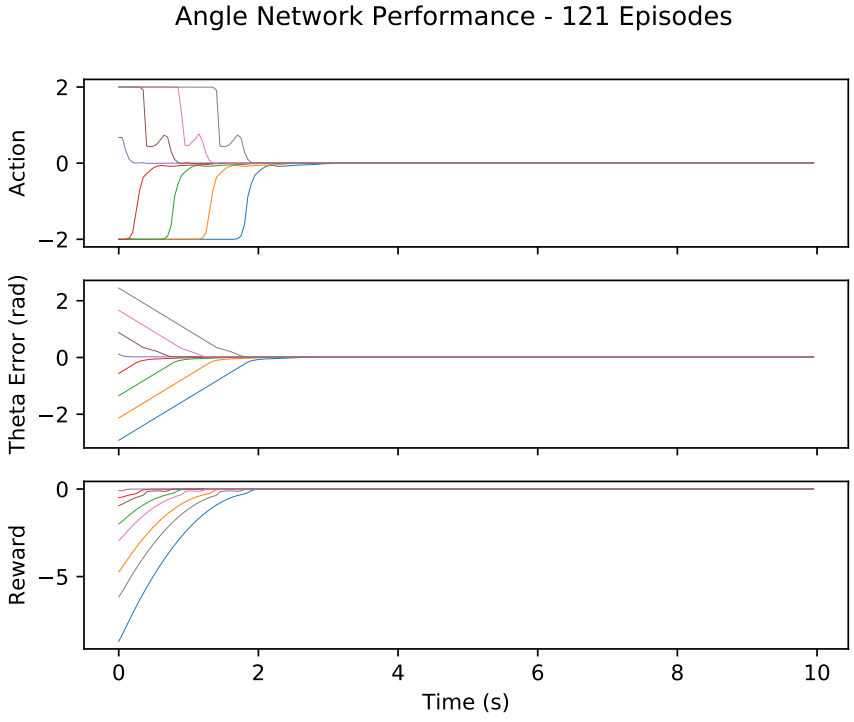


Figure H.7: Angle Network Performance – 121 Episodes

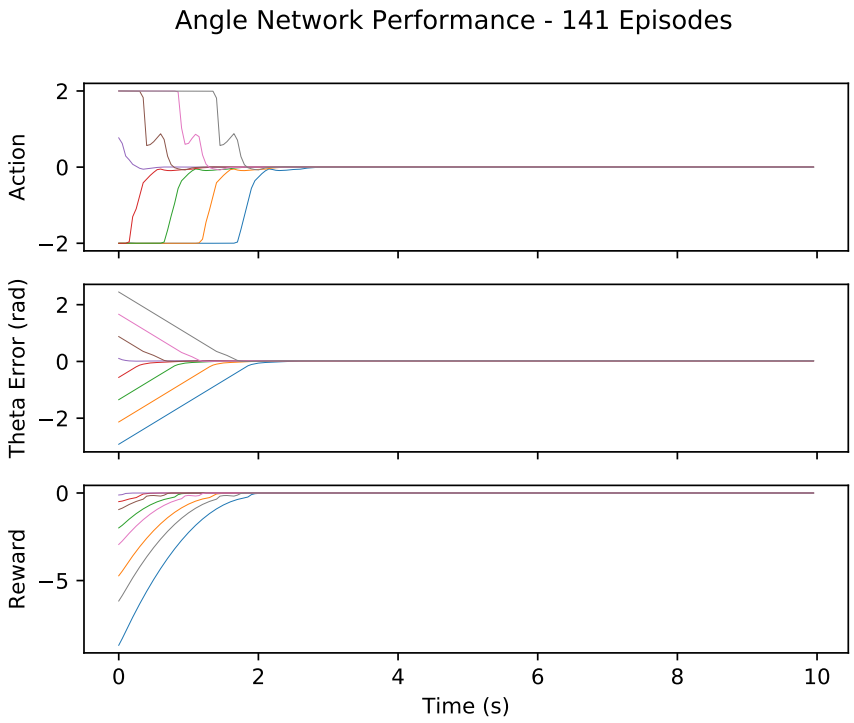


Figure H.8: Angle Network Performance – 141 Episodes

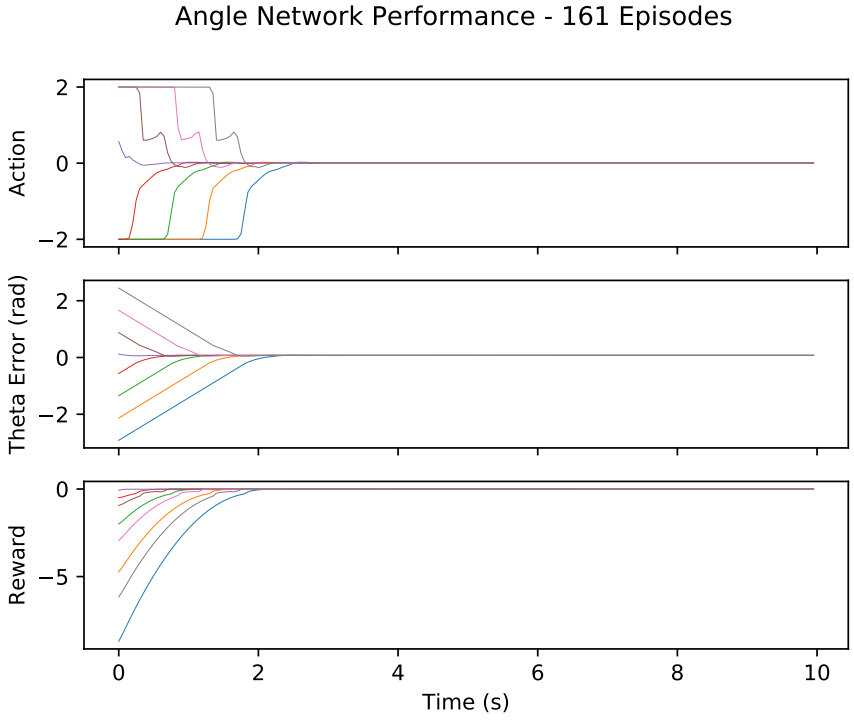


Figure H.9: Angle Network Performance – 161 Episodes

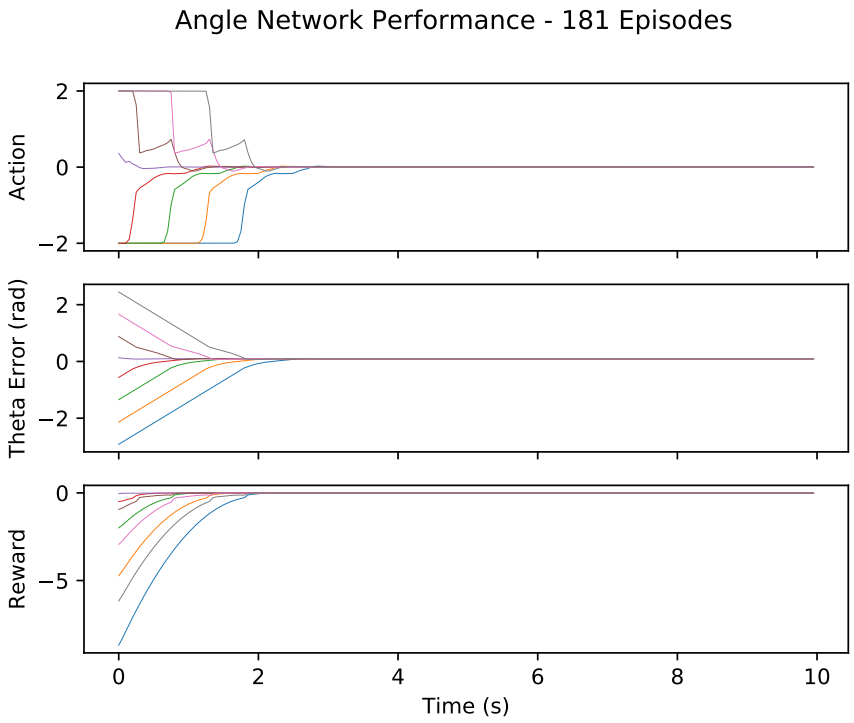


Figure H.10: Angle Network Performance – 181 Episodes

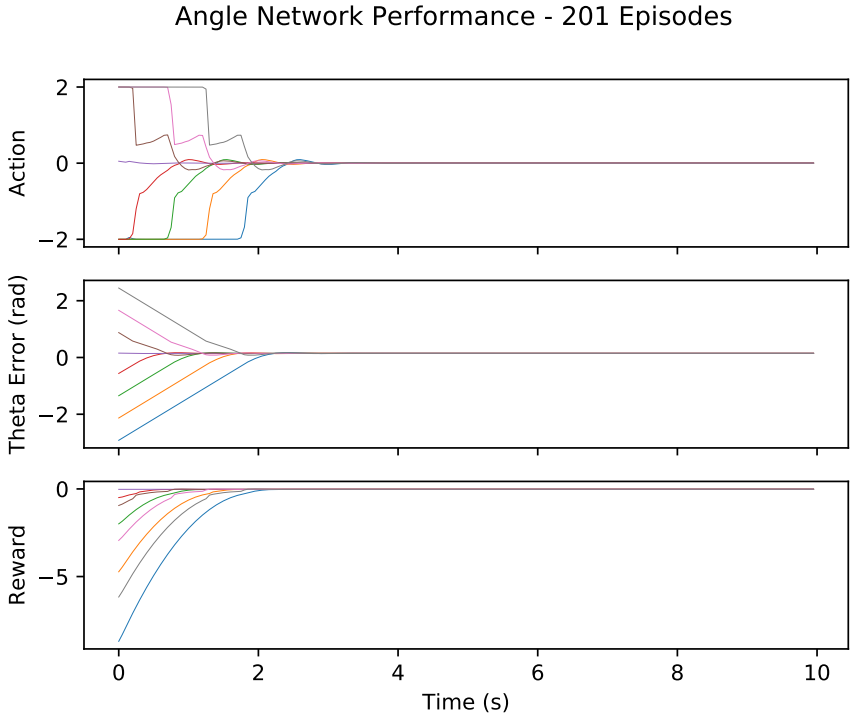


Figure H.11: Angle Network Performance – 201 Episodes

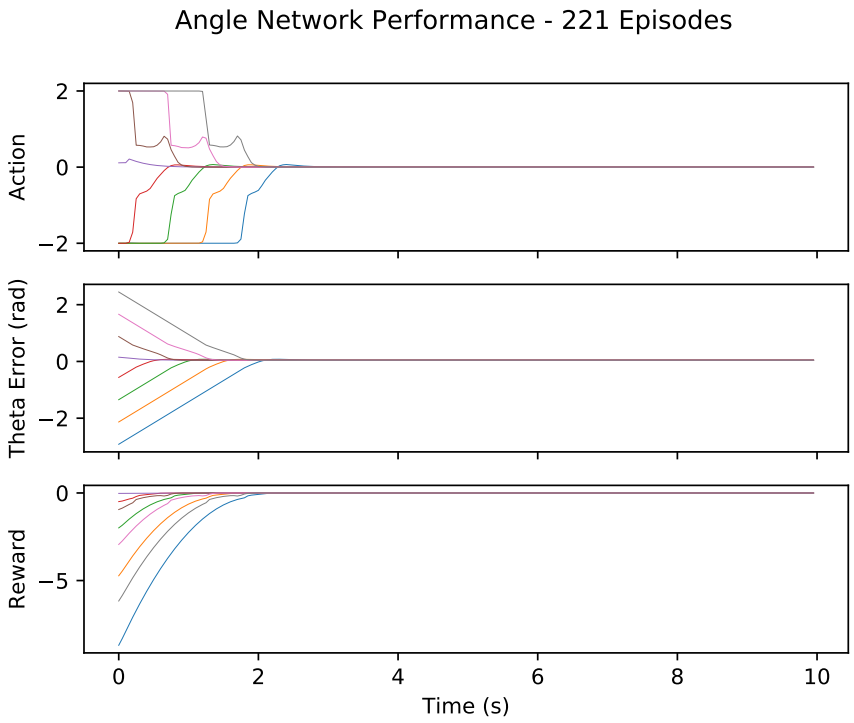


Figure H.12: Angle Network Performance – 221 Episodes

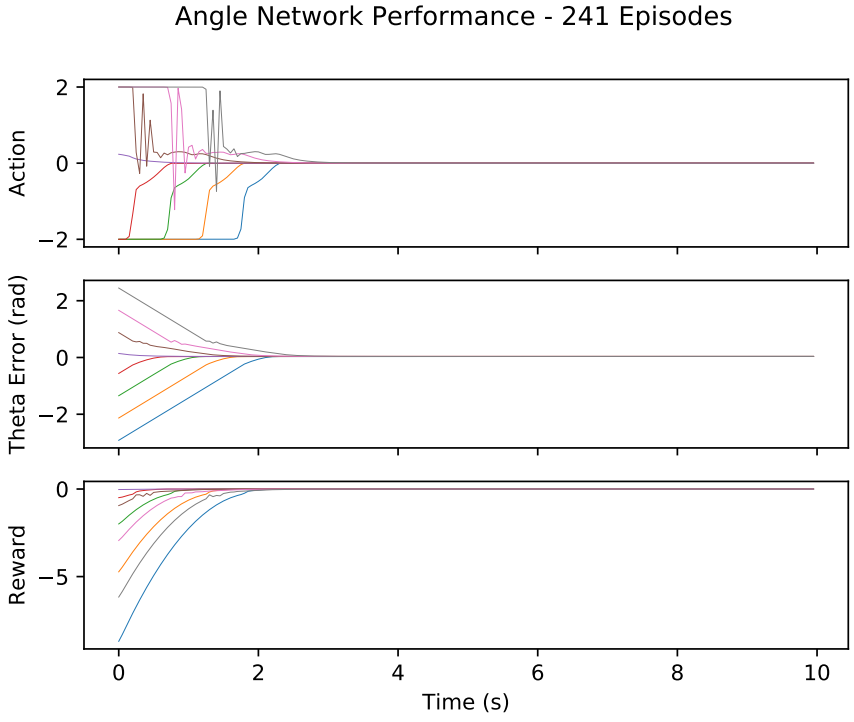


Figure H.13: Angle Network Performance – 241 Episodes

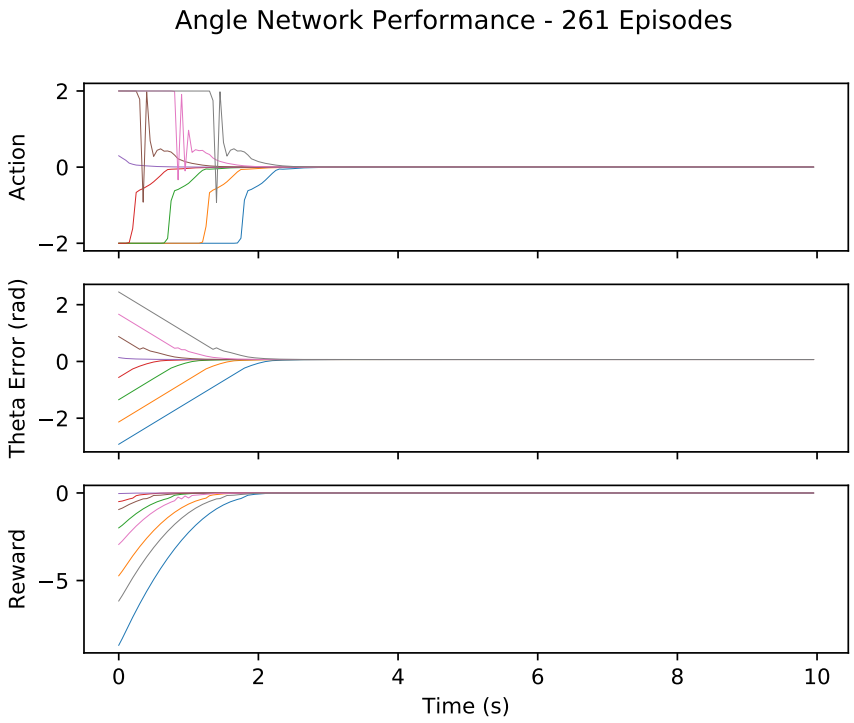


Figure H.14: Angle Network Performance – 261 Episodes

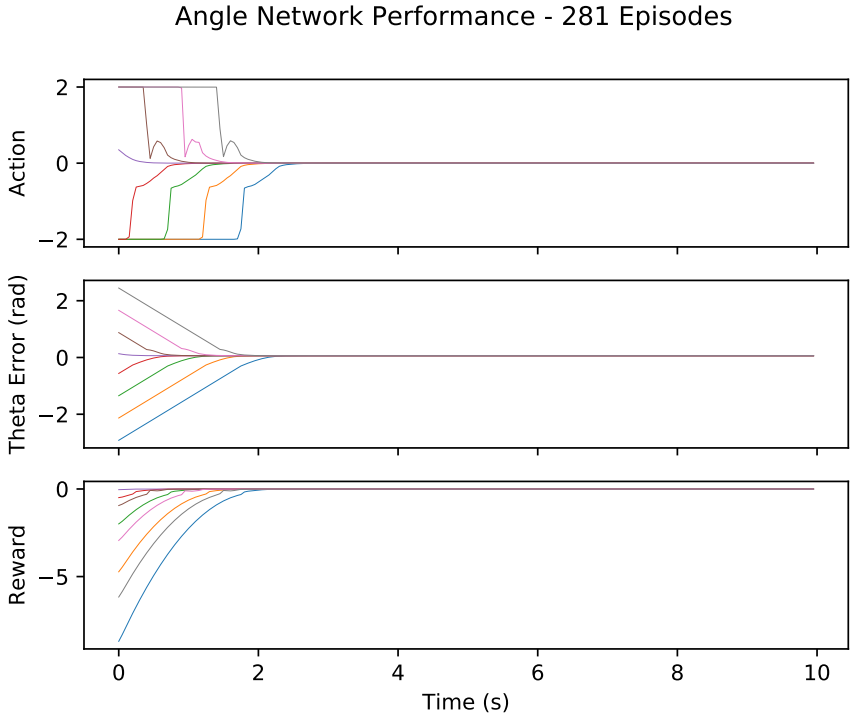


Figure H.15: Angle Network Performance – 281 Episodes

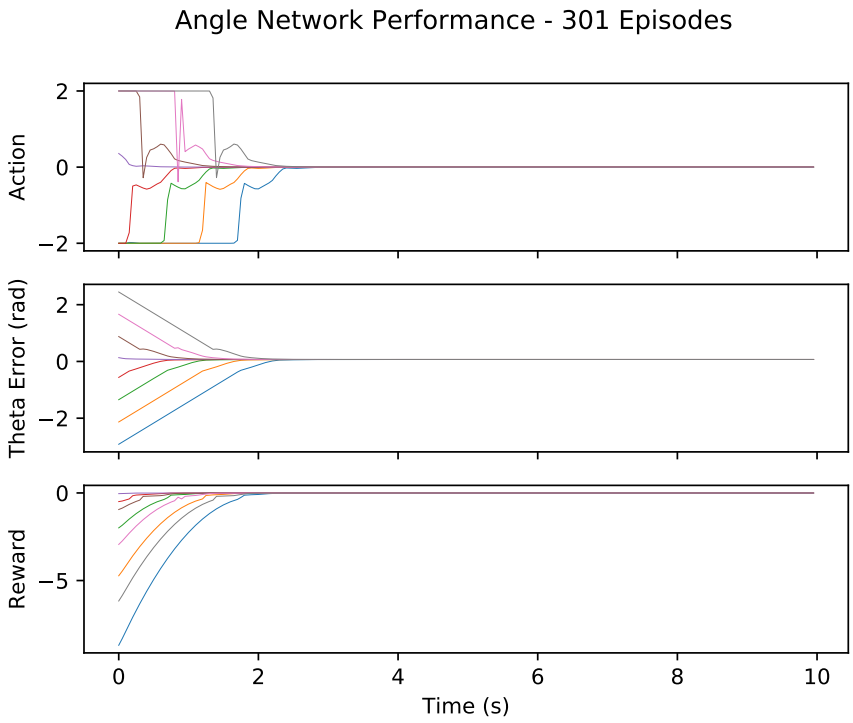


Figure H.16: Angle Network Performance – 301 Episodes

Appendix I

SINGLE ACTOR NETWORK PERFORMANCE

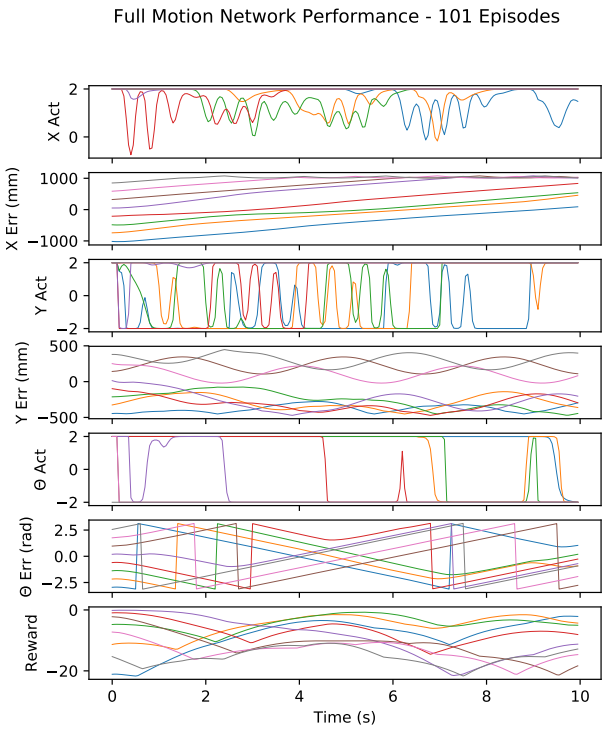


Figure I.1: Single Actor Network Performance – 101 Episodes

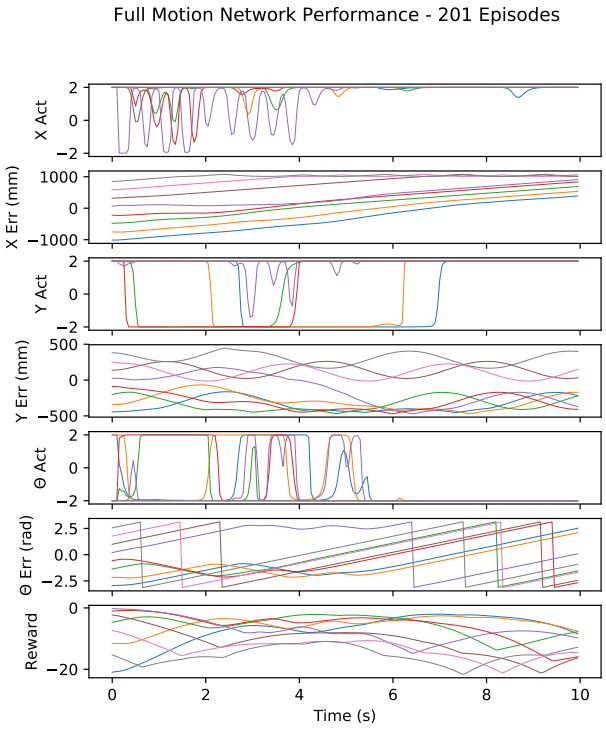


Figure I.2: Single Actor Network Performance – 201 Episodes

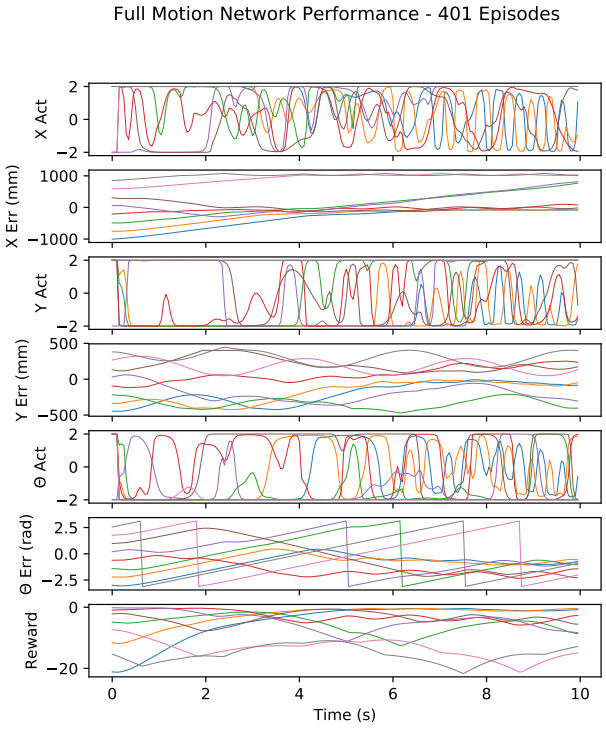


Figure I.3: Single Actor Network Performance – 401 Episodes

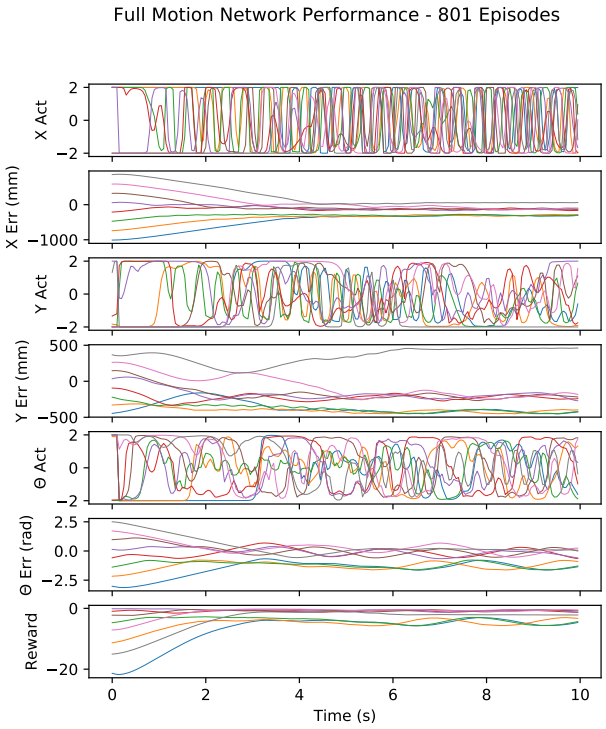


Figure I.4: Single Actor Network Performance – 801 Episodes

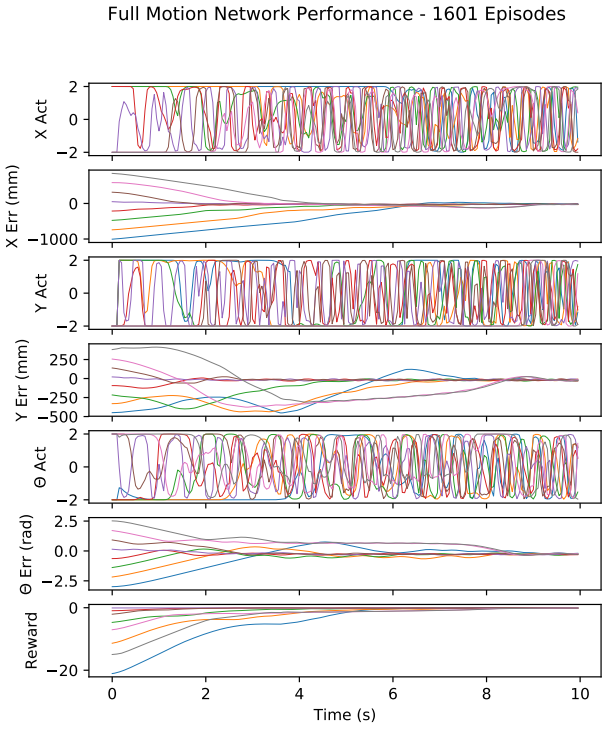


Figure I.5: Single Actor Network Performance – 1601 Episodes

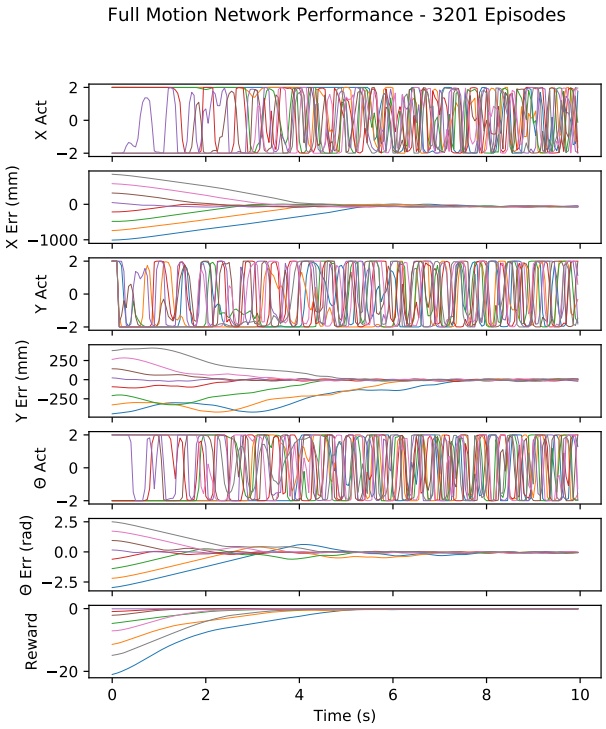


Figure I.6: Single Actor Network Performance – 3201 Episodes

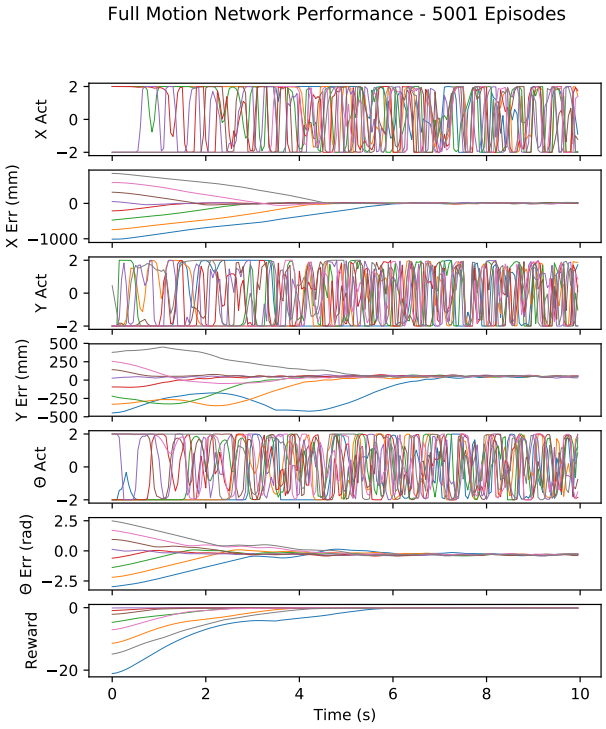


Figure I.7: Single Actor Network Performance – 5001 Episodes

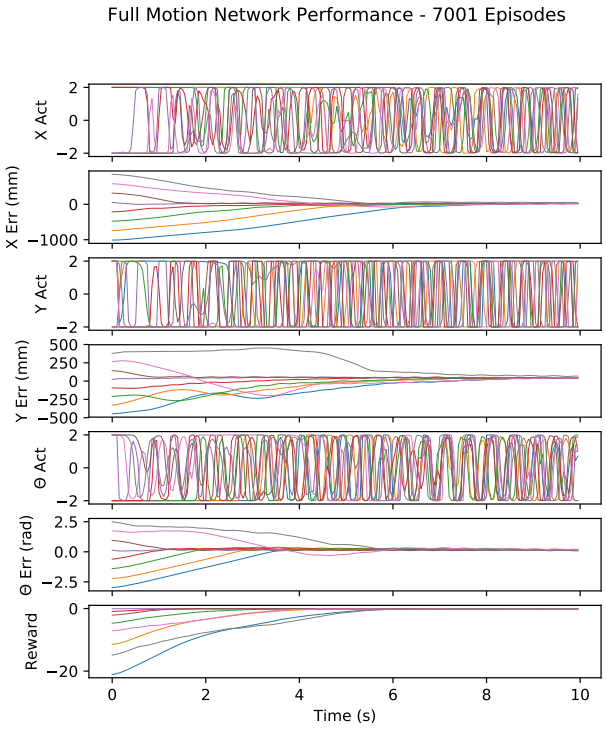


Figure I.8: Single Actor Network Performance – 7001 Episodes

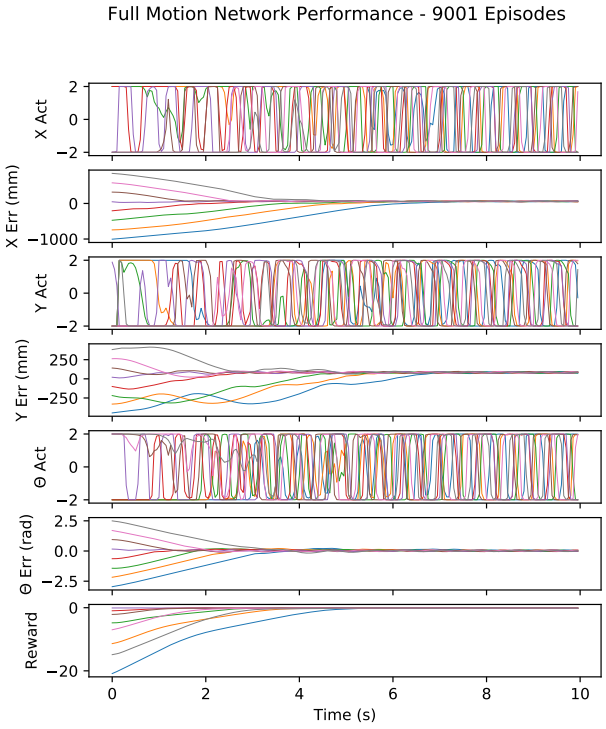


Figure I.9: Single Actor Network Performance – 9001 Episodes

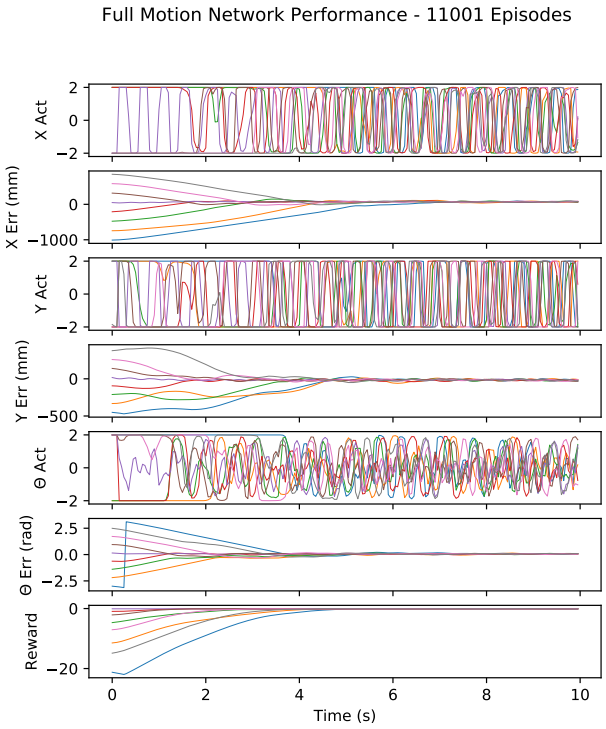


Figure I.10: Single Actor Network Performance –11001 Episodes

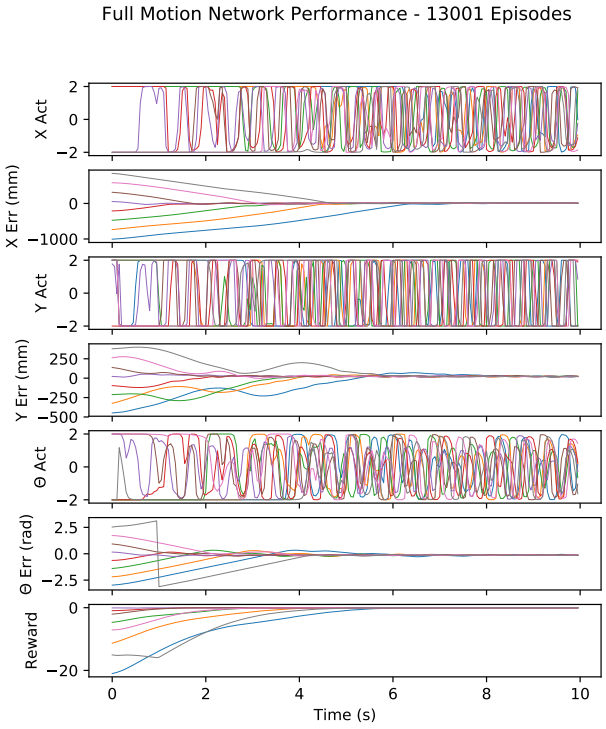


Figure I.11: Single Actor Network Performance – 13001 Episodes

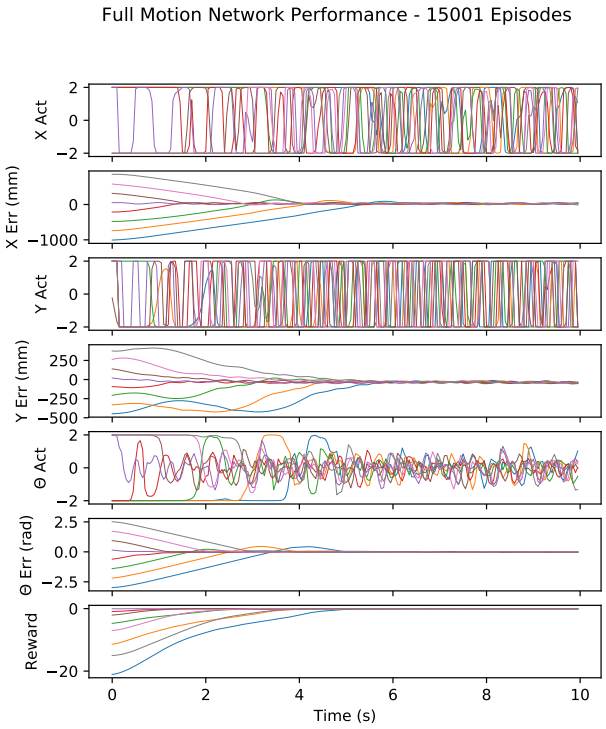


Figure I.12: Single Actor Network Performance – 15001 Episodes

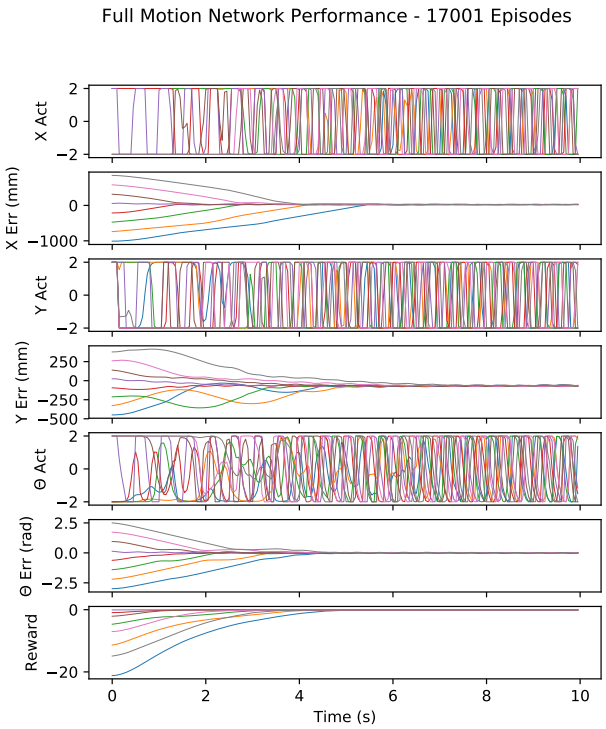


Figure I.13: Single Actor Network Performance – 17001 Episodes

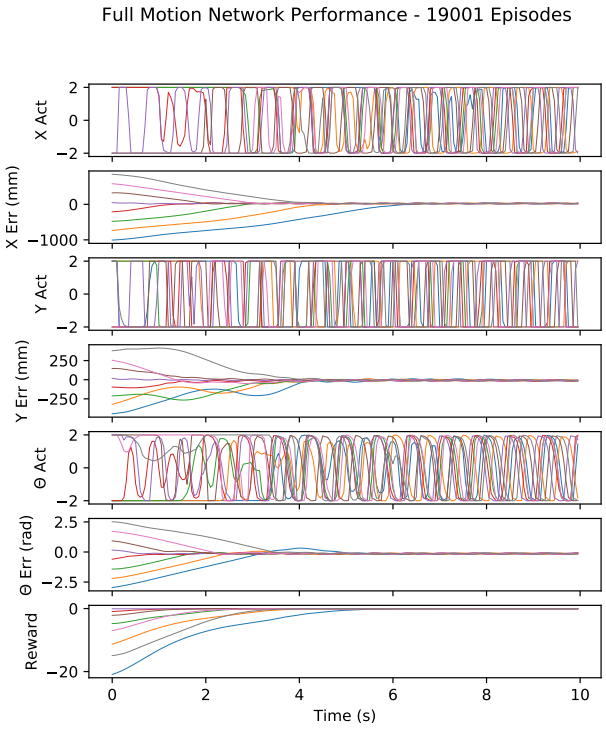


Figure I.14: Single Actor Network Performance – 19001 Episodes