

GRANT: Ground Roaming Autonomous Neuromorphic Targeter

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Jonathan Daniel Ambrose

May 2020

© by Jonathan Daniel Ambrose, 2020
All Rights Reserved.

*In memory of my grandparents, Don and Wilma Ambrose,
who always knew I would get here but didn't get to see me make it...*

Acknowledgments

I wish to thank my primary research advisor, Dr. Mark Dean, for all of his help throughout the completion of this project. His continual support throughout the biggest challenges that this project presented, coupled with advice, attention to detail, and aid in pushing through the challenges was invaluable.

I would also like to express gratitude towards members of the TENNLab Neuromorphic Research Group, both present and former. I thank Grant Bruer for his guidance in modifying the simulator to add in a target. I also especially thank Aaron Young for his advice throughout the entire design and implementation of the robot. His support and help in debugging the numerous issues presented by the robot could not have went unnoticed. I also thank all of the other unmentioned members of the research group who have contributed to this project.

A special thanks must be extended to Adam Foshie for the nearly inexhaustible list of ways in which he aided in the development of the robot. For his initial help in the development of the robot itself, help in debugging issues in both software and hardware, creation of documentation for the robot, power delivery board design, and never-ending support, I offer my sincerest gratitude.

Furthermore, I would like to thank my parents for their support, emotionally and financially, throughout the entirety of this project. I could never repay the kindness and love they have shown me throughout the pursuit of my education.

Lastly, to my girlfriend, Karen, I offer my heart-felt appreciation. From just being there to listen to my frustrations to providing emotional support and encouragement when I thought the completion of this project would be impossible, I truly can never thank her enough.

Abstract

Within the last 50 years, researchers have shifted from conventional programming paradigms to machine learning based approaches. Further development has been done to apply this artificial intelligence to the task of robotic control; however, Deep Neural Networks (DNNs) and Artificial Neural Networks (ANNs) created by these machine learning approaches often require too much computational intensity to be directly run on an embedded platform. In more recent times, focus has shifted toward the application of Recurrent Spiking Neural Networks (RSNNs) to these embedded systems, because these networks have a temporal element relevant to control problems, and small RSNNs have been shown to accomplish complex tasks. This work addresses the design, implementation, training, and testing of a neuromorphic robot using RSNNs. Specifically, the Ground Roaming Autonomous Neuromorphic Targeter (GRANT) uses the second generation Dynamic Adaptive Neural Network Array (DANNA2) neuromorphic processor to accomplish a variety of objectives, such as obstacle avoidance, grid coverage, object targeting, and object pursuit. Furthermore, the use of the TENNLab neuromorphic framework allows for the training of small RSNNs capable of running on a resource-constrained platform. This work also discusses how DANNA2 arrays may be dynamically reconfigured in real-time, such that multi-objective tasks may be accomplished by a set of these small networks. Finally, results are presented discussing a variety of distinct network performances against traditional algorithms, as well as testing the capabilities of such networks to be ran in simulation or on a remote host as the DNNs and ANNs of many other neuromorphic platforms would do. It was found that not only can these networks perform nearly as well as traditional algorithms, even when given less information, but these networks can also be run directly on the hardware of an embedded platform without the need for an external host.

Table of Contents

1	Introduction	1
1.1	Overview	2
2	Related Work	3
2.1	Traditional Methods of Robotics	3
2.1.1	Sonar-based SLAM	3
2.1.2	LIDAR-based SLAM Supplemented with an IMU on a UAV	5
2.2	Neuromorphic Robotics	6
2.2.1	Trail Following using the IBM NS1e	6
2.2.2	Obstacle Avoidance and Target Acquisition with the ROLLS Processor	8
3	Previous Work	10
3.1	TENNNLab Framework	10
3.2	EONS	12
3.3	NeoN	12
3.4	DANNA2	14
3.4.1	Architecture	15
3.4.2	Network Array Types	17
4	Objective for GRANT	20
5	Hardware Design	21
5.1	Control System	21
5.2	Chassis	25

5.3	Power Delivery	26
5.4	3D-Printed Parts	27
6	Digital Design	33
6.1	Base Robot Project	33
6.2	Modifications for Robot Control	34
6.3	Addition of Neuromorphic Processor	35
7	Software	37
7.1	Petalinux	37
7.2	Robot Programming	38
7.2.1	Existing Libraries	38
7.2.2	Newly Created Libraries and Classes	40
7.2.3	Common Algorithm Structure	42
7.2.4	Traditional Algorithm Implementation	42
7.2.5	Neuromorphic Algorithm Implementation	44
7.3	Simulator Modifications	45
7.3.1	Traditional Algorithms in Simulation	46
8	Training and Testing	47
8.1	Network Input and Output	47
8.1.1	Roaming	47
8.1.2	Targeting	48
8.2	Training Methodologies	51
8.2.1	Islands Training Construct	51
8.3	Simulator Parameters	52
8.3.1	Moving Target	54
8.3.2	Cubic Obstacles	55
8.3.3	Crash Penalty	55
8.3.4	Equal Turning Weight	56
8.3.5	Time Step Length	57

8.3.6	Cycles Between Sensor Updates	58
8.3.7	Weak and Bad Time Steps	58
8.3.8	Rotated Start Angle	59
8.3.9	LIDAR Beam Width	60
8.3.10	Adapting Training Rooms	60
8.4	Hardware and Software Complications	66
8.4.1	Petalinux	66
8.4.2	Wheel Problems	66
8.4.3	Power Delivery	67
8.5	Tuning Robot Parameters	68
8.6	Removing Network Restrictions	69
9	Results	71
9.1	Simulated Performance	72
9.1.1	Behavioral Performance	72
9.1.2	Performance Metrics	76
9.1.3	Network Size and Connectivity	78
9.2	Robot Performance	82
9.2.1	Behavioral Performance	82
9.2.2	Update Evaluation Time	83
10	Future Work	86
10.1	Hardware and Architecture	86
10.2	Running Networks Concurrently	87
10.3	Target Interaction	88
10.4	Navigating a 3D Environment	88
11	Conclusion	90
	Bibliography	91

Appendix	96
A Wiring Schematic	97
B Bill of Materials	98
Vita	99

List of Tables

8.1	Grid coverage input scheme	49
8.2	Grid coverage output scheme	49
8.3	Targeting input scheme	50
8.4	Targeting output scheme	51
8.5	Summary of parameter effects on training	53
B.1	Bill of Materials	98

List of Figures

2.1	Sonar map after thresholding	4
2.2	Sonar obstacles after thresholding (height represents certainty)	5
2.3	Architecture of the LIDAR/MEMS IMU SLAM algorithm	6
2.4	CARLorado robot with NS1e	7
2.5	Networks utilized by PushBot	8
3.1	TENNNLab hardware/software co-design framework	11
3.2	Example reproductive operations performed by EONS	12
3.3	NeoN	13
3.4	NeoN digital architecture	14
3.5	DANNA2 connectivity pattern	15
3.6	DANNA2 element block diagram	16
3.7	Configuration Packet Description	18
3.8	Step and Fire Packet Description	18
3.9	Reset Packet Description	19
3.10	Output Fire Packet Description	19
5.1	High-level diagram of GRANT hardware design	22
5.2	Front view of an assembled GRANT robot	22
5.3	Internal view of an assembled GRANT robot	23
5.4	Example Pixy2 Target Recognition	24
5.5	Chassis used for GRANT	25
5.6	Power distribution on GRANT	26
5.7	Circuit diagram of undervoltage circuit	27

5.8	Battery holder CAD model	29
5.9	Bottom panel CAD model without components	29
5.10	Bottom panel CAD model with components	30
5.11	Top panel CAD model without components	30
5.12	Top panel CAD model with components	31
5.13	Outside piece of wheel adapter	31
5.14	Inside piece of wheel adapter	32
6.1	GRANT digital design block diagram	34
6.2	Architecture adding the DANNA2 processor	35
7.1	Structure of the software stack	38
7.2	Example of LIDAR binning	41
7.3	Generic Robot Algorithm Structure	43
7.4	Decision tree for traditional algorithms	44
8.1	Simulator explanation and network input information	48
8.2	Example grid coverage network	49
8.3	Example targeting network	52
8.4	Small room 1	61
8.5	Small room 2	62
8.6	Small room 3	62
8.7	Big room 1	63
8.8	Big room 2	63
8.9	Big room 3	64
8.10	Hard room 1	64
8.11	Hard room 2	65
8.12	Hard room 3	65
8.13	Original rubber wheel	67
8.14	Banebot rubber wheel	68
8.15	Omni-directional wheel	69

8.16	Model of the SNACC system	70
9.1	Grid coverage performance	72
9.2	Grid_7 example path	73
9.3	Grid_33 example path	74
9.4	Sparse_33 example path	75
9.5	Trad_7 example path	75
9.6	Trad_33 example path	76
9.7	Implementation room results	77
9.8	Implementation average time to reach target	78
9.9	Network sizes for best networks	79
9.10	Grid_7 network	79
9.11	Grid_33 network	80
9.12	Sparse_33 network	81
9.13	Update time for cycles without network switches or sensor updates	84
9.14	Update time for cycles with network switches	84
A.1	Wiring schematic for a GRANT robot	97

Chapter 1

Introduction

Artificial intelligence is about replacing human decision making with more sophisticated technologies.

– Falguni Desai

Work in autonomous robotics and vehicles in the past 30 years has been centralized around the use of Artificial Neural Networks (ANNs) and Deep Neural Networks (DNNs) [2, 14]. The networks capable of controlling such systems, while successful, are incredibly large in size, such that many of these networks are composed of thousands of neurons and millions of synapses [7, 5]. Because of the size of these networks, they can rarely be implemented or simulated on a robot or vehicle due to the computational complexity and size required, encouraging researchers to use a remote, tethered, computationally powerful host to make decisions. In these systems, the robot simply transmits information to the host and then receives the decision that it should execute.

Due to the limitations of resources on these robots, researchers have begun to use Recurrent Spiking Neural Networks (RSNNs) that offer greater computational complexity with less resources through the use of encoding spatio-temporal data [41]. These researchers have worked on multiple methods of developing these RSNNs architecturally, both in hardware and software; however, virtually no work exists utilizing these architectures in an embedded system such as those mentioned above.

1.1 Overview

This work describes the development of an embedded, neuromorphic system that requires no remote processing. Furthermore, it describes the use of network reconfiguration in order to use independent networks to accomplish a multi-objective function. In [Chapter 2](#), traditional Simultaneous Localization and Mapping (SLAM) implementations [\[12\]](#) are explored as well as neuromorphic robots. In [Chapter 3](#), previous work accomplished by the TENNLab neuromorphic group is discussed, with specific work including the hardware/software co-design framework, evolutionary optimization of neuromorphic systems, a previous neuromorphic robot referred to as NeoN [\[21\]](#), and DANNA2, the newest neuromorphic processor developed by the group. In [Chapter 4](#), the purpose for this work is discussed. In [Chapter 5](#), the component selection process and the associated hardware design is presented. [Chapter 6](#) discusses the underlying architecture associated with this work. [Chapter 7](#) presents the development of both the traditional and neuromorphic algorithms used for this work, as well as the simulator used for the purpose of training. In [Chapter 8](#), information on training methodologies, training parameters, and the testing of the physical system is discussed. [Chapter 9](#) presents the findings of this work, including the performance of trained networks, the performance of traditional algorithms, and the capabilities of the system to use a remote host without connectivity restrictions is explored. In [Chapter 10](#), avenues to continue and improve upon this work are discussed.

Chapter 2

Related Work

The saddest aspect of life right now is that science gathers knowledge faster than society gathers wisdom.

– Isaac Asimov

2.1 Traditional Methods of Robotics

Since the work presented in this paper is focused around the ideas of obstacle avoidance and object targeting, this section presents classical approaches to these tasks. In particular, different systems utilizing Simultaneous Localization and Mapping (SLAM) approaches are presented.

2.1.1 Sonar-based SLAM

In 1987, Alberto Elfes published work detailing a methodology in which an array of sonar sensors could be used in order to perform a SLAM algorithm [6]. The apparatus consisted of sonar sensors, each with a 15 degree wide angular range, spaced evenly around an apparatus that could be affixed to a robot. This array utilized a microcontroller to collect samples from each sensor at regular intervals and sent the samples to a host computer where the map building could be done. Utilizing these readings, areas were predicted to be “probably empty” or “probably occupied.” Over time, readings were combined using a probabilistic model, such that more empty readings increased the certainty of empty spaces and more

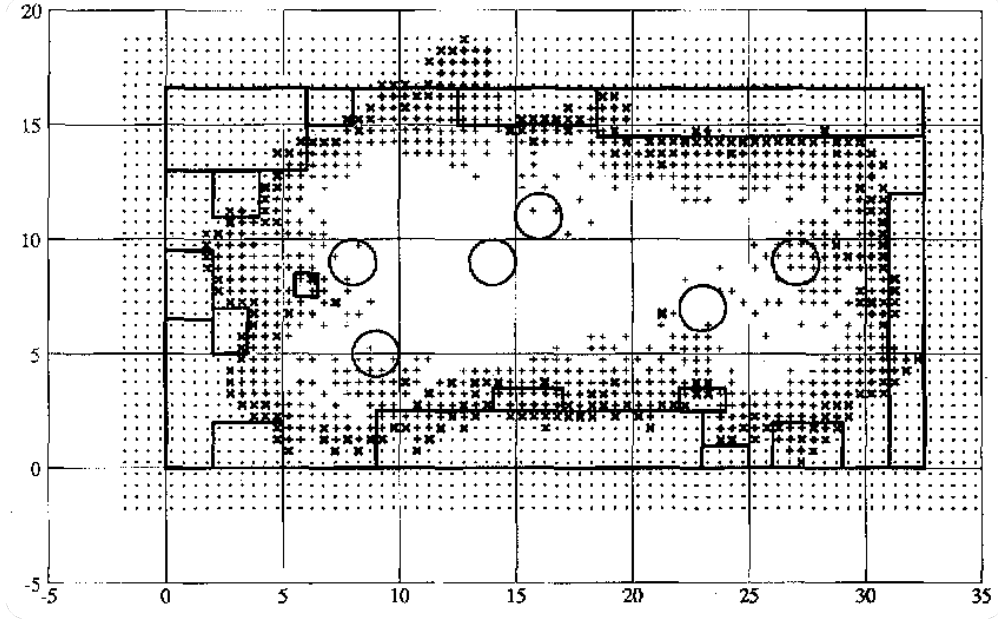


Figure 2.1: Sonar map after thresholding [6]

occupied readings increased the certainty of occupied spaces. Additionally, the resolution of these spaces became more clear as more readings were gathered, such that a full map of the room could be produced [6].

Results

Using his model, Elfes found success in creating proper maps of rooms. It was found that by using thresholding across the map, additional clarity could be resolved. Figures 2.1 and 2.2 show the mapping of a given room sampled with the system [6].

It can be seen that this map could be useful for path planning, as one could avoid obstacles with a high occupied certainty and also be able to recognize their position at any place throughout the room. Furthermore, Elfes theorizes that this system could also be useful for landmark recognition, as an outline of an object with a high certainty of occupancy would allow simple recognition of said object [6].

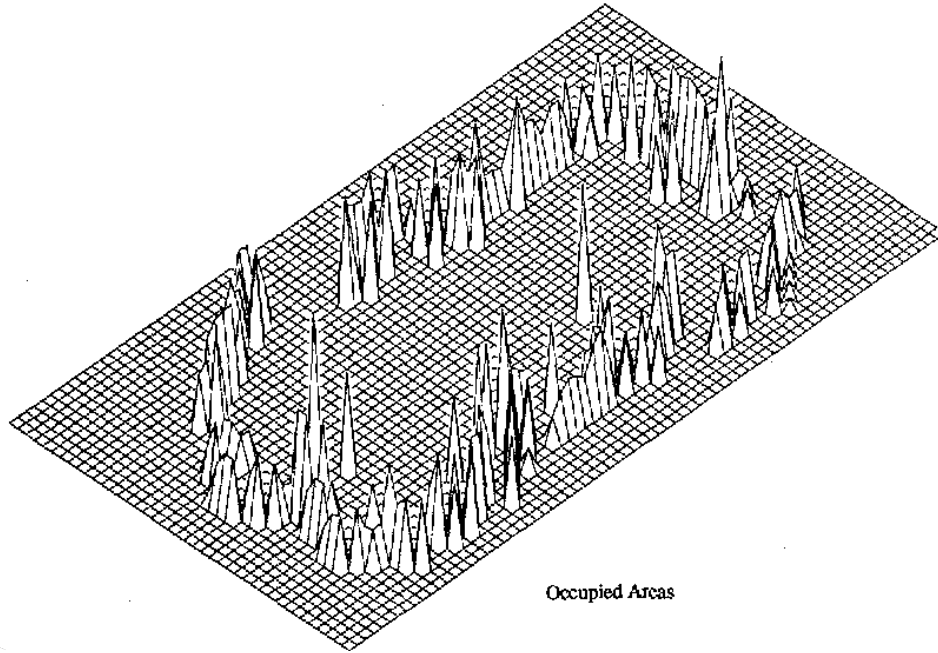


Figure 2.2: Sonar obstacles after thresholding (height represents certainty) [6]

2.1.2 LIDAR-based SLAM Supplemented with an IMU on a UAV

A group at the Nanjing University of Aeronautics and Astronautics published work that discussed a method by which one could supplement LIDAR-based SLAM algorithms with an Micro-Electro-Mechanical System (MEMS) Inertial Measurement Unit (IMU) for work on a Unmanned Aerial Vehicle (UAV) [13]. In essence, this work sought to use the IMU to detect the rotation of the UAV such that the three-dimensional mapping of the room may be maintained. A LIDAR point cloud was taken from the system and converted to a position in space using the orientation of the drone gathered from the IMU. This data was then used to detect edges and corners, perform a distance calculation, and update the map accordingly [13]. The group's presented architecture is presented in Figure 2.3.

Results

In practice, the group found that the addition of the IMU proved to increase the accuracy of the system. The projected trajectory derived from the combination of the LIDAR/IMU was smoother than that of the LIDAR only approach. They also found that the resulting path of

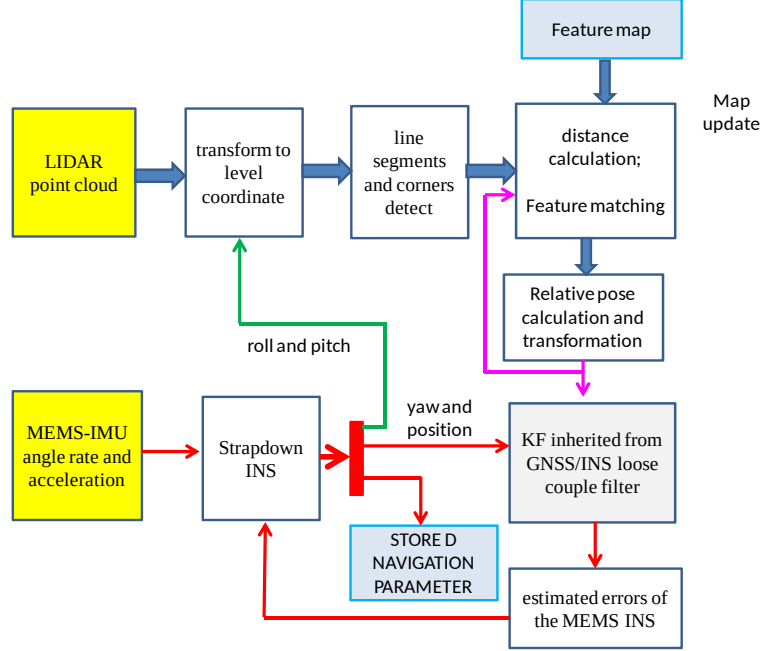


Figure 2.3: Architecture of the LIDAR/MEMS IMU SLAM algorithm [13]

the robot seemed more robust in exploration of the environment and that this methodology suffered from less error than the LIDAR only approach [13].

2.2 Neuromorphic Robotics

In this section, robots utilizing neuromorphic hardware are explored, including a neuromorphic robot that performs similar tasks to those discussed for this work. Furthermore, these systems use varying network topologies and architectures in order to accomplish their independent tasks.

2.2.1 Trail Following using the IBM NS1e

The University of Maryland partnered with the University of California, Irvine in order to create a robot capable of following a mountainous trail in Colorado. The robot utilized the IBM NS1e, which contains 4096 TrueNorth neuromorphic cores [1]. This board was affixed to the CARLorado robot which is a robotic platform controlled by a Samsung Galaxy S5 smart

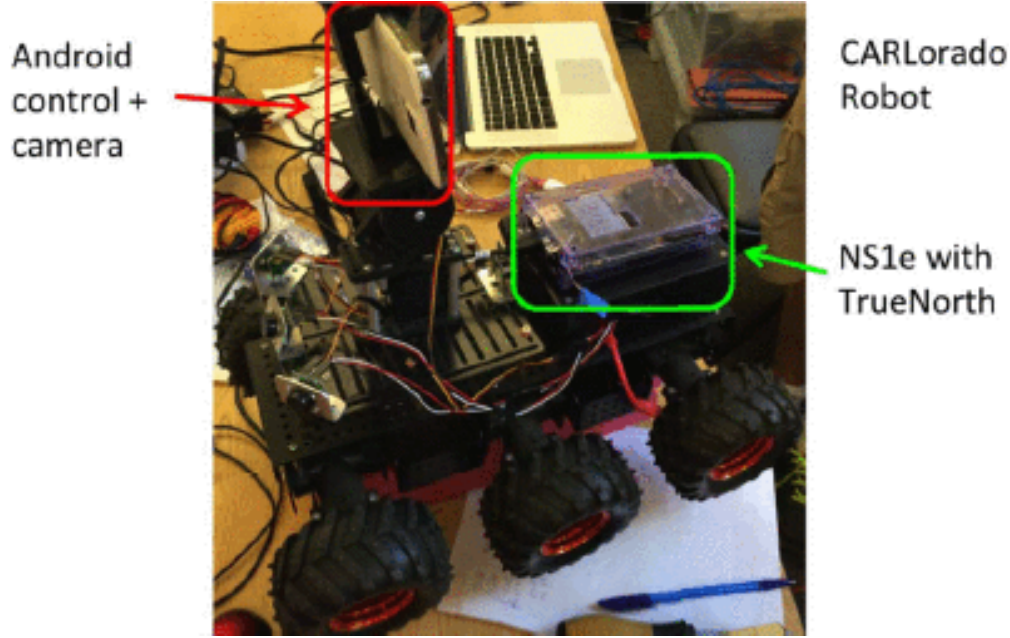


Figure 2.4: CARLorado robot with NS1e [9]

phone as seen in Figure 2.4. With both of these components on the robot, a local, wireless hotspot was used for the two to communicate with one another [9].

Network Development

In order to properly train a deep convolutional neural network to control the robot, a dataset had to be developed. The dataset was created by manually driving the robot along the trail four times and recording the decisions made by the driver. The combination of the decisions made by the driver and the scenes captured by the camera combined to form a classification dataset with classifications indicating whether the robot should turn right, turn left, or move forward. The network was trained with backpropagation and then was translated to be an Energy-Efficient Deep Neuromorphic Network (EDNN), a type of network proven to perform well and map convolutional networks onto the spiking hardware of TrueNorth [9].

Results

In testing, the resulting network performed well in the control application. It was capable of navigating the 0.5 kilometer trail presented in the dataset; however, on occasion, the robot

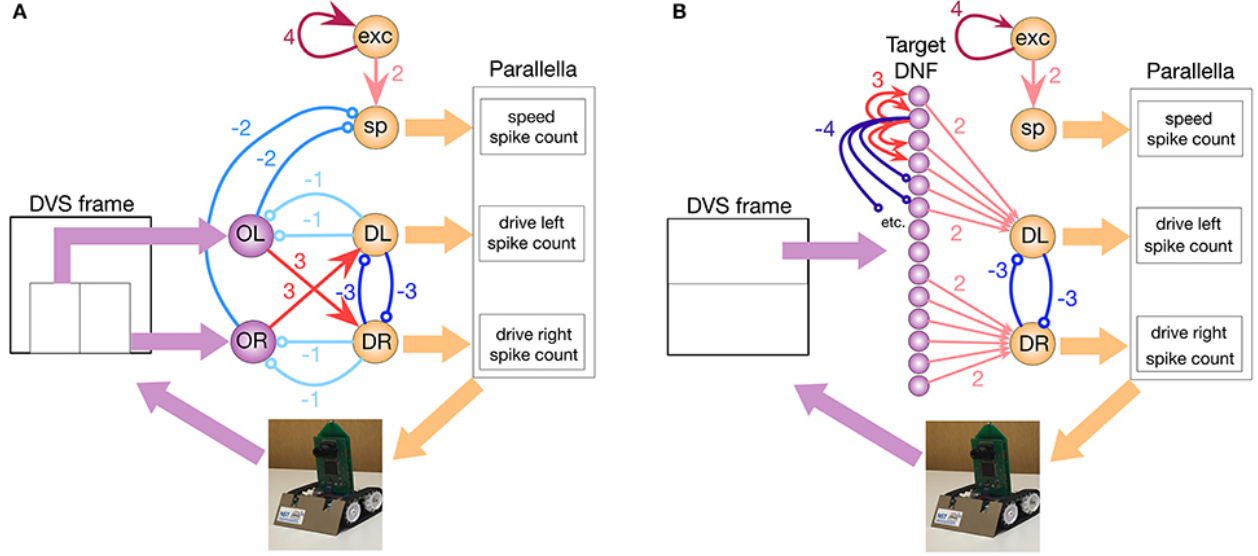


Figure 2.5: Networks utilized by PushBot [9]

required human intervention when it drifted off of the trail. Nonetheless, this robot proved to be an impressive system capable of performing a real-time control application using neural networks on neuromorphic hardware [9].

2.2.2 Obstacle Avoidance and Target Acquisition with the ROLLS Processor

Researchers at the University of Zurich and the Technical University of Munich worked together to create another neuromorphic robot. This robot was tasked to perform obstacle avoidance and object targeting. The robot was to do this by sending sensor data from a Dynamic Vision Sensor (DVS) camera to a Reconfigurable On-line Learning Spiking (ROLLS) neuromorphic processor wirelessly, and then executing the decision chosen by the network [19].

Networks

The networks discussed in this section, along with their inputs and outputs, may be viewed in Figure 2.5.

Instead of training networks, the networks used in this work were manually created. The manufactured obstacle avoidance network used groups of neurons to represent areas of the camera’s vision for the left and right sides in order to detect obstacles. These groups were connected through synapses to output neurons corresponding to the decisions of turning left or right. In total, this network required 96 neurons [19].

Similarly, for object targeting, populations of neurons were created to detect the target in a given region of the camera’s vision. These neurons were then connected to the output neurons just as those in the obstacle avoidance network were. This network required 128 neurons in total [19].

In order to both avoid obstacles and move towards a target simultaneously, both networks were ran simultaneously, with a function to combine the output decisions of both networks into a singular decision to be used by the robot. In this function, priority was given to the obstacle avoidance network, such that the network would prioritize avoiding obstacles over pursuit of the target [19].

Results

Evaluating the physical system, it was found that this robot was capable of avoiding obstacles while navigating the room at a speed of 0.5 meters per second. This performance was recorded in a variety of rooms, including ones with variably sized obstacles. Additionally, this performance was repeatable, despite the fact that the ROLLS processor inherently has inconsistent behavior due to the analog components that it uses [19].

For targeting, it was found that the network was often incapable of discerning the difference between an obstacle and a target. In 80% of the test cases in which the robot approached the target, the robot believed that the target was an obstacle and turned away. While the robot exhibited some impressive behavior for a neuromorphic system, it did not achieve as favorable of results as one may hope for [19].

Chapter 3

Previous Work

*Our future success is directly proportional to our ability to understand,
adopt and integrate new technology into our work.*

– Sukant Ratnakar

In this chapter, we discuss previous work exploring methodologies associated with neuromorphic computing that make up the foundation that this work is built upon.

3.1 TENNLab Framework

The TENNLab hardware/software co-design framework serves as a software method of linking together applications, neuromorphic processors, and learning/training algorithms together [28]. The general structure of this framework is presented in [Figure 3.1](#). Applications are essentially simulations of a problem to be solved, with these simulations implementing a common application programming interface (API) that can communicate with the framework. Additionally, these applications implement a method of scoring performance in the form of a fitness function as well as a set of inputs and output for a neural network.

A variety of neuromorphic processors are implemented as part of the framework. These processors also implement a common API for use with the framework, and this includes functions for converting a network to and from a generic graph model that aids in the common interface for the framework. In this way, learning and modification operations can be done generically on this graph model and each independent processor will be able to

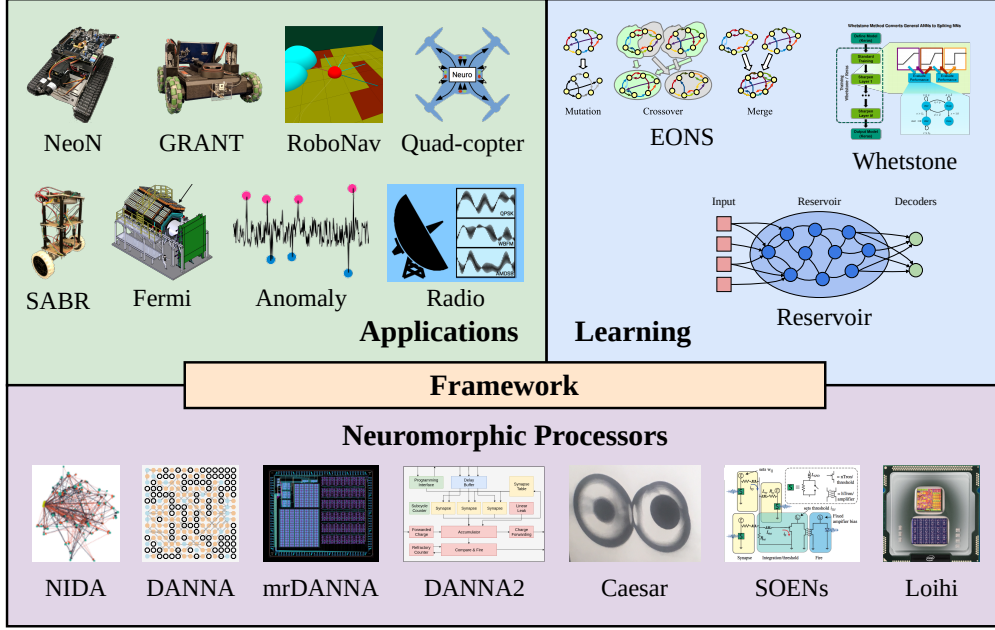


Figure 3.1: TENNLab hardware/software co-design framework [28]

import the resulting graph back as a model for simulation. The backend of each processor can be implemented in a unique way such that a variety of processors may be implemented for use with the framework.

A variety of learning algorithms may be implemented, as well, such that any of them may be used to train networks for a given neuromorphic processor and application. In particular, a genetic algorithm, Evolution Optimization of Neuromorphic Systems (EONS), is the primary learning algorithm used with the framework; however, other learning methods have been implemented such as backpropagation [36, 35] and reservoir computing [32].

Through the use of this framework, a given application can be trained with a variety of processors and a myriad of learning algorithms in order to achieve optimal performance. Additionally, if a given processor offers a hardware implementation and the neuromorphic processor simulator supports hardware compatibility, the trained networks can be put onto hardware as a physical implementation. In this way, the framework offers a lot of versatility and tools in order to create a usable neuromorphic system for a given application. Furthermore, the suite of applications may serve as a testing ground for a neuromorphic processor, such that the processor’s usability may be evaluated against a known sample space [27, 28].

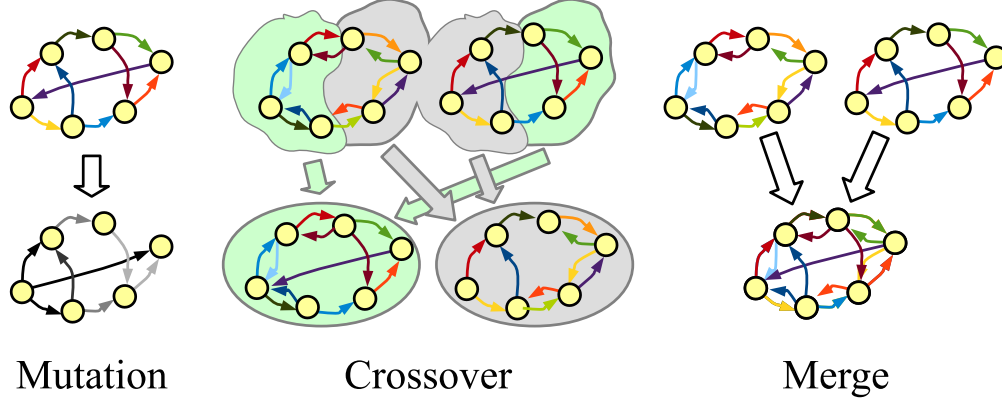


Figure 3.2: Example reproductive operations performed by EONS [34]

3.2 EONS

EONS functions as a learning algorithm that uses a generic graph representation of neural networks in order to perform operations on them [34]. The learning algorithm first creates a population of random graphs with the given number of inputs and outputs specific to an application. These graphs are passed to the neuromorphic processor to convert them into the specific network for that processor, and then the application is run using each of these networks. The performance for each network is stored and returned to the learning algorithm.

Upon receiving the performance scores for each of the graphs, EONS chooses a configurable number of the best networks to act as seeds for the next generation of potential networks. It performs operations such as crossover, duplication, mutation, and merging of graphs to act as reproduction as depicted in Figure 3.2. The resulting graphs from the crossover, duplication, and merging operations may then be mutated by modifying synapse (edge) weights, neuron (node) thresholds, or adding/removing synapses and neurons. These graphs are then sent to the neuromorphic processor for this learning operation to continue [34].

3.3 NeoN

NeoN (Neuromorphic Control System for Autonomous Robotic Navigation) was a robot designed to perform obstacle avoidance and grid coverage using a neuromorphic processor implemented in hardware [21]. The robot consisted of a tank drive frame with a Xilinx

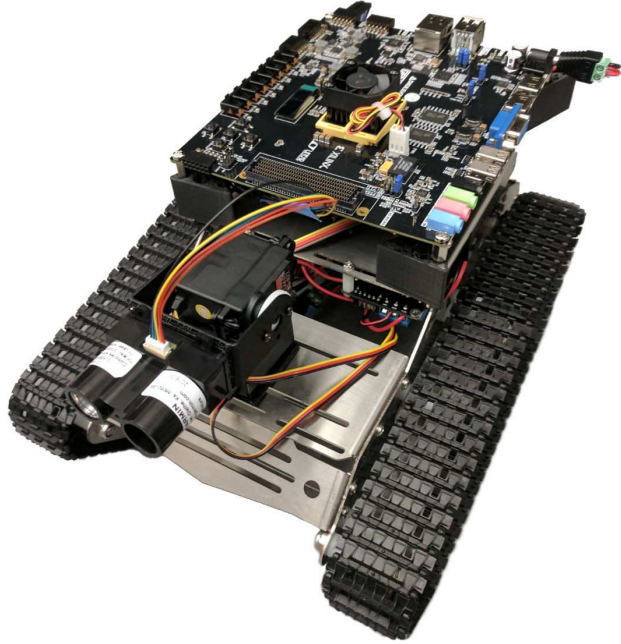


Figure 3.3: NeoN [21]

Kintex-7 FPGA, limit switches to detect edges, and a LIDAR affixed to a servo in order to sweep 120 degrees and take measurements. An image of NeoN may be seen in [Figure 3.3](#).

NeoN utilized a Microblaze core implemented in the FPGA to act as a central processing unit. This core connected to peripheral interfaces, memory, and a DMA engine used to communicate with the neuromorphic processor. For the neuromorphic processor, they chose to use the first generation Dynamic Adaptive Neural Network Array (DANNA). An overview of the architecture used by NeoN can be seen in [Figure 3.4](#).

A simulated model of the DANNA neuromorphic processor was utilized within the TENNLab framework. By leveraging this, an application was written that would model the robot’s sensors and movements within an environment defined by obstacles, walls, and ledges. This application was used along with the DANNA neuromorphic processor and the EONS learning algorithm to train a network to be implemented on the robot. The training that produced the optimal network was trained on 18,000 nodes on ORNL’s Titan supercomputer [3].

The resulting optimal network was observed to turn away from obstacles earlier than it needed to, but otherwise, it seemed to perform quite well in a variety of rooms on the

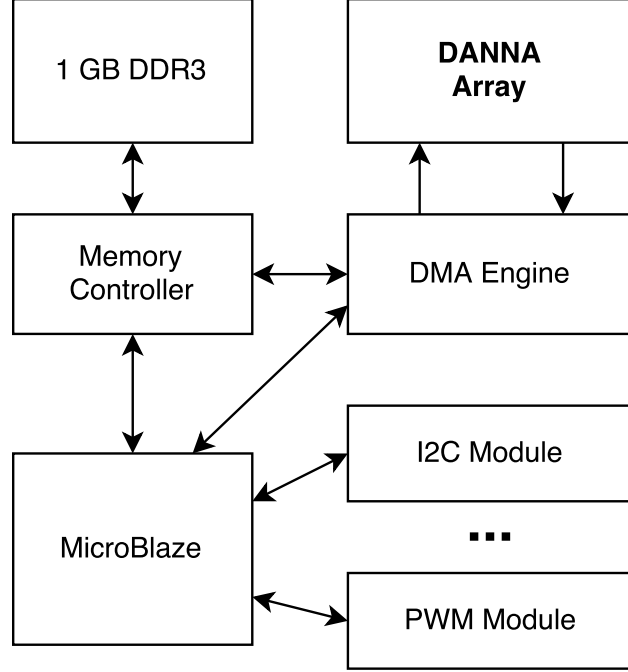


Figure 3.4: NeoN digital architecture [21]

physical system. It was found to rarely get caught in loops of decisions that the system did not eventually escape. The primary discrepancy observed between simulation and the physical system was with low power motor responses, but it was presented that these could be mitigated through the use of a Proportional-Integrated-Derivative (PID) controller, such that the speeds that occurred in simulation could be exactly matched on the physical system [21].

3.4 DANNA2

The second generation Dynamic Adaptive Neural Network Array (DANNA2) is among the neuromorphic processors with support in the TENNLab framework [20]. This processor is a relatively generic model with a focus on hardware implementations. A space-efficient design of the processor, made to target an FPGA, is available in VHDL, with an AXI4-Stream interface for communication support.

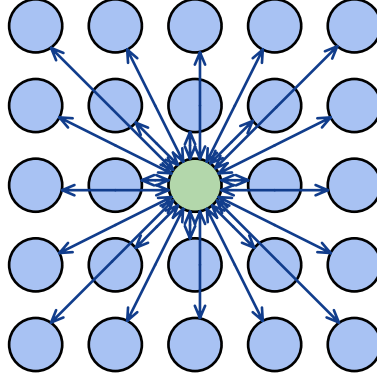


Figure 3.5: DANNNA2 connectivity pattern

3.4.1 Architecture

Architecturally, a DANNNA2 array is made up of a two-dimensional grid of elements. A singular element is composed of a leaky integrate-and-fire neuron with 24 potential synaptic connections. These synaptic connections reach to the 24 nearest neighbors of the given element as depicted in [Figure 3.5](#). An element outputs a binary spike that is then carried to elements that have connected synapses, with the weight corresponding to each synapse being the weight fired into the connected element. An additional feature of the arrays allows elements to be treated as “fan-in” elements that simply work as a passthrough with added delay that can extend the reach of an element. The structure of an element may be seen in [Figure 3.6](#).

The inputs to a DANNNA2 array exist within the elements at the leftmost edge of the array and the outputs exist within the elements at the rightmost edge of the array. Output spikes from the array are binary, but the input spikes have four bits with which they may encode a fire weight representative of the input’s synaptic strength.

The array is configured to run with a 100 MHz global clock. Calculations for a given array cycle are completed every 10 global cycles, such that the effective array cycle frequency is 10 MHz. The speed at which these arrays are capable of running is substantially faster when compared to other neuromorphic systems, executing at a rate of more than two orders of magnitude faster [17].

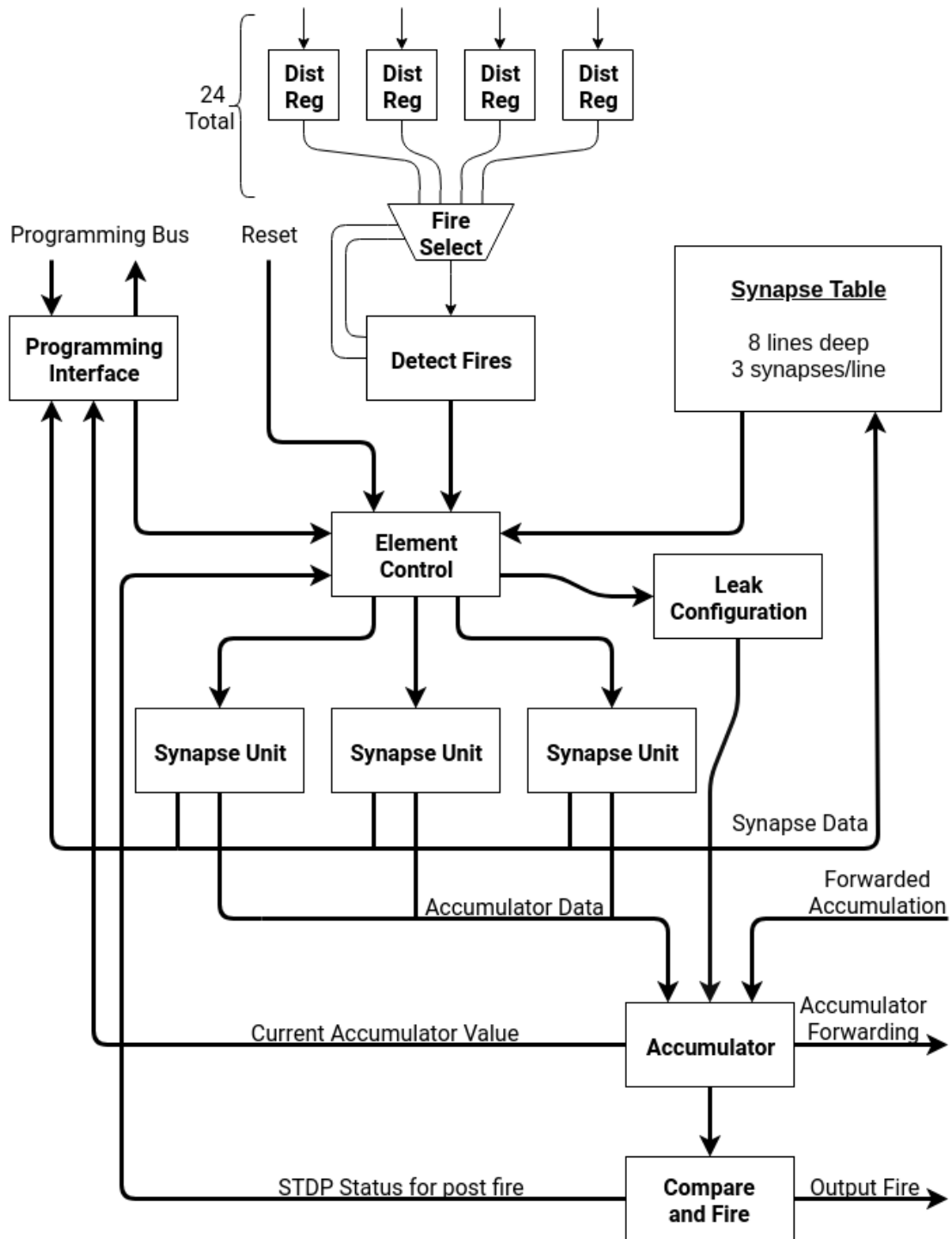


Figure 3.6: DANNA2 element block diagram [20]

Additionally, the hardware implementation of the DANNA2 array has support for Spike-Timing-Dependent Plasticity (STDP) and leak [40, 37]. These features allow the processor to perform online learning, in the form of STDP, as well as short-term operation memory management, in the form of leak. These arrays also feature a communication structure in the form of 512-bit packets. These packets may represent fire commands to fire a given input, simulate commands to advance the network a given number of time steps, reset commands to reset accumulated charge in the neurons, or configuration commands to set the internal state of a grid array. The structure of the configuration packets may be seen in Figure 3.7. The structure of the simulate and fire commands may be seen in Figure 3.8. The structure of the reset packets may be seen in Figure 3.9. Lastly, the array returns output packets of the structure seen in Figure 3.10.

3.4.2 Network Array Types

DANNA2 arrays exist in two varieties: sparse and grid. Sparse arrays have no connectivity restrictions, such that they do not have to follow the connectivity pattern outlined above. Additionally, they are implemented as special entities in the hardware implementation of DANNA2 arrays. When using these, a script must convert an array to the VHDL code that represents it in order for the hardware to be synthesized. These arrays would ideally be useful for an ASIC, such that a chip may be fabricated for the purpose of solving a specific task; however, grid arrays offer a lot more flexibility in terms of their usability.

Grid arrays function by generically creating a full, two-dimensional grid of elements. These elements are connected together with a communication chain that allows packets of configuration data to be sent in order to set the connections, thresholds, and synaptic weights within the network. Through this communication chain, the grid array offers reconfigurability in real-time, such that new configuration packets may be sent and used to set the hardware to function entirely different than it previously had [22].

0	3	7	11	15	19	23	27	31
Packet Type								
Element Address (32 bits)								
Threshold						F	R	L S
Weight 0			Delay		Weight 1			Delay
Weight 2			Delay		Weight 3			Delay
Weight 4			Delay		Weight 5			Delay
Weight 6			Delay		Weight 7			Delay
Weight 8			Delay		Weight 9			Delay
Weight 10			Delay		Weight 11			Delay
Weight 12			Delay		Weight 13			Delay
Weight 14			Delay		Weight 15			Delay
Weight 16			Delay		Weight 17			Delay
Weight 18			Delay		Weight 19			Delay
Weight 20			Delay		Weight 21			Delay
Weight 22			Delay		Weight 23			Delay

Figure 3.7: Configuration Packet Description. The ‘F’ bit signifies if the element is a fan-in element. The ‘R’ bits set the refractory period. The ‘L’ bits specify the charge leak. The ‘S’ bit enables or disables STDP.

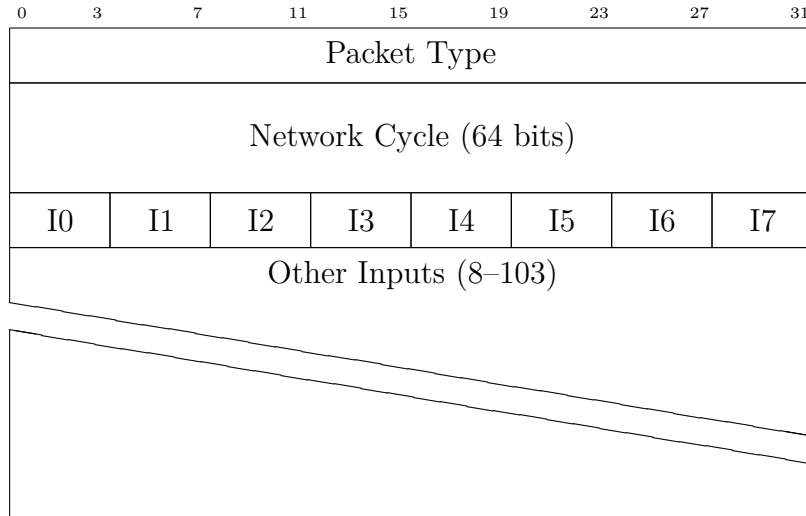


Figure 3.8: Step and Fire Packet Description

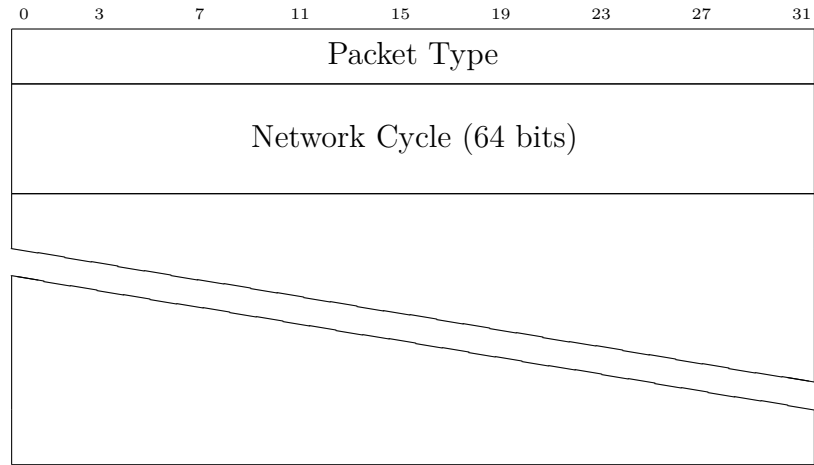


Figure 3.9: Reset Packet Description

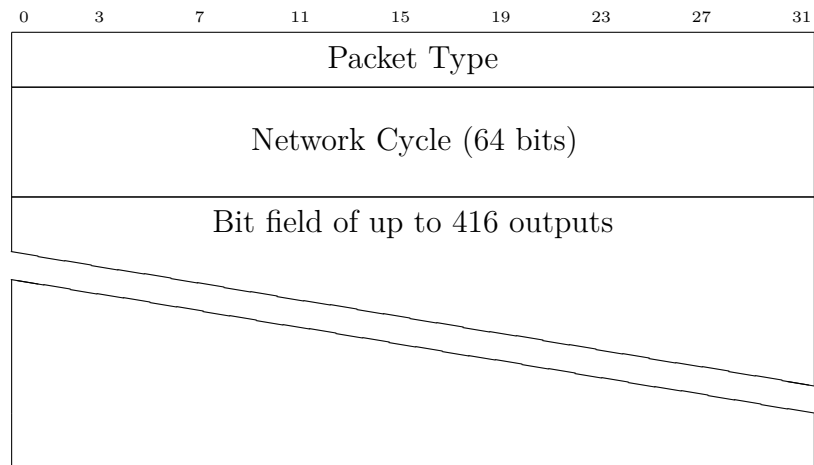


Figure 3.10: Output Fire Packet Description

Chapter 4

Objective for GRANT

Minds are like parachutes. They only function when they are open.

– James Dewar

The robot designed in this project had the intention of meeting a variety of purposes. The first intention of GRANT was to be a natural successor to NeoN. Through the use of an improved set of sensors along with better build components, GRANT was intended to overcome some of the physical limitations experienced by NeoN. Furthermore, it was intended that this robot would be the first hardware application of the DANNA2 neuromorphic architecture. Using DANNA2 over the DANNA architecture would ideally allow faster network speed and a greatly improved density in terms of the number of potential neurons and synapses.

As far as the application of this robot is concerned, the objective for this robot was to surpass the capabilities of NeoN, adding the additional functionality of targeting an object in a given space. This would mean that not only could GRANT perform grid coverage while avoiding obstacles, but the robot could also find a specific object in that field and navigate to it. This objective adds a whole new layer of complexity that was not required of NeoN. In order to combat this additional complexity, the designed intention was for GRANT to utilize independent networks for grid coverage and targeting. The robot, dependent upon the scenario, would choose which network to have configured in hardware, such that the desired function could be performed. In this way, the robot could perform grid coverage until it found the target and then utilize the targeting network in order to navigate to that target while still avoiding obstacles.

Chapter 5

Hardware Design

The robots of the cartoons and movies from the 1970s are going to be the reality of the 2020s.

– Alec Ross

In this chapter, we detail the hardware used on GRANT and the component selection process. A high-level diagram of GRANT, including only the most essential components, may be seen in [Figure 5.1](#). A fully assembled GRANT may be seen in [Figure 5.2](#). An internal view of GRANT may be seen in [Figure 5.3](#). A detailed wiring schematic may be seen in [Appendix A](#) in [Figure A.1](#). References for all of the parts discussed in this section can be found in the bill of materials in [Appendix B](#) where additional information about each of them may be found.

5.1 Control System

In the design of GRANT, many objectives had to be taken into account. First and foremost, GRANT was conceptualized to be the first embedded platform using the DANNA2 neuromorphic processor, and, as such, needed to include an FPGA in order to incorporate this component directly into the hardware. The Xilinx PYNQ-Z1 that incorporates a Xilinx ZYNQ-7000 SoC was selected for this purpose, as this board would allow the DANNA2 processor to run alongside an ARM processor [\[31\]](#). This ARM processor would allow the inclusion of sensors with pre-existing libraries to be fully utilized. It also offers the ability

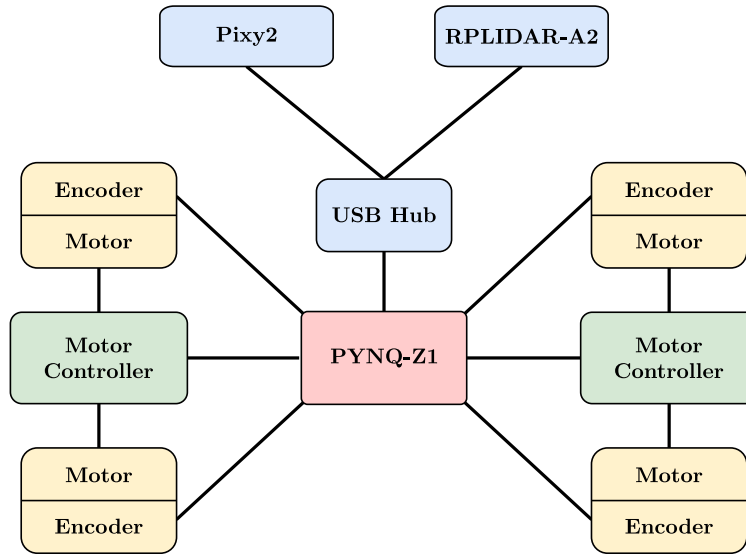


Figure 5.1: High-level diagram of GRANT hardware design

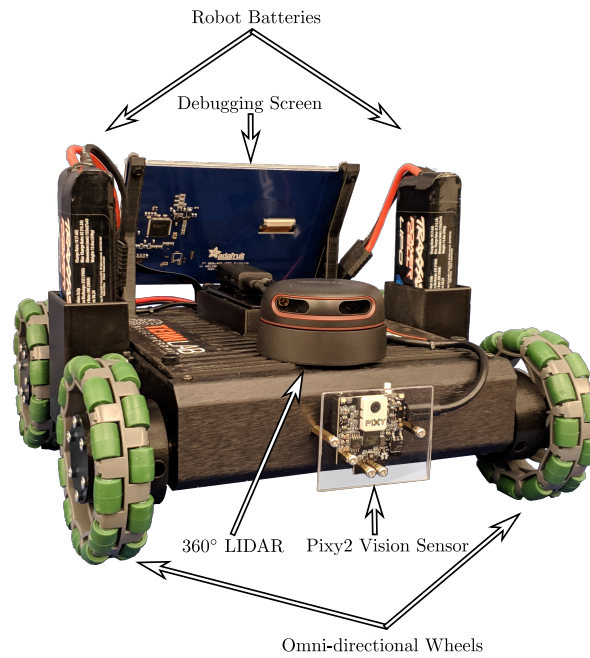


Figure 5.2: Front view of an assembled GRANT robot

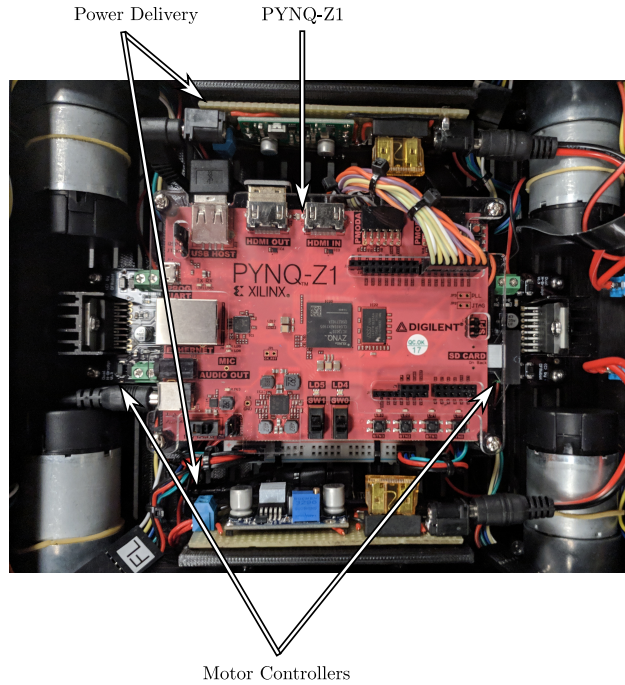


Figure 5.3: Internal view of an assembled GRANT robot

to encode inputs for sending to the DANNA2 processor as well as decoding the output of the processor to make decisions and exhibit control over the robot. It furthermore would serve to emphasize the ability of EONS to train very small RSNNs, as the PYNQ-Z1 is a resource-constrained platform, which only allows about 150 DANNA2 elements to fit alongside the processing system fabric on the FPGA component of the ZYNQ-7000.

The next step in the design process was sensor selection. The first desired component was a LIDAR that was to be used for obstacle avoidance tasks. The RPLIDAR A2 [33] was used as a vast improvement to the original LIDAR used on NeoN [21]. Not only did it allow a 360 degree range of motion, but it also ran at a maximum frequency of 15 Hz. This increased frequency would, in theory, allow the network to better make decisions as it received new information more often. The RPLIDAR A2 operates by pulsing a laser and measuring the time it takes to receive the reflected response. This operation can be performed up to 8000 times per rotation, allowing very fine granularity and virtually removing blindspots. Since it was desired, however, that the robot never move backwards, only the front 180 degrees



Figure 5.4: Example Pixy2 Target Recognition

gathered from the sensor were used. This allows the rear 180 degrees to be obstructed by other components of the robot.

For the purpose of identifying a given target, a Pixy2 vision sensor was used [26]. The Pixy2 utilizes on-board vision processing with colored connected component analysis. In essence, it identifies patches of color with allowable settings for threshold. An example of this color analysis can be seen in Figure 5.4 where the orange cone's bounding box is identified by the camera. One can see that the cone is recognized without falsely identifying the other orange object. Also, to account for various lighting conditions, the Pixy2 has built-in LEDs that allow the lighting level to be near constant for the purposes of recognizing an object.

It was decided that by using this camera, one would be able to discern a distinguishable target from a field of obstacles without having to perform vision processing directly on the PYNQ-Z1.

Since each of the above sensors can utilize USB for communication and the PYNQ-Z1 has a host USB port, it was decided that a USB hub would be used to connect these sensors. Furthermore, it was decided that a powered USB hub would be used for the peripherals to add increased stability, as it was found that the PYNQ-Z1 could not handle the current demands of these two sensors in tandem. In addition to this, a seven inch HDMI screen was



Figure 5.5: Chassis used for GRANT

added to the robot for the purpose of debugging directly on the robot. A wireless keyboard was also used for sending commands to the robot, and this was also connected to the USB hub via a wireless dongle.

For controlling the motors for the robot’s drivetrain, dual channel motor controllers were used. These motor controllers offered a small footprint while still being able to independently control each of the motors on the robot through Pulse-Width Modulation (PWM). By using two of these, the entire drivetrain could be controlled.

5.2 Chassis

To house all of these components, it was decided that a lightweight, aluminum frame made by Lynxmotion would be used [16]. This frame, measuring eight inches wide, 10 inches long, and two and one half inches tall, may be seen in [Figure 5.5](#). This chassis offered adequate room for all of the components necessary for the robot. It also offered a myriad of mounting options for motors, greatly reducing the required time to assemble the robot. A set of omni-directional wheels were also selected for use with this robot to allow for effortless, in-place turning.

Motors were selected that would adequately meet the demand of bearing the load of all of these components while still being relatively fast. These motors also had to be small such that they would take up a minimal amount of room in the interior of the robot chassis. The selected motors, made by Pololu, offered the desired speed and torque requirements for this application while still maintaining a small form factor [29]. These motors also featured

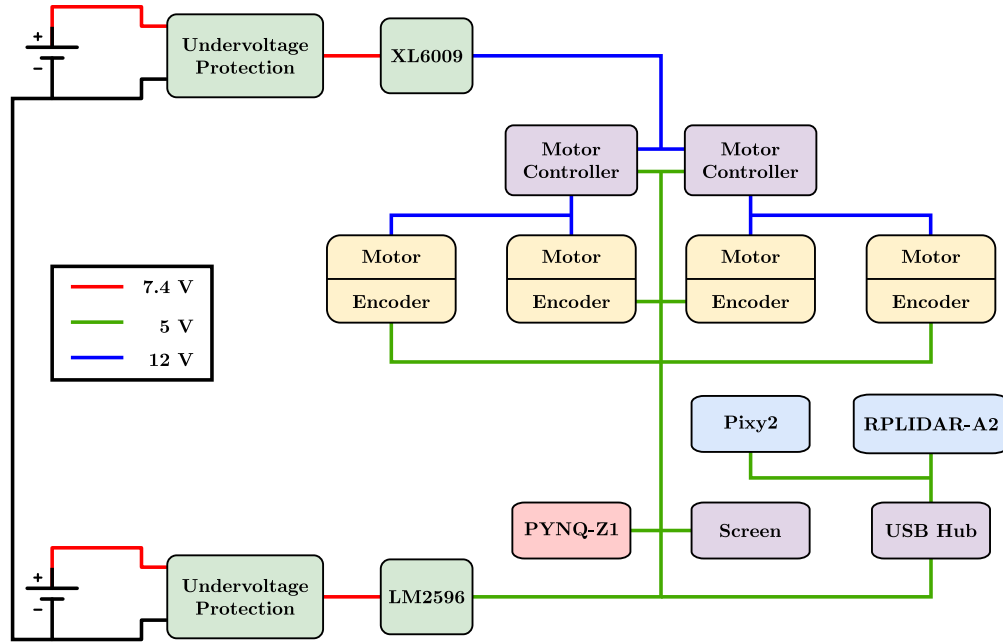


Figure 5.6: Power distribution on GRANT

integrated quadrature encoders with 1920 counts per revolution of the output shaft. This feature was necessary, as it would allow for the robot to keep track of its position change. The relevancy of this feature will be discussed in [Chapter 7](#).

5.3 Power Delivery

For the robot, it was known that the controls system would be ran at five volts while the motors needed to operate at 12 volts, as can be seen in [Figure 5.6](#). To accomplish this, it was decided that two power systems would be used with only a shared ground between them. Also, to stabilize voltage, it was decided that regulators should be used, as they can maintain a steady output voltage as the input voltage from the batteries decreases. For batteries, 7.4 volt RC car batteries, made by Traxxas, were used [38]. These batteries offered a large capacity of 7400 maH while maintaining a small form factor. For power regulation, a boost converter using the XL6009 chip and a buck converter using the LM2596 chip were used to raise and lower the battery voltage, respectively.

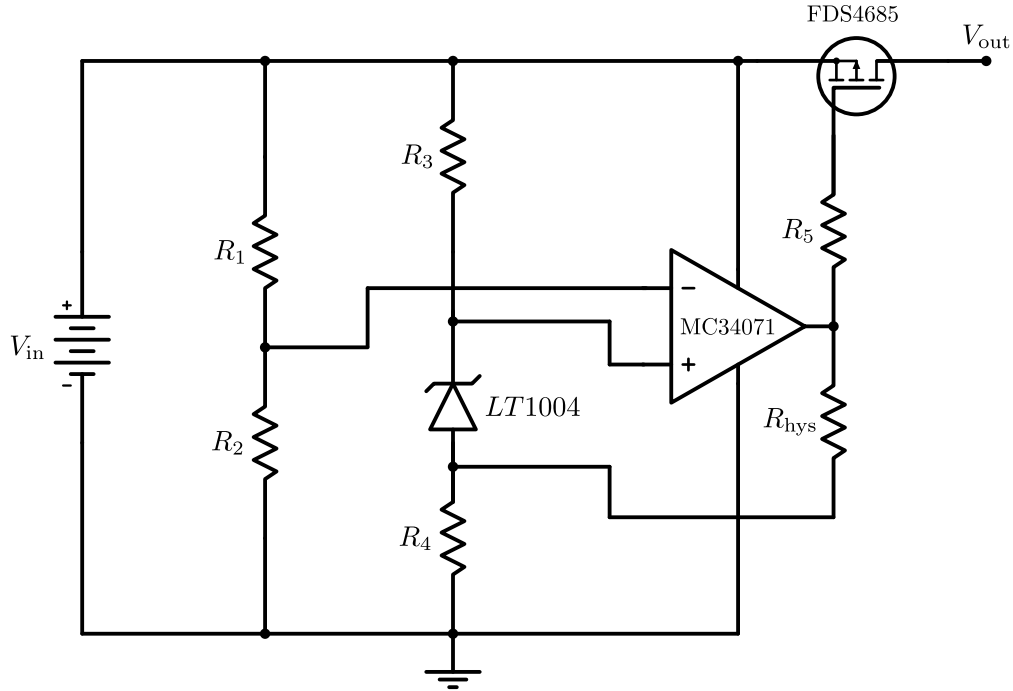


Figure 5.7: Circuit diagram of undervoltage circuit

The voltage regulators were affixed to Printed Circuit Boards (PCBs) that included a small undervoltage protection circuit as well as fuse protection. This undervoltage protection functions by using a micro-power voltage reference with a voltage feedback amplifier and with voltage division to enable/disable a PMOS that allows the input voltage through. Furthermore, using a feedback resistor, hysteresis is implemented, such that once the circuit stops allowing power through, a voltage higher than the cutoff voltage must be applied to allow the power through again. A circuit diagram for the undervoltage protection circuit may be seen in [Figure 5.7](#).

5.4 3D-Printed Parts

Additional parts that were not purchasable were needed for the construction of this robot. These parts would, instead, have to be designed and 3D-printed. The first part designed was a simple battery holder that could be affixed to the sides of the chassis. These would allow

the batteries to be easily accessible and held into place. The final design of this part can be seen in [Figure 5.8](#).

The next part designed was a bottom panel for the chassis. This panel was responsible for holding the motor controllers, PCBs for power delivery, PYNQ-Z1, and any relevant components for power distribution. The panel had slits throughout it to allow for neat cable management and for air flow to cool the components. The final design of this part, unpopulated, can be seen in [Figure 5.9](#), while the populated version of this part can be seen in [Figure 5.10](#).

The third part designed was the top panel for the chassis. This panel was responsible for holding the screen, USB hub, and RPLIDAR A2. Like the bottom panel, the top panel also had slits to allow for neat cable management as well as air flow for the components in the interior of the robot. The final design of this part, unpopulated, can be seen in [Figure 5.11](#) while the populated version of this part can be seen in [Figure 5.12](#).

The final two parts designed were for a wheel adapter for the omni-directional wheels to properly affix to the output shaft of the motors. The wheels were designed with 0.125 inch bar shafts while the motors had a 6 millimeter D-shaft for output. To affix to the motors, one part offers a round hole with two set screws separated by 90 degrees. To attach to the wheels, the two parts attach to one another via screws, sandwiching the spokes of the wheel in between. All the screws are affixed by embedding nylon stop nuts into the parts during printing. The final design of these parts can be seen in [Figures 5.13](#) and [5.14](#).

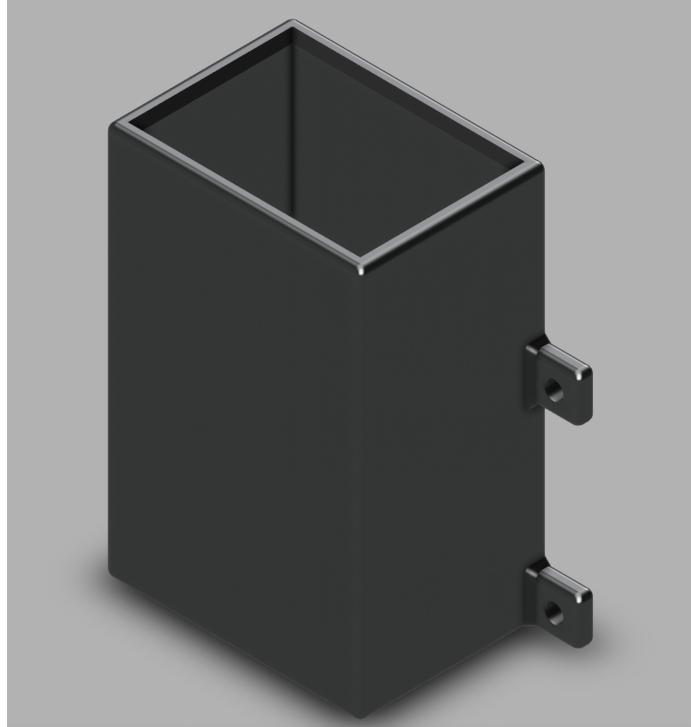


Figure 5.8: Battery holder CAD model

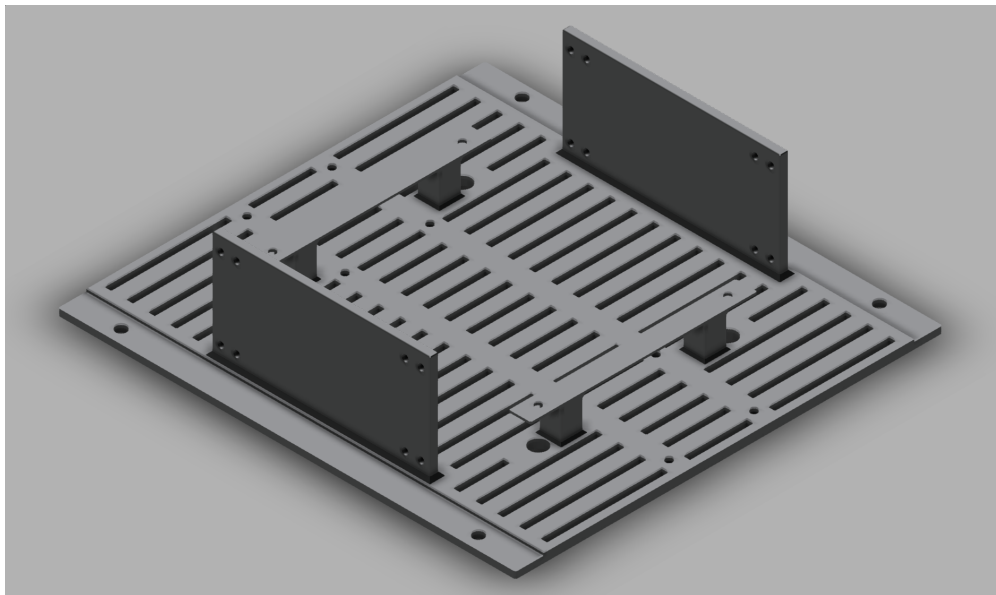


Figure 5.9: Bottom panel CAD model without components

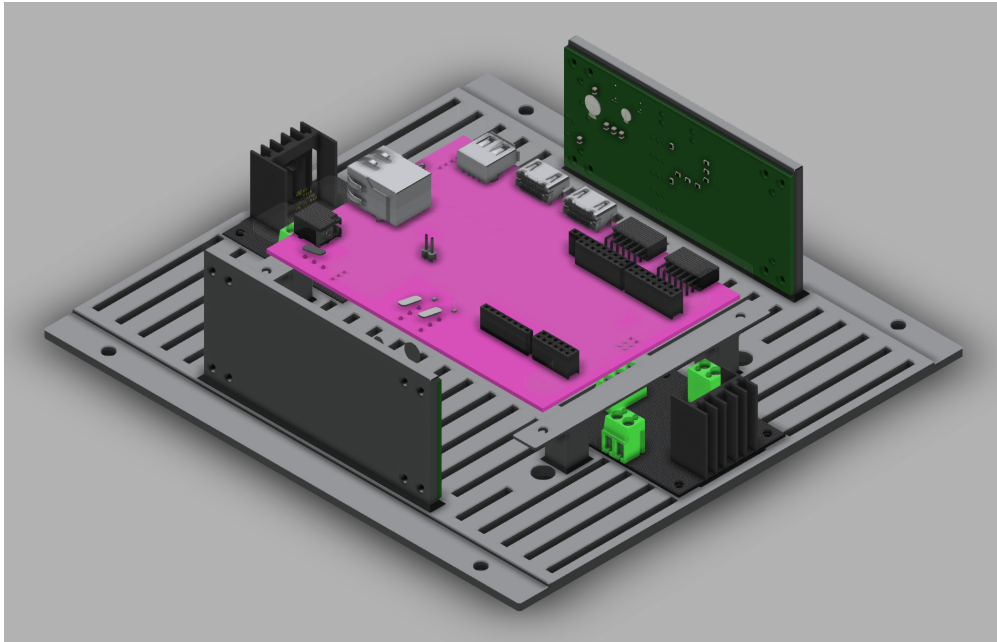


Figure 5.10: Bottom panel CAD model with components

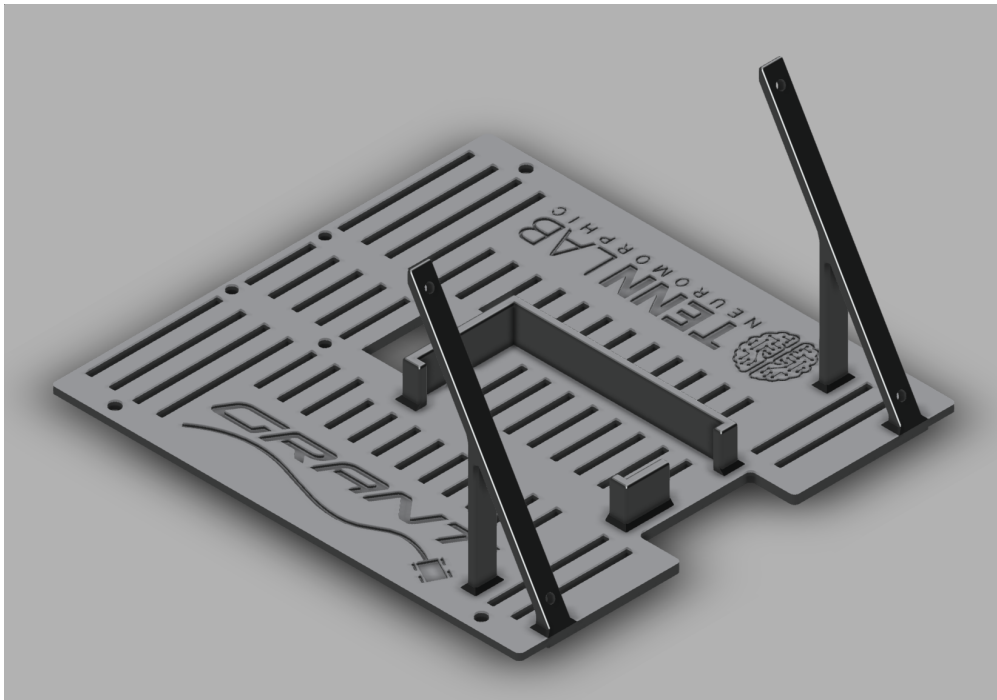


Figure 5.11: Top panel CAD model without components

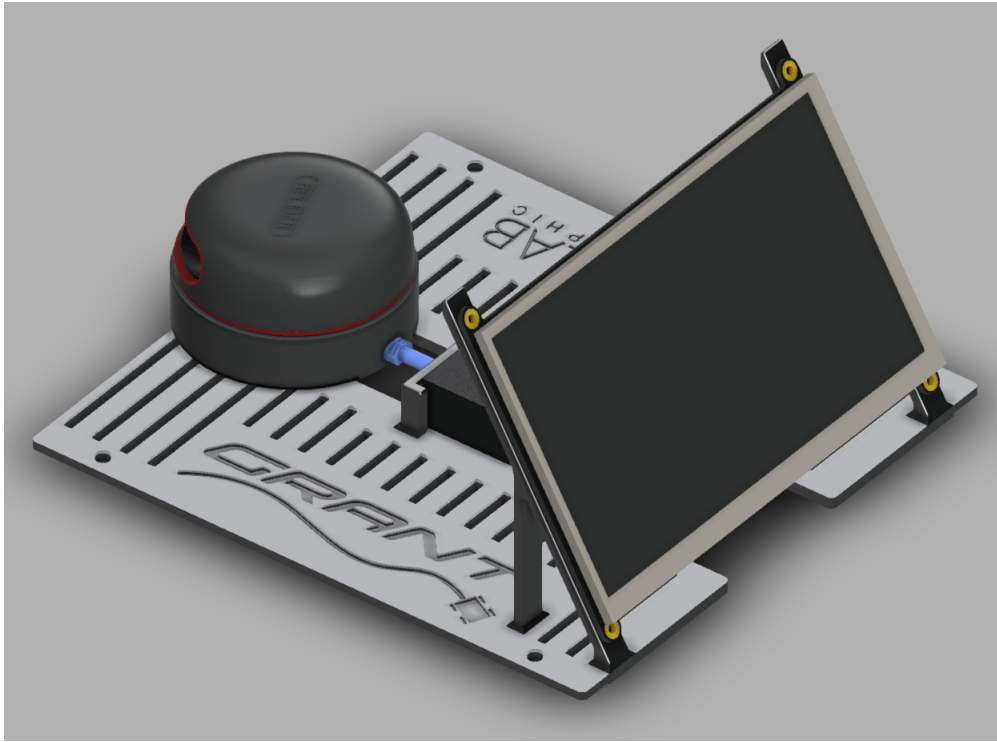


Figure 5.12: Top panel CAD model with components

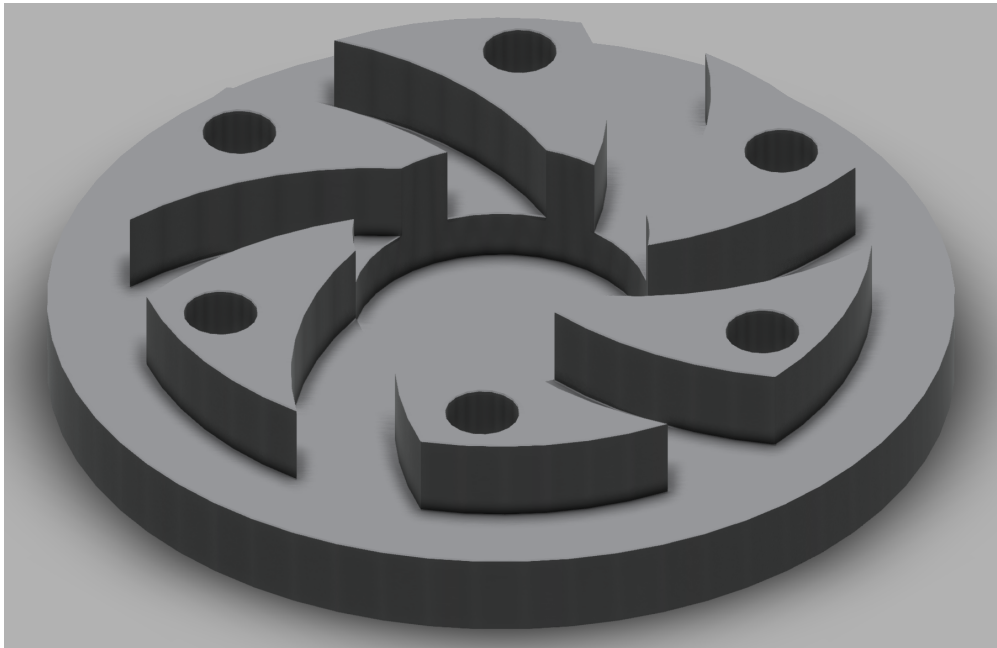


Figure 5.13: Outside piece of wheel adapter

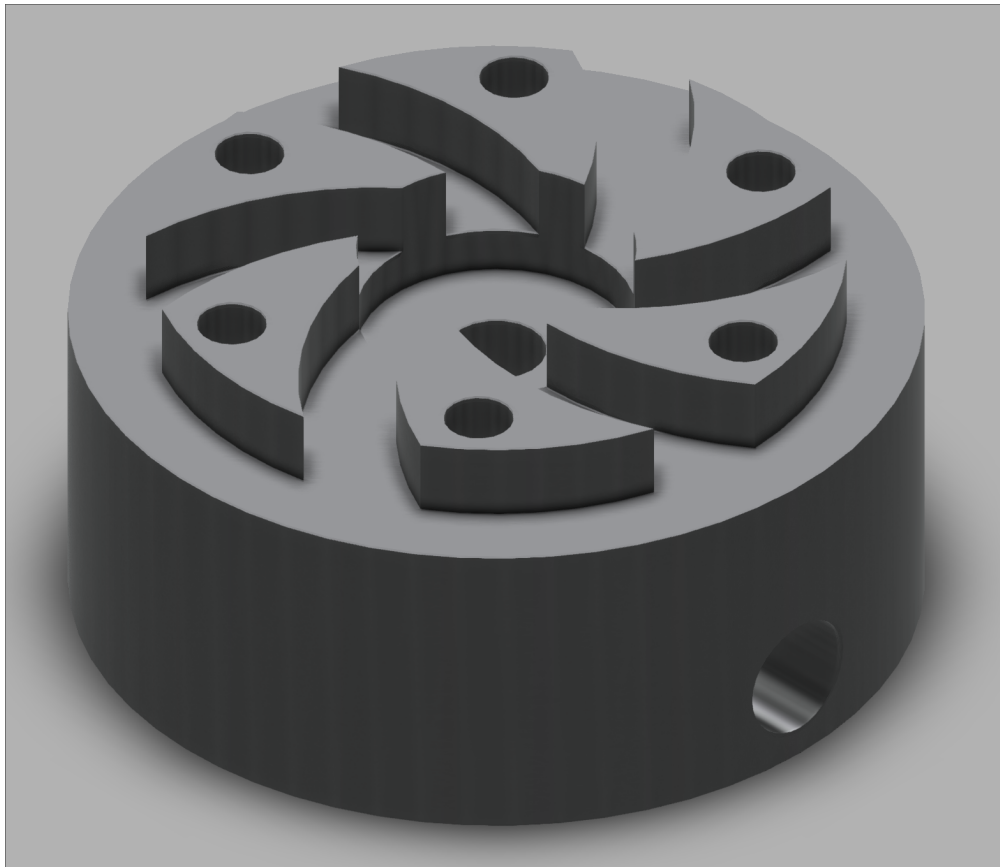


Figure 5.14: Inside piece of wheel adapter

Chapter 6

Digital Design

People who are really serious about software should make their own hardware.

– Alan Kay

In the design of GRANT, it was decided that it would be desirable to fully utilize available peripherals and capabilities of the PYNQ-Z1 as well as the USB libraries associated with the sensors. In order to do this, it was decided that a custom operating system built using Petalinux would be run on the SoC, as this would offer a Linux environment capable of interfacing directly with the FPGA of the PYNQ-Z1. Petalinux will be discussed in greater detail in [Chapter 7](#). A high-level diagram of the architectural structure of GRANT can be seen in [Figure 6.1](#).

6.1 Base Robot Project

To properly interface with the custom operating system, a base hardware design project, implemented as a Vivado IP Project, for the PYNQ-Z1 was used that connected the ARM core to the various peripherals correctly [4]. This base project utilized an AXI4 interconnect for communication between the processor and the peripherals. It included the proper components for interfacing with USB, HDMI-in, HDMI-out, as well as interfacing to the buttons, switches, and LEDs included on the System on Chip (SoC) via General Purpose Input/Output (GPIO).

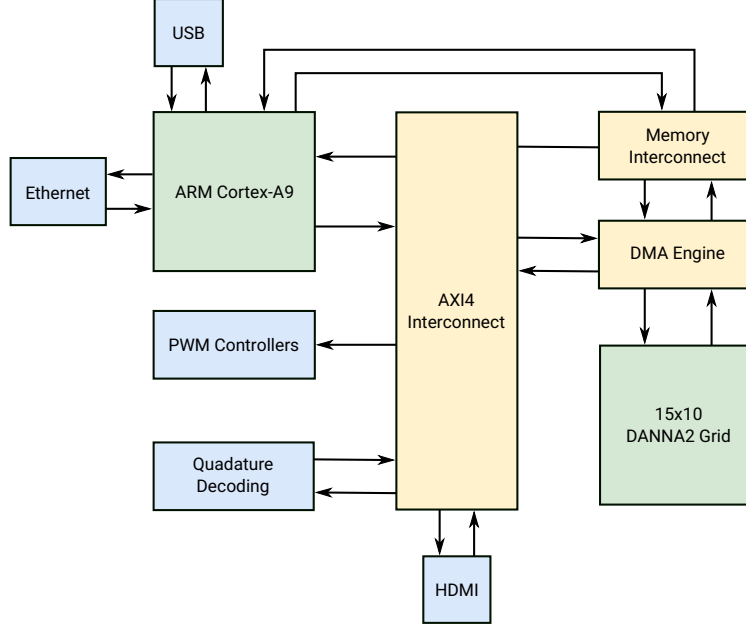


Figure 6.1: GRANT digital design block diagram

This base project offered a great starting point for modifications specific to GRANT to be built upon.

6.2 Modifications for Robot Control

The first modification made to this base project was the addition of hardware PWM control for the motors' speed controllers. Utilizing the PWM IP block and connecting it to the interconnect via a GPIO IP block, the speed of a given motor could be directly controlled from the processor. Four of these were implemented, with one being used for each motor, and the outputs for each of these PWM IP blocks was configured to go to an I/O pin on the PYNQ-Z1. Furthermore, the direction pins on each of the speed controllers were attached to the interconnect via another GPIO block. The outputs of this GPIO block were tied to I/O pins on the PYNQ-Z1, thus allowing the processor to not only control the speed of each motor, but the direction, as well.

The next modification to the base project was the addition of components for quadrature decoding. It was found that Digi-Key provides basic VHDL code for quadrature decoding [10].

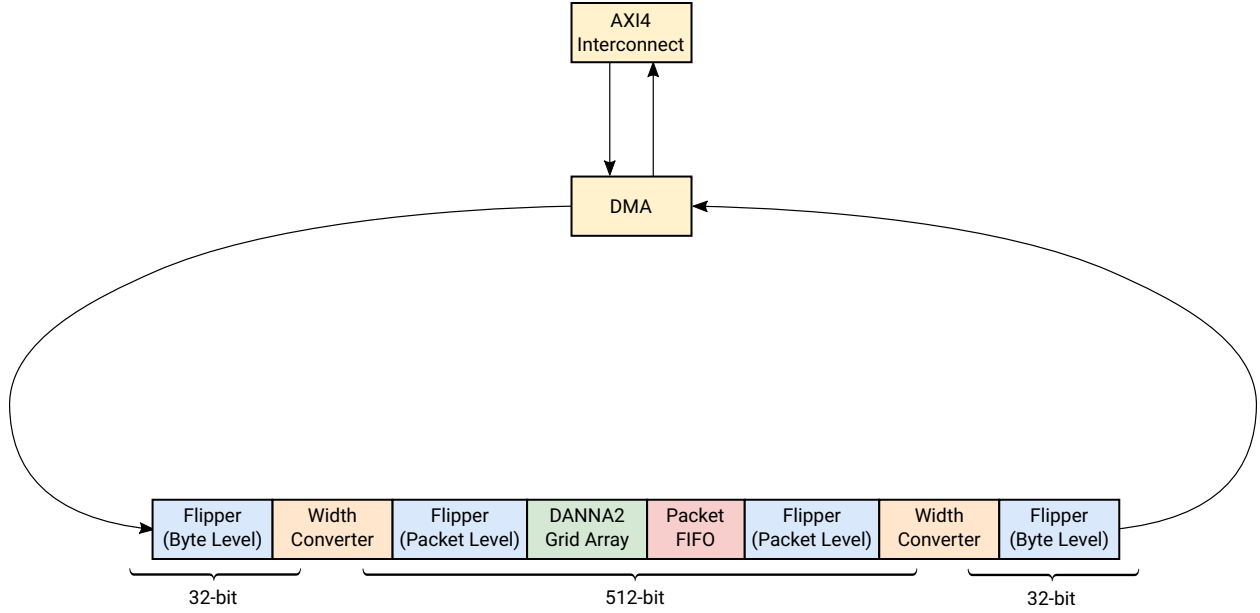


Figure 6.2: Architecture adding the DANNA2 processor

This code was modified to output a 32-bit integer for the current count from the encoders as well as a bit to represent the last direction that the motors turned. One of these modified quadrature decoding blocks was connected to the interconnect via a dual-channel GPIO block, where one of the channels was utilized for the 32-bit count while the other channel was used for the last turning direction. I/O pins on the PYNQ-Z1 were connected to the input of the quadrature decoding block, such that the count of a given encoder could be read directly by the processor. Four of these structures were created, with one for each of the encoders.

In order to maximize the FPGA utilization and the size of the DANNA2 grid array that could be placed on the SoC, extraneous parts that existed in the base hardware design project had to be removed. These removed components were primarily associated with the HDMI-in port of the SoC. The logic required to operate this port was quite significant, and the removal of it freed approximately seven percent of the FPGA’s resources.

6.3 Addition of Neuromorphic Processor

A detailed diagram showing the configuration of the surrounding architecture to add the neuromorphic processor can be seen in [Figure 6.2](#).

As previously mentioned, the DANNA2 array supports the AXI4-Stream protocol of communication for network input and output. As such, it was decided that the best way to send and receive packets to the network was through a Direct Memory Access (DMA). An AXI4-Stream DMA was added to the hardware project and directly connected to the interconnect. Unfortunately, the maximum supported bus width of the DMA was 32 bits while the DANNA2 packet size is 512 bits. This meant that the individual 32-bit packets would have to be stitched together with an AXI4-Stream width converter.

AXI4-Stream width converters were placed on both the output and input of the DMA. One of these width converters was responsible for gathering 32-bit packets from the DMA and stitching them together to form a 512-bit packet for network input. The other was responsible for splitting the 512-bit packets from the network into the proper 32-bit packets to send back to the DMA. It was found, however, that the bits from the DMA were not sent or received in the expected order. To combat this, a module had to be designed that could properly reorder the bits to the expected order for the network. The module was designed generically, such that the same module could be used to flip the DMA output within a packet that went into the width converters and one could be used on the output of the width converter to reorder the 32-bit packets that were sent in the wrong order from the DMA. Similarly, these modules were used in the same configuration on the output of the network and on the input back to the DMA. Configuring them in this way allowed the processing system to send and receive packets in the expected format, and doing these conversions in hardware allowed simplicity in the design of the software.

A 15x10 DANNA2 grid array was added in the middle of this chain as the neuromorphic processor [20]. This array was configured to have STDP, leak, and fan-in elements disabled. Disabling these features was the only way to fit this many elements. The network was given a 100 MHz clock, allowing time steps within the network to be completed at a rate of 10 MHz. Due to the sheer speed that the network was capable of running, a First-In-First-Out (FIFO) queue was added to the output of the network such that the network speed would not be bottlenecked by the rate at which output packets were consumed.

Chapter 7

Software

Computers will overtake humans with AI within the next 100 years. When that happens, we need to make sure the computers have goals aligned with ours.

– Stephen Hawking

7.1 Petalinux

Petalinux is a program for configuring a custom operating system designed for SoCs designed by Xilinx and built upon the Linux kernel [25]. This tool was used as it offered a simplistic way of integrating the given peripherals and the FPGA of the PYNQ-Z1 SoC into a singular environment. In addition, since it is built upon the Linux kernel, common software libraries can be built and used within the operating system. This gave liberty to use the sensors' libraries instead of having to port these into some new environment.

A base project provided for the PYNQ-Z1 was used as a starting point for this project [23]. The Petalinux utility allows customization of installed software, and since this is provided, a few pieces of software were installed in order to ease the development process. The most important software addition was undoubtedly Valgrind, as this program allowed the tracing of the raw memory that was being manipulated to interface with the DMA [39]. Furthermore, the Petalinux utility allows customization of the device tree, such that all of the additional GPIO and the DMA could be accessible from within the operating system itself. Using the

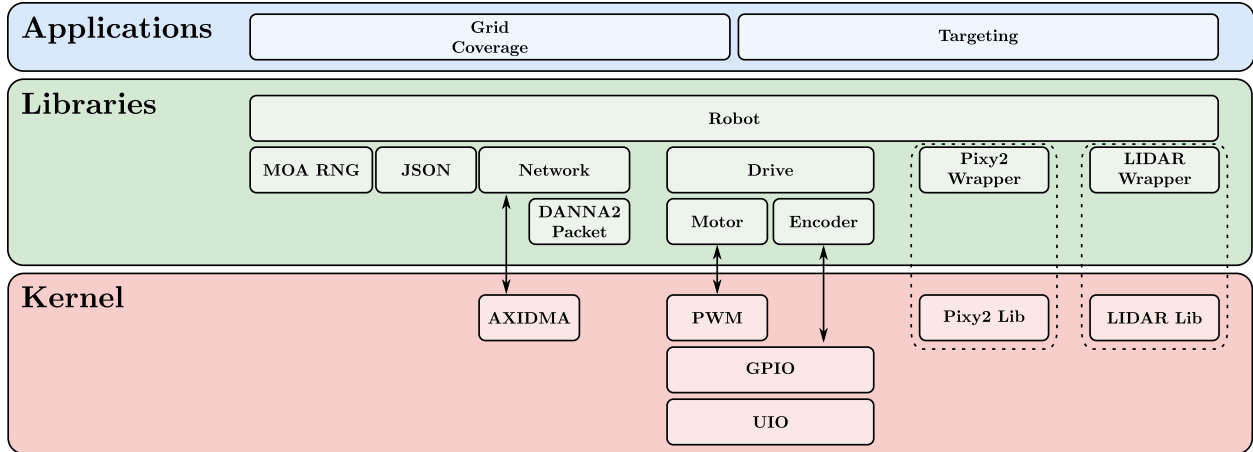


Figure 7.1: Structure of the software stack

exported hardware definitions from the project discussed in [Chapter 6](#), the device tree was configured so all of these devices were recognized.

7.2 Robot Programming

For the implementation of the physical system, a considerable amount of software development had to be done. In order to simplify this implementation, existing libraries were used to integrate with the sensors and peripherals such that the algorithms could be developed independent of the exact sensors used. This would theoretically allow one to continue the work on this project with a different system utilizing the algorithms contained herein. In this section, one may gain a better understanding of the libraries used, as well as the algorithms developed to support this project and the simulator modifications to support network training. The general structure of the software stack may be seen in [Figure 7.1](#).

7.2.1 Existing Libraries

Three primary libraries were used that were included as a part of the base Petalinux project. These libraries allowed the user to access the GPIO blocks and control the PWM blocks that were a part of the digital design that interfaced with the processor. This allowed the

counts from the encoders to be read and the speed controllers for the motors to be properly controlled via PWM.

In order to control the sensors and access their readings, a USB library for each was used. In the case of the Pixy2, the library finds all objects that match a trained signature and returns both the position and size of each relative to the resolution of the camera [11]. For the RPLIDAR A2, the library returns readings from a full rotation of the sensor [8]. These readings include the angle of the reading and the distance read in millimeters.

A library for random number generation used in the TENNLab Framework was used for the purposes of random number generation for anything that required such tasks in the algorithms. This was used instead of builtin random number generation because this library had better distribution in testing [18].

A JSON library was also used to configure various parameters with respect to how the robot operates [15]. These parameters encompassed information as to how the robot encoded and decoded information with respect to the network, what was considered completion of the targeting task, and even how to adjust the speed of the sides of the robot to properly account for inconsistencies between the response of these sides. Using this library, the robot could be reconfigured at run-time instead of having to recompile or make a large number of command line arguments.

Another external library that was used allowed the processor to directly communicate with the DMA from userspace [24]. This was necessary as packets were generated given the sensor readings, sent to the neuromorphic processor via this library, and then the corresponding response packets were interpreted to make decisions for the response of the robot. The inclusion of this library greatly reduced the complexity of this communication process.

The final external library that was used served the purpose of creating and interpreting packets for communicating with a DANNA2 network [20]. Using this library, input packets can be created with an array of fire weights and a time. Programming packets can also be created that define the connections between elements and tell the network how to reconfigure itself. The library is also capable of interpreting output packets into an array of output fires. All of these capabilities eased the difficulty of being sure that communication with the DANNA2 array was being done properly.

7.2.2 Newly Created Libraries and Classes

The first important libraries that were created were for controlling the drivetrain. A library was written for the motors that allowed complete control over its functionality, including setting speed, enabling, and disabling. Similarly, a library was written for the interface from the GPIO to the encoders, such that one could read the count of the encoder, read the distance traveled by the wheel associated with that encoder, and reset the encoder such that one could measure displacement between two subsequent readings easily. Another library served to integrate the motors and encoders into a drive train, allowing it to be controlled as a tank would drive: that is, both motors on the right side are set to the same speed and both motors on the left are set to the same speed.

Additionally, wrapper classes were written around the Pixy2 library and the LIDAR library. For the Pixy2, this wrapper class would record if it had seen the target on a given time step, store the position of the largest object recognized as a target, and filter out objects deemed to be so small that they were likely artifacts. The filtering functionality was arguably the most important part of this, as it helped to filter out false identifications of a target. The wrapper class for the LIDAR was more complex, as this class did a number of tasks to preprocess the data. First, the class took the floats associated with the measurement angle and converted them to integers to know the value associated with a given angle more easily. If multiple values would correspond to a given measurement angle, the one that had a reading closer to the LIDAR became the stored value. Furthermore, if any measurement angles had no reading associated with them, these were sentinelized to be an object far away from the LIDAR. Lastly, this class could separate the measurement angles into bins as illustrated in [Figure 7.2](#), where each colored region represents a bin and the lines within these regions are the lines that are sampled. In this case, every measurement angle within a bin would be set to have the reading closest to the LIDAR within a bin. For example, if 50 degrees and 51 degrees were in the same bin, and the measurement at 50 degrees was for an object 10 meters from the robot and the measurement at 51 degrees was for an object one meter from the robot, the measurement for both of these angles would be set to one meter. This allowed increased resolution of the LIDAR while still only reading the measurements at individual

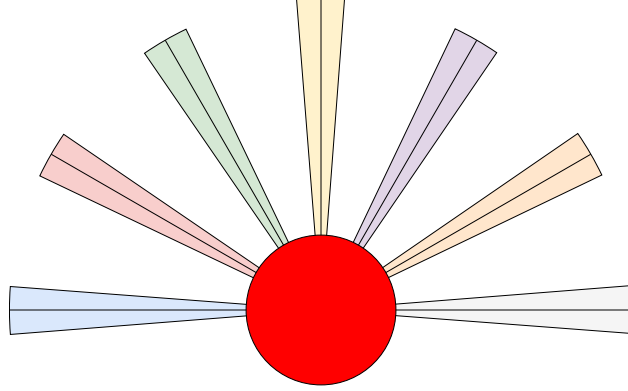


Figure 7.2: Example of LIDAR binning

angles. It also essentially allowed for more information to be encoded to the network and was capable of eliminating blind spots completely.

A class was also created that was for communicating with the network via DMA. This class utilized functions of the DANNA2 packet creating library and the DMA library that allowed communication from userspace in order to create a simplistic interface for loading networks, sending packets, and parsing output packets. Furthermore, this class also kept up with parameters like the current time step, the number of inputs and outputs for the network, and the size of the network, such that surrounding functionality could be written as generically as possible.

The final created class was for the entirety of the robot itself and utilized all of the libraries previously mentioned. This class included functionality to keep track of a target using only the measured change in the encoder counts of the four wheels, such that one could maintain the relative position of the target without seeing it. This class included the functions for encoding and decoding packets for communication with the network in the neuromorphic implementations as well as the supporting functions for the traditional algorithm implementation. Further details of these algorithms will be discussed in the following subsections.

7.2.3 Common Algorithm Structure

Both the traditional algorithms and the neuromorphic algorithms share a common structure in the operation of a singular update cycle, in which input information and motor speeds are updated. First, a timer is checked to see if enough time has elapsed for a new update of the sensors. This is done because it was decided that there was benefit to allowing the neuromorphic algorithms to continue to make new decisions even with old sensor data, so this structure allows those decisions to be made without having to have new sensor information every cycle. After this check is made, sensor updates are performed if it is applicable. Then, if a sensor update has occurred and the target has not been seen in this update, the position of the target relative to the robot is updated by using the change in the encoder counts to determine the rotation and displacement of the robot between consecutive update cycles. Also a counter is incremented to signify that this is another sensor update cycle without seeing the target. Once this counter crosses a definable threshold, it is believed that the robot no longer knows where the target is at due to the compounded error from the encoders, and thus, the robot reverts to a state of roaming and a state transition is done for both algorithms. After this, the individual algorithms each handle how to interpret the sensor information and how to set the motor speeds in accordance with this if the robot has not reached the target. The robot is considered to have reached the target if the distance to the target is less than a configurable maximum value and the camera reports that it has seen the object that cycle, signifying that the target must be in front of the robot. Upon setting the motor speeds, the algorithm then may enforce a minimum time that must occur for an update cycle, such that if the execution time of the update cycle has taken less than this minimum, the algorithm will simply call a sleep command and sleep for the remainder of this minimum time. A flow chart of this logic is presented graphically in [Figure 7.3](#).

7.2.4 Traditional Algorithm Implementation

In order to provide a basis of comparison for the neuromorphic algorithms, as well as test correct functionality of the sensors, a traditional version of targeting and exploration algorithms were written. In the targeting state, the two contributing factors to how the robot

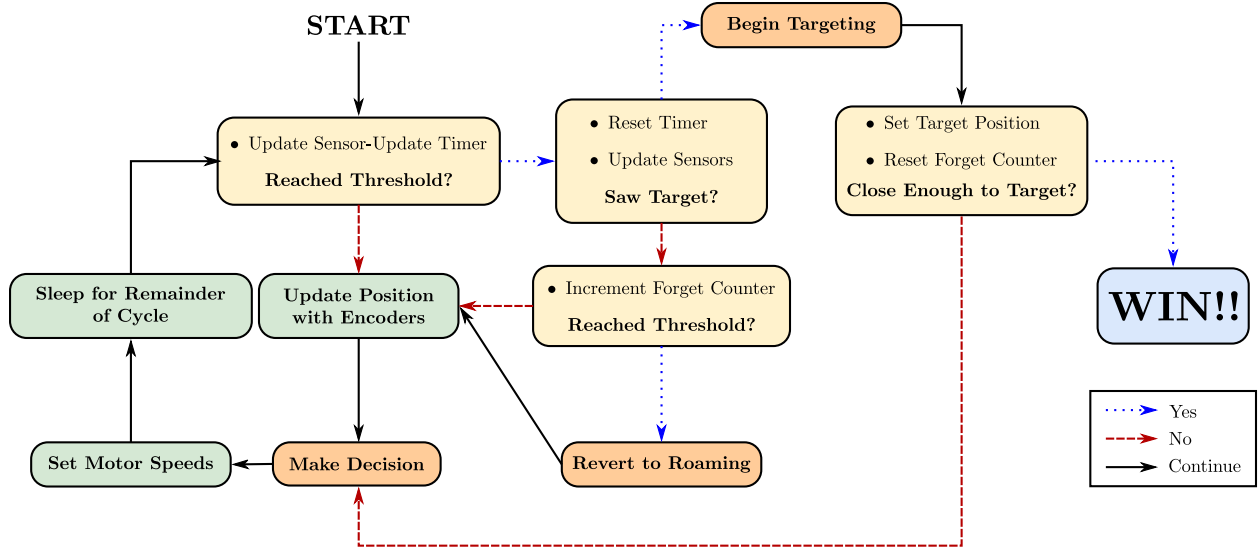


Figure 7.3: Generic Robot Algorithm Structure

will move are the position of the target and which way the robot has obstructions. The target is considered to be in front of the robot if it is within a configurable angular threshold with respect to the center of the front of the robot. Otherwise, the target is considered to be to the left or the right of the robot. The robot breaks obstructions into regions, with the right and left regions each being 90 degrees wide starting from the center of the front of the robot. The remaining region, the center, is the front 60 degrees of the robot. If an object is detected to be too close in a region, the robot is considered to be obstructed in that region. Based on these factors of obstructed regions and the direction to the target, a decision tree, illustrated in Figure 7.4, is formed to determine the response of the robot. In the event that a random turning direction is chosen, this turning direction is stored and is used for a configurable number of time steps in which a random turning direction is chosen before a new random direction is selected. This is done to encourage diversity in turning direction without causing a "wiggling" behavior due to successive different directions being chosen as the random turning direction.

Obviously, the robot cannot always know the position of the target, and, in fact, it starts out without knowing this position. In the event that it does not know where the target is, it must roam to find the target, and once the target is found by the camera, it can then use this gathered position to actually track it. To roam with the traditional algorithms,

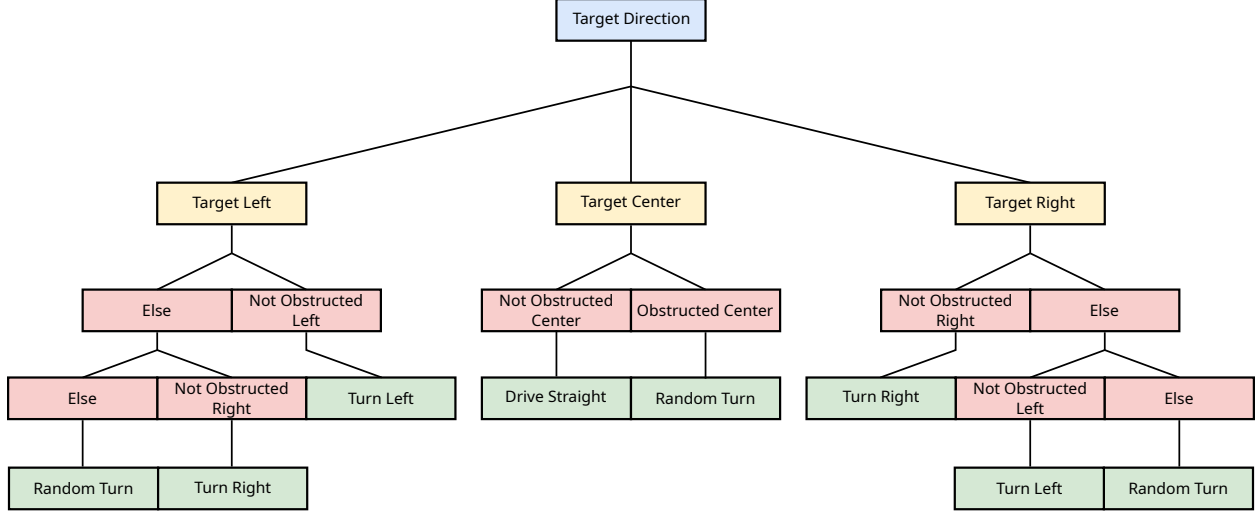


Figure 7.4: Decision tree for traditional algorithms

a random position is set to be the target, with a new random position selected after a configurable number of time steps. The robot then moves towards this position using the targeting algorithm described above. It was found that this was an effective way to roam the environment using an algorithm that was already implemented.

7.2.5 Neuromorphic Algorithm Implementation

In terms of behavior, the neuromorphic algorithm implementation is done the same way that the traditional implementation is done for updating sensor readings and configuring motor speeds. The differences between the two lie in what happens when transitioning from the roaming state to the targeting state, and how the decisions for configuring the motor speeds are made.

Input Encoding

For making decisions, the sensor data was converted into an input packet for the network. This conversion was done by scaling the given sensor reading into a definable range and then converting the corresponding value into the range of DANNA2 fire weights. The specific inputs used will be discussed in [Chapter 8](#). After properly converting all of the inputs, these inputs were placed into an input fire packet that was sent through the DMA to the network.

Dynamic Switching of Networks

The most difficult part of getting to a target in a room full of obstacles is finding the target in the first place. Furthermore, constructing a network that can both search the room to find the target and then pursue the target is an immensely difficult task, especially using genetic algorithms. To combat this, it was decided that two distinct networks would be used to perform these two independent tasks. In the same way that the traditional algorithms switch between roaming and targeting states, the neuromorphic algorithms switch between roaming and targeting networks.

Removing Connectivity and Size Restrictions for Networks

Exploration was done to determine if the connectivity restrictions imposed by the DANNA2 grid array or the size restrictions imposed by the size of the FPGA on the PYNQ-Z1 hindered the potential performance of the robot. To facilitate this exploration, functionality was built into the neuromorphic algorithms that was capable of sending DANNA2 packets to a remote host and receiving the output packets from this same host across a socket. Furthermore, other functionality was added that allowed the creation of a high-level packet structure that could be forwarded to a remote host using a simulated DANNA2 network. These high-level packets allowed for small packets, representing input fires and simulate calls, to be sent to the network, and for a condensed response packet to be returned. This response packet could then be parsed to determine what decision the robot should make. The results of these tests will be discussed in [Chapter 9](#).

7.3 Simulator Modifications

The simulator for this robot is built upon the simulator used for the training of NeoN, but of course, to add the additional complexity of targeting, a number of modifications had to be made [21]. To add the target, an additional entity was added to the game state with a definable position and size. These parameters of this entity are set at runtime. An additional method of scoring the robot's performance was added, such that the robot's performance was measured by how much of the initial distance between it and the target was covered as well

as if the robot successfully reached the target. The distance coverage itself was only defined to be worth 67% of its score, while the remaining 33% was reliant on the robot actually reaching the target. In this way, the fitness function could be designed to emphasize actually reaching the target as this made a substantial difference in the score.

Additionally, an input encoding scheme was added that would convert the position of the target relative to the front of the robot into four components: front, back, left, and right. The magnitudes of these components were then scaled into a configurable range that would effectively allow one to set the resolution of the target's position. This was necessary because the DANNA2 neuromorphic architecture only allows for 16 distinct fire weights, and since the component magnitudes had to be converted to fires to give to a network, proper consideration had to be given to this mapping [20].

There were a number of additional configuration parameters that were added purely based on the results of training and testing, and these will be discussed in [Chapter 8](#).

7.3.1 Traditional Algorithms in Simulation

In order to properly compare targeting using traditional algorithms compared to the neuromorphic ones in all facets, the traditional targeting algorithm used on the robot was ported to the simulated application environment. In essence, the simulator is capable of providing input information that would come from sensors and then take motor speed settings to update the environment. A program was written that would take this input information and then make a decision, based upon the decision tree for the traditional algorithms, on how to set the motor speeds. This was tested to be sure that it functioned properly, and the results of this will be discussed in [Chapter 9](#).

Chapter 8

Training and Testing

Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time.

– Thomas A. Edison

8.1 Network Input and Output

In this section, the input information given to the neural networks, as well as the encoding of these inputs, is discussed. In general, inputs are encoded between a minimum and maximum value, with the minimum value corresponding to a fire weight of zero and a maximum value corresponding to the maximum fire weight of a DANNA2 input which is 15 [20]. Furthermore, the output encoding is also discussed for these networks. Given the sensors used on the robot, one may see the information available for the purpose of network input in [Figure 8.1](#).

8.1.1 Roaming

For the purpose of roaming with obstacle avoidance, the work from NeoN could be used to find an optimal configuration for training [21]. A summarization of the the input scheme may be viewed in [Table 8.1](#). Particularly, notable inputs include the random and bias neurons. The random neuron was added to try to encourage diversity in turning direction. The value for this neuron is changed after a configurable number of time steps. The bias neuron

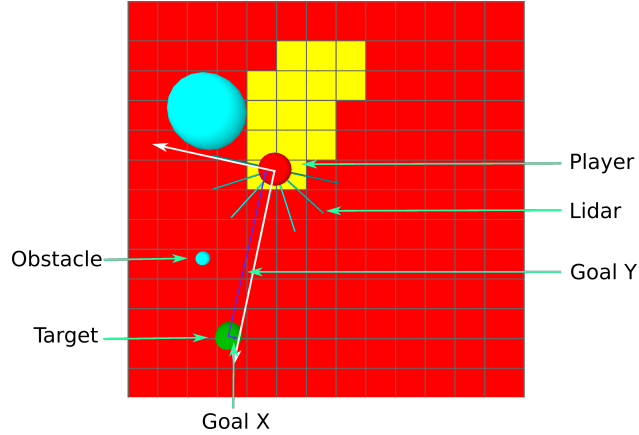


Figure 8.1: Simulator explanation and network input information

makes sure that the network always has input by constantly pulsing a maximum value fire. Additionally, a summarization of the output scheme may be seen in [Table 8.2](#). The output works by the robot beginning to move with a speed of 50% forward on each side. For each fire on a backwards neuron, the side corresponding to that neuron has its speed decreased by 10% and for each fire on a forwards neuron, the side is increased by 10%. This output configuration is referred to as the “speed scheme.” Outputs are collected for 100 network cycles, all ran within a single simulated time step, before a decision is made. It was found that using this input and output scheme from NeoN performed well on GRANT, as well. An example network using this configuration may be seen in [Figure 8.2](#).

8.1.2 Targeting

In order to train targeting networks, more exploration had to be done to determine the proper input and output schemes. It was found that for targeting, adding LIDAR lines perpendicular to the front of the robot aided in obstacle avoidance. Furthermore, it was found that the best way of encoding targeting information was to separate the distance to the target into displacements relative to the front and right sides of the robot. In this way, these displacements could be encoded as X and Y coordinates of the target relative to the robot. In practice, this was implemented with two neurons for each component, such that

Table 8.1: Grid coverage input scheme

Input	Min Val	Max Val	Purpose
LIDAR 1	0.33 m	1 m	Let the network see obstructions
LIDAR 2	0.33 m	1 m	Let the network see obstructions
LIDAR 3	0.33 m	1 m	Let the network see obstructions
LIDAR 4	0.33 m	1 m	Let the network see obstructions
LIDAR 5	0.33 m	1 m	Let the network see obstructions
Random	0	1	Insert randomness for equal turning
Bias	1	1	Always give network input

Table 8.2: Grid coverage output scheme

Output	Purpose
Left Motor Forwards	Increase left motor speed by 10%
Left Motor Backwards	Decrease left motor speed by 10%
Right Motor Forwards	Increase right motor speed by 10%
Right Motor Backwards	Decrease right motor speed by 10%

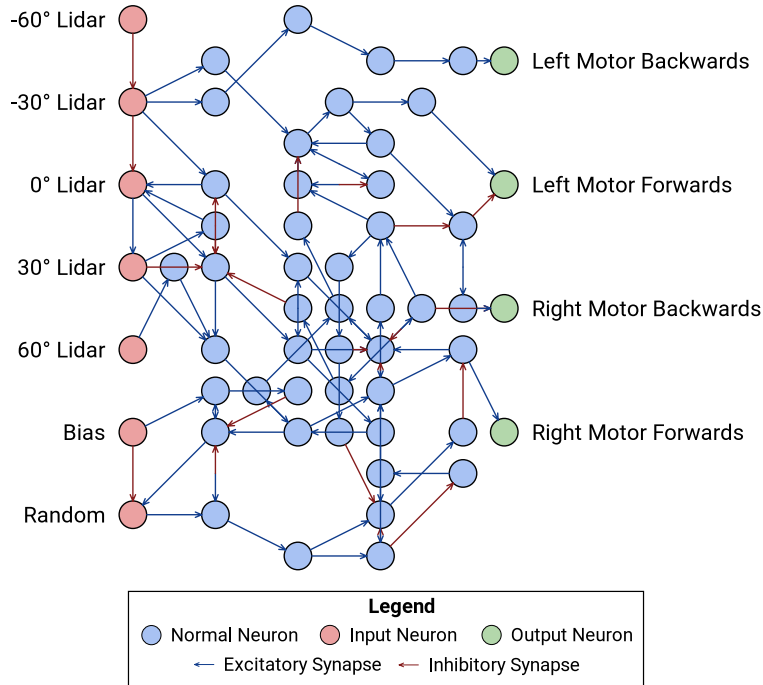


Figure 8.2: Example grid coverage network

Table 8.3: Targeting input scheme

Input	Min Val	Max Val	Num Bins	Purpose
LIDAR 1	0.33 m	1 m	1	Let the network see obstructions
LIDAR 2	0.33 m	1 m	1	Let the network see obstructions
LIDAR 3	0.33 m	1 m	1	Let the network see obstructions
LIDAR 4	0.33 m	1 m	1	Let the network see obstructions
LIDAR 5	0.33 m	1 m	1	Let the network see obstructions
LIDAR 6	0.33 m	1 m	1	Let the network see obstructions
LIDAR 7	0.33 m	1 m	1	Let the network see obstructions
Random	0	1	1	Insert randomness for equal turning
Bias	1	1	1	Always give network input
Target X	-1 m	1 m	2	X component of Target relative to the robot
Target Y	-1 m	1 m	2	Y component of Target relative to the robot

there was a neuron corresponding to the left, right, forward, and backward components of the target’s position. The corresponding input scheme can be seen in [Table 8.3](#).

Further exploration also had to be done for the output scheme. After training attempts were made using the speed output scheme discussed previously, it was found that these networks often translated poorly to the physical system. It was determined that this was due to the difficulty in accurately modeling the motor’s response to a given input. In order to mitigate these effects, a new output scheme was designed. In this new output scheme, output neurons represented a decision to turn right, turn left, go straight, or go backwards. Then, the action with the most votes was the action that was taken. This was designed to use a set turn speed and movement speed, such that when this was translated to the physical system, these speeds could be adjusted until the response matched the response seen in simulation. This new output scheme was named the decision output scheme.

After performing training with the decision output scheme, it was found that the ability for the network to move backwards often hindered its training. In order to combat this result, an additional training parameter was added that allowed for this output neuron to be removed. Furthermore, testing also showed that the priority of these decisions mattered in the event that decisions had an equal number of votes. In this case, it was found that prioritizing turning over moving forward was optimal, as doing otherwise caused the simulated robot to

Table 8.4: Targeting output scheme

Output	Priority	Purpose
Turn Left	3	Vote to turn left
Turn Right	2	Vote to turn right
Move Forwards	1	Vote to move forwards

crash more often. This finalized output scheme used for the training of networks can be seen in [Table 8.4](#) in which a higher number for priority means that the action is taken over one with a lower priority in the event of a tie. The exception to this is in the event that no votes are made, in which case the network will move forwards. Also, an example targeting network can be seen in [Figure 8.3](#).

8.2 Training Methodologies

In this section, the methods through which training occurred and the required resources are explored. The foundational algorithms not previously discussed are also presented here. In general, all trainings were performed on a computing cluster, utilizing 300 CPU cores and the Islands Training Construct. Also, since it was previously determined that different networks would be used for grid coverage and targeting, trainings for each task were done independently.

8.2.1 Islands Training Construct

In order to perform training, the TENNLab framework was used in conjunction with a simulator for the DANNA2 neuromorphic processor and the simulated targeting environment previously discussed. Furthermore, the use of the EONS algorithm allowed networks to be created and evolved over many generations [34]. To facilitate this algorithm, the Islands Training Construct was used. This construct utilized a number of execution threads to allocate a manager and workers. Workers are used to launch independent trainings of networks in their own environments with different random starting seeds, simply transmitting the best network to the manager after a given number of generations passes and potentially receiving a

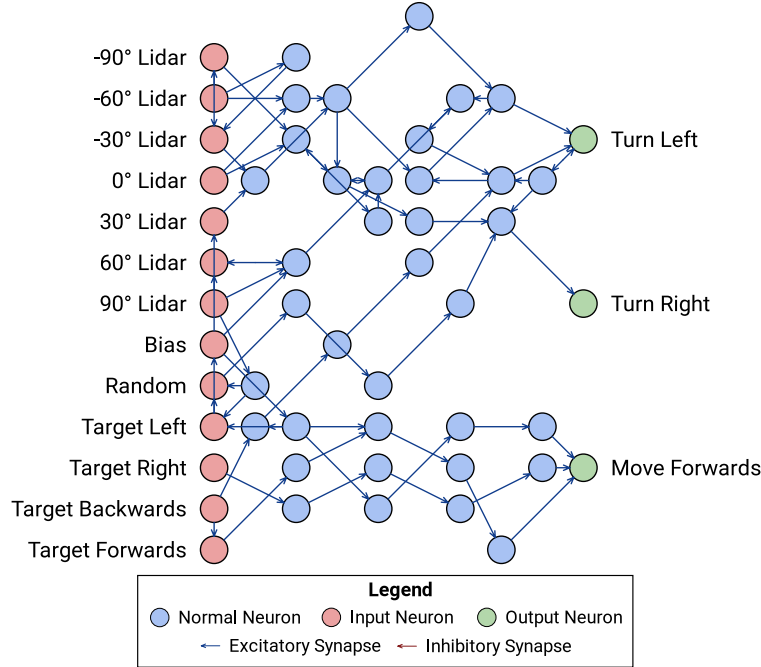


Figure 8.3: Example targeting network

network from the manager. The received network can then be used as a part of the mutation and crossover operations for the next generation within that training.

The manager handles communicating the best networks between various trainings, such that the trainings may function as populations that are receiving additional genetic variation by means of migration. This allows an increased search space and genetic diversity. Furthermore, the manager also has the ability to eliminate a training run if the population stagnates, such that a new training run may be put in its place. By doing this, more genetic variation may be introduced throughout the training process.

8.3 Simulator Parameters

In this section, the various parameters that were added or adjusted for training are addressed. Furthermore, the reasons for adding these parameters as well as how adding them affected both the simulated and physical systems are discussed. A summary of the purpose and the results for each of these parameters can be seen in [Table 8.5](#).

Table 8.5: Summary of parameter effects on training

Parameter	Intention	Results
Moving Target	Account for noise from camera and encoders	Improved behavior on robot. Also allowed target following.
Rotate Target Vel.	Follow target instead of intercepting it.	Removed intercepting behavior.
Cubic Obstacles	Allow handling of sharp corners. Reduce crashing	Crashing was reduced on the robot.
Crash Penalty	Greatly discourage crashing into obstacles.	Trained networks rarely crashed into objects.
Equal Turning Weight	Improve variability in turn direction for avoiding obstacles.	Trained networks “wiggled” while progressing forwards.
Time Step Length	Find optimal setting for time step length.	Short time step caused the robot to not move enough. Long time step caused the robot to move more than its simulated counterpart. At the optimal setting, trained networks on the robot better matched behavior in simulation.
Cycles Between Sensor Updates	Better match the robot by only allowing LIDAR updates every 150 ms.	Trained networks on the robot better matched behavior in simulation. Some loss in performance.
Weak Time Steps	Remove weak turning with the speed output scheme, ideally allowing better network performance on the robot.	Weak turn decisions were removed, but behavior was not improved on the robot.
Bad Time Steps	Emphasize facing the target by penalizing time steps where the network chooses not to.	Resulting networks had better behavior of turning toward the target when possible and maintaining sight line to the target.
Bad Time Steps End Game	Put a cap on the maximum bad time steps and maximum consecutive bad time steps to end the game early in order to force the robot to face the target.	Resulting networks would not move around obstacles at the risk of losing sight of the target. This also caused a wiggling behavior.
Rotated Start Angle	Improve variability and stop networks from always making the same decision at the start.	Networks were more reactive at the beginning of runs as opposed to always making the same decision.
LIDAR Beam Width	Better match the robot by “binning” LIDAR for input.	Trained networks on the robot better matched behavior in simulation.
Training Rooms	Add test cases for large empty spaces, big rooms, and large obstacles.	Trained networks better adapted to a broad range of environments.

8.3.1 Moving Target

One notable difference between the simulator and the physical system was the fact that the simulator always knew the precise location of the target at any given time. On the physical system, calculating the placement of the target with LIDAR data, camera information, and the encoder counts certainly yielded uncertainty in the position of the target. In fact, particular scenarios could yield drastic changes in the believed position of the target between consecutive time steps. For instance, if an obstacle obstructed the view of the target such that only a small portion of the target could be seen, the robot would believe that the target was far away due to its size. Upon navigating around the obstacle and seeing the target again, the camera would see the entirety of the target and the robot would then believe that the target was very close.

It was believed that this discontinuity in target position would greatly hinder the performance of a trained network that had never encountered this. In order to mitigate the effects of the discontinuity, parameters were added to the simulator that allowed the target to move around the space. The target was set to move at a velocity with a configurable magnitude. The direction at which the target moved was set to be a random angle. Also, because the physical system suffers from less noise the closer it gets to the target, parameters were added that made the target stop moving once the robot was within a configurable distance of the target. Through the use of these parameters, the robot would not have to chase the target once it was close enough.

It was found that with this configuration, the simulated robot would often try to intercept the target, moving where it believed the target was going to move to instead of moving toward the target itself. While this was an interesting result, it was not desirable for the effect it was intended to mitigate. To combat this predictive behavior, an additional parameter was added to set the probability of the velocity of the target to rotate. When the velocity rotated, the target would instantaneously change direction, and this was seen to be a better representation of the noisy environment presented to the physical system. In simulation, this provided better resulting behavior, with the robot following the target instead of trying to

intercept it. It was found through testing that the optimal values for the parameters were a speed of 0.3 meters per second and a probability of rotation of 0.1% per time step.

8.3.2 Cubic Obstacles

Originally, in the simulator, all obstacles were implemented as spheres. While one may believe that the shape of the obstacles in simulation would have little to no bearing on the performance of the network on the physical system, this was simply not the case. Because objects were round in simulation, the network had learned how to stay up against the object with sweeping turns, such that it could minimize the amount that its angle to the target changed. This certainly seemed like desirable behavior, but when the physical system encountered an object with hard, straight edges, its attempt at sweeping turns ultimately led to it crashing into the obstacle.

To combat the issues presented by spherical obstacles in the simulator, a parameter was added that allowed the simulated obstacles to be cubes instead. This was implemented by adding walls around the spherical obstacles that were tangent to the surface of the sphere. By doing this, the simulated robot often encountered objects with hard corners, and would thus have to exhibit the behavior of actually having to move away from the target in some capacity in order to avoid an obstacle. This new behavior proved to be ideal on the physical system as it allowed avoidance of obstacles, even at the cost of increasing the distance to the target when necessary.

8.3.3 Crash Penalty

Originally, the simulator contained a parameter for multiplying the fitness score by a value if the simulated robot crashed into an obstacle. This was used as a means of greatly discouraging the robot from choosing to do so, and, in general, it worked well for the purpose of roaming. When training targeting networks, however, this was found to be less effective because the targeting application had to be trained with a large variety of rooms. Even when setting the fitness score for a room to be zero in an event of a crash, it had little impact on the overall fitness score if the simulated robot performed well in the other rooms. Furthermore, a

negative number could not be used since the fitness score could also be a negative value if the network chose to move away from the target instead of towards it. In fact, when this value was set to be negative, resulting networks developed the habit of driving away from the target before crashing. These networks would receive a negative fitness score with a large magnitude that was then multiplied by a negative number, such that the resulting score for the network was a large, positive value.

In order to make singular rooms in which the robot crashed carry more weight, an additional parameter was created that allowed one to set a quantity to be subtracted from the fitness score in the case of a crash. By setting this value to be a relatively large number, one could cause training to emphasize not crashing above everything else; however, if the magnitude of this number was too large, the resulting networks would never learn to approach the target. It was found that the optimal setting of this parameter was a value of 0.5, as this value greatly penalized the network for crashing while still allowing it to learn to approach the target.

8.3.4 Equal Turning Weight

It was found through training that resulting networks often trained to be biased to turn in a singular direction in order to avoid obstacles. This was, of course, undesirable behavior, because this could often result in the network choosing to make a 270 degree rotation instead of a 90 degree rotation. On the physical system, this could cause further issues, as error in tracking the position of the target compounds as the robot goes longer without seeing the target.

In an attempt to combat the unequal nature of turn direction, it was attempted to use a parameter that already existed in the simulator. This parameter made the network emphasize turning both directions as it made a portion of the network's fitness score based on the difference in the frequency it chose to turn each direction. If it chose to make right and left turns an equal amount of times, it would receive a boost to the fitness score. While it may seem that this would have developed good behavior, it actually did quite the opposite. By making equal turning a portion of the fitness score, resulting networks would essentially choose to "wiggle" as they progressed towards the target. In doing so, these networks would

still only turn one direction to avoid obstacles, but these periods of turning were partially balanced by it making so many little turns. In essence, the frequency with which it chose to turn at all helped mitigate the impact of those times in which it had prolonged periods of turning a single direction. Since this wiggling behavior was detrimental to the physical system's operation, it was decided that this equal turning emphasis would not continue to be used.

8.3.5 Time Step Length

In the simulator, the robot operates in discrete time steps, such that input is gathered, a decision is made, the new position and orientation of the robot based on this decision is calculated, and the robot is moved to this position and orientation. The length of this discrete time step is configurable, and changing this parameter can have a significant impact on training and the performance of the network on the physical system. If this time step is too long, the physical system's response will not match the simulated system, usually causing turning beyond the amount seen in the simulated environment. If this time step is too short, the physical system will not move due to the time it takes for the robot to reach full speed from a stopped state.

Furthermore, it was found that the fastest the sensors could be updated on the physical system was 150 milliseconds. This was due to the preprocessing required on the LIDAR data and the LIDAR's sample rate. In an attempt to accommodate this, training was performed using 150 milliseconds for the length of the time step. While networks could be trained to perform targeting in simulation with this setting, the network's performance never translated properly to the robot. After further testing, it was found that 10 milliseconds was the optimal length for the time step as it allowed the simulator and physical system to best match one another. This, of course, meant additional functionality had to be added to the simulator to accommodate the maximum update rate of the sensors.

8.3.6 Cycles Between Sensor Updates

Due to the maximum update rate of the sensors on the physical system as well as the necessity of a short time step length in simulation, a parameter was added to the simulator to allow the LIDAR to only be updated after a set number of time steps. In essence, network's were required to work with old LIDAR information for a set number of cycles before the information was updated; however, the position of the target was still updated at every time step since its position was tracked using the encoder readings on the physical system. In order to accommodate the 150 milliseconds required for sensor updates, this parameter was set to a value of 15, such that the LIDAR was updated after 15 time steps of 10 milliseconds each.

Trained networks that used this parameter saw some loss in performance in simulation, which was sensible as these networks were trained with less information than their predecessors. The behavior of these networks in simulation was, however, mirrored by the physical system. These networks, in general, showed excellent tracking behavior and good obstacle avoidance to one side of the robot. If the robot approached an obstacle on its other side, though, it would often turn towards this obstacle and wiggle. In general, the network is usually able to get around the obstacle where it can then continue exploring, but, occasionally, the robot would crash into the obstacle if it is significantly large, such as a wall. It is theorized that the robot developed this behavior because it generally approached the target with the target being on that side of the robot, such that it needed to turn towards it. Furthermore, in watching the simulated environments, it was apparent that obstacles, purely by chance, were usually approached with them being on the side of the robot that was good at avoidance.

8.3.7 Weak and Bad Time Steps

In initial testing of the physical system with networks using the speed output scheme, it was found that the robot struggled to emulate turns in which the difference between the speeds on each side was small. This led to the creation of a new parameter that allowed one to define both weak turns as well as a penalty in the fitness score associated with it. In essence, if the difference between the motor speeds was not greater than the defined threshold, that

time step was considered to be a “weak” time step. Then, the percentage of weak time steps out of the total number of time steps was multiplied by the defined penalty factor and the resulting quantity was subtracted from the fitness score. This did, in fact, yield the desired behavior of forcing the network to make strong turns when the threshold was defined to be 100% and the penalty multiplier was set to four; however, on the physical system, it still appeared that the movement did not match the simulator. The result of this test was the primary driving force in the development of the decision output scheme.

With the development of the decision output scheme, “bad” time steps were also added to track time steps in which the target was not within the front 180 degrees of the robot. This was intended to enforce optimal movement towards the target and enforce that the network learn to turn both directions to avoid obstacles. Furthermore, because the robot would ideally always face the target, prediction of target location would be improved. Development began with a system that worked identically to the weak time steps where bad time steps were simply counted and the percentage of bad time steps multiplied by a penalty was subtracted from fitness. Further development and testing was done beyond this to further facilitate the idea of always facing the target. A parameter was added that enforced a maximum number of bad time steps before the game ended. Upon the game ending by the robot reaching the maximum number of bad time steps, a penalty was subtracted from the fitness score, such that the robot suffered further penalty from this behavior. A maximum number of consecutive bad time steps was also added as a parameter, such that the robot could only persist for a short period of time without facing the target before the game would be ended.

Through training and testing, it was found that prematurely ending the game for not facing the target yielded networks that would never turn away from the target, even if it was necessary to reach the target. It was, however, useful to still count these bad time steps and penalize the network for them, as doing so yielded networks that learned to turn both ways to avoid obstacles.

8.3.8 Rotated Start Angle

The original simulator was configured to always start with the robot facing down the x-axis if viewed from above. Through experimentation, it was found that this caused networks to

be biased in the direction they turn. In order to mitigate this issue, an additional parameter was added that was simply a boolean used to determine if the starting direction of the robot should be rotated. When enabled, the robot was rotated a random angular amount from its initial orientation. This further randomized the scenarios the robot faced, such that the robot could ideally learn better behavior.

Networks trained with this parameter enabled showed improvement over their predecessors. In general, these networks were more likely to turn in both directions to avoid obstacles.

8.3.9 LIDAR Beam Width

As mentioned in [Chapter 7](#), the wrapper library implemented for the LIDAR supported the ability to “bin” the LIDAR measurements. This was done for the purpose of mitigating blind spots and increasing the information that could be given to the network. In order to accurately train a network that would perform the same in simulation as it would on the physical system, this same behavior had to be implemented in the simulator.

To implement this, a parameter was added to represent the beam width which is how many LIDAR angles should be combined to represent a single LIDAR input to the network. Then, this number of LIDAR angles were added to the simulator for each LIDAR measurement that was supposed to exist, centered on the angle that would have been selected by the simulator. Each of these additional LIDAR angles were checked, and the minimum reading was checked for each of the groupings.

Using this parameter made the performance of the network on the robot better match the performance in simulation. It was found that the optimal beam width was nine, such that each LIDAR input to the network effectively had the information from a nine degree wide area.

8.3.10 Adapting Training Rooms

After performing a large amount of testing on the physical system, it was found that certain scenarios yielded poor results. In particular, it was first discovered that the networks could not handle situations in which they did not have at least one obstacle intersecting a LIDAR.

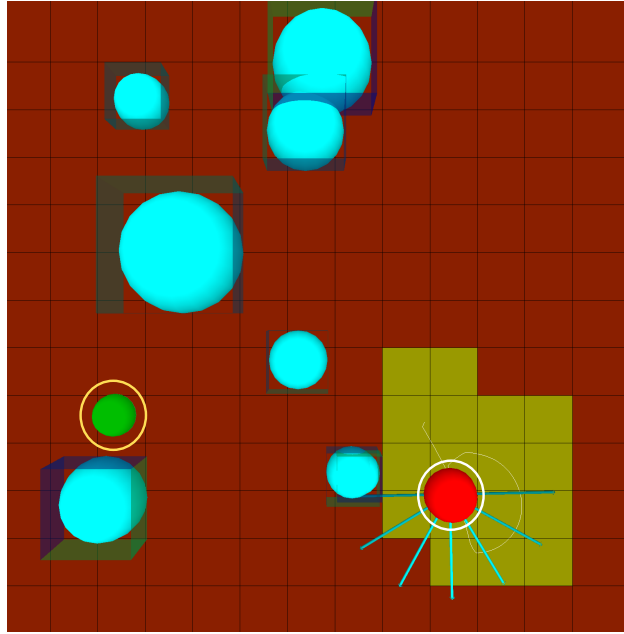


Figure 8.4: Small room 1

When in these scenarios for an extended period of time, the robot would perform seemingly random motion. Upon inspection of the performance in simulation, it was found that the trained rooms only mildly addressed this scenario with a small, empty room, as the initial set of rooms was composed of the rooms in [Figures 8.4, 8.5, and 8.6](#). In order to combat this, large rooms were added like those shown in [Figures 8.7, 8.8, and 8.9](#). Also, it is worth noting that each of these rooms are merely examples, and variations of each of these rooms were tested with different obstacle distributions. Additionally, all of these rooms use a white circle to identify the position of the robot and a yellow circle to identify the position of the target.

The networks that resulted from the inclusion of these larger rooms in training showed the ability to navigate large spaces in pursuit of the target; however, if the physical system was placed in an area with very large obstacles, such as a wall, it was found that the network would choose to crash into the wall after attempting to go around it for some amount of time. Upon inspecting the simulated environments, it was found that this scenario was also not accounted for in the rooms used for training, as the robot never had to encounter the walls or large obstacles. For this reason, additional large rooms were added that contained large objects and additional walls like those in [Figures 8.10, 8.11, 8.12](#).

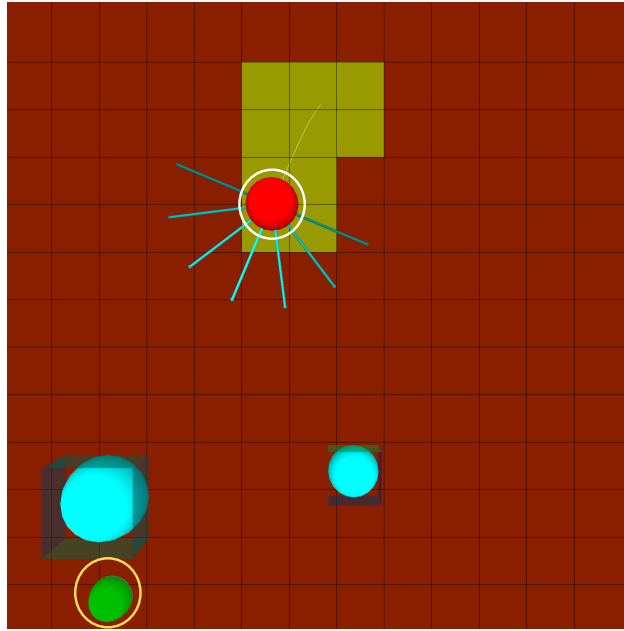


Figure 8.5: Small room 2

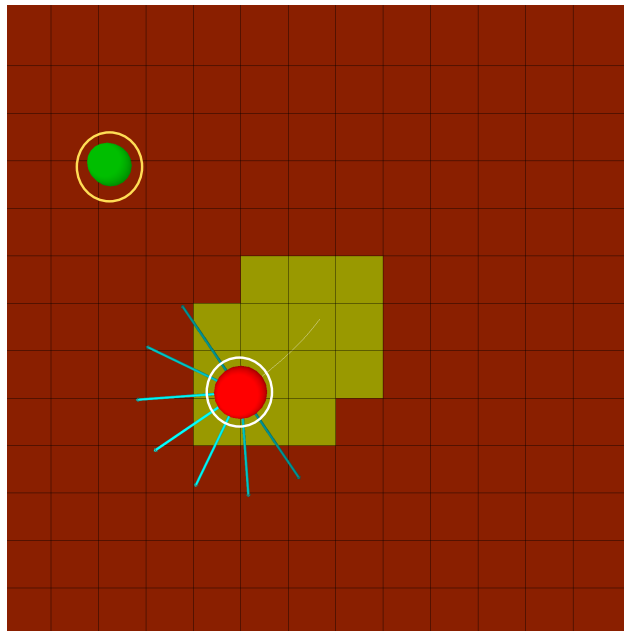


Figure 8.6: Small room 3

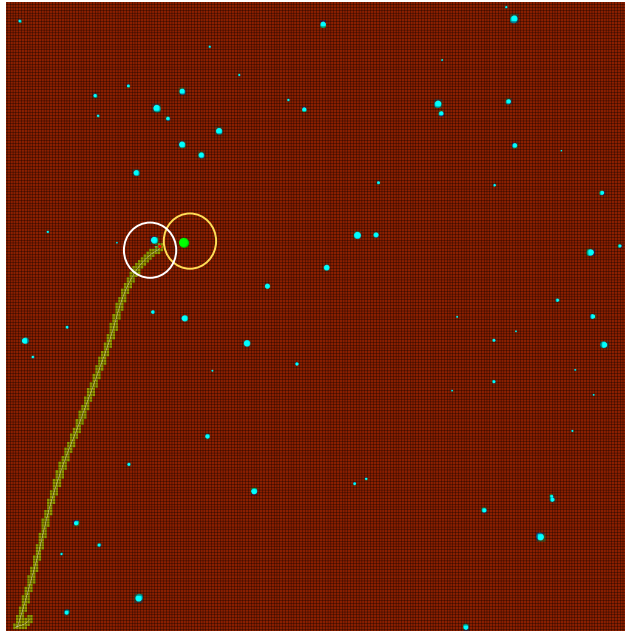


Figure 8.7: Big room 1

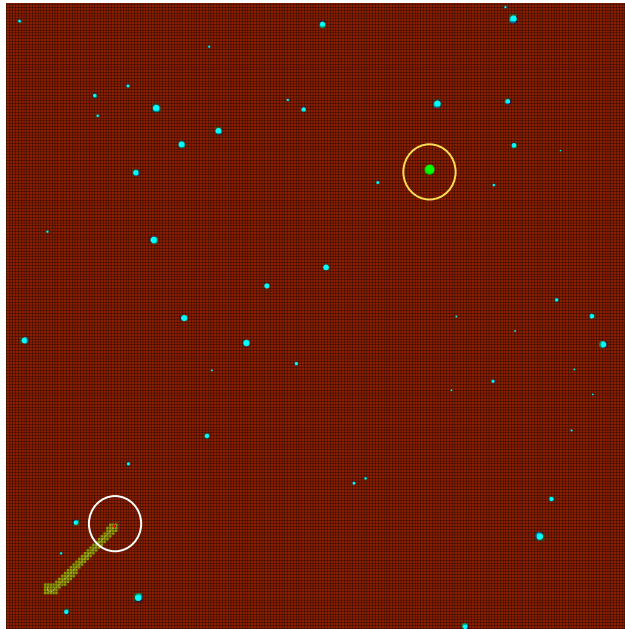


Figure 8.8: Big room 2

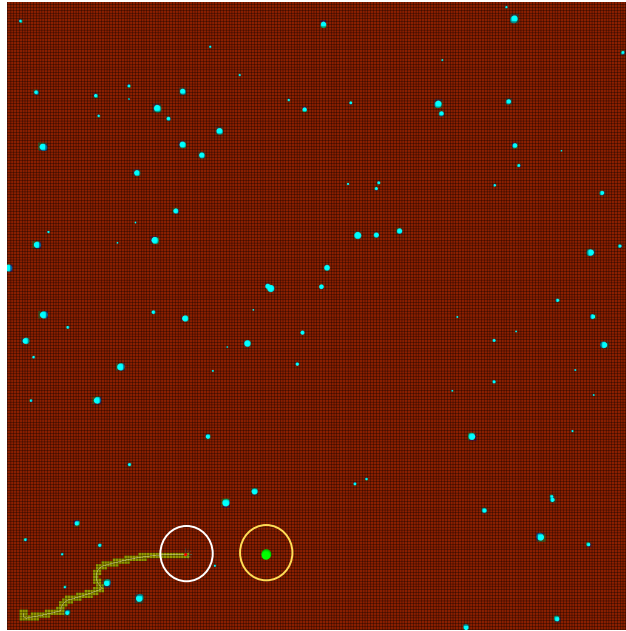


Figure 8.9: Big room 3

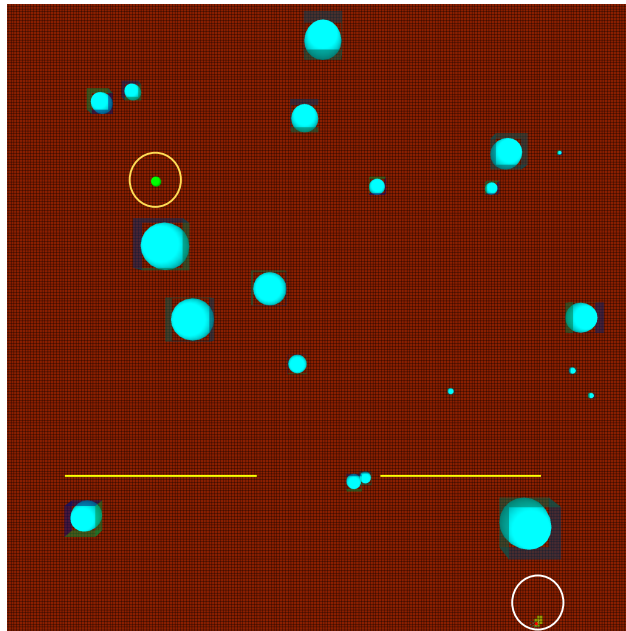


Figure 8.10: Hard room 1

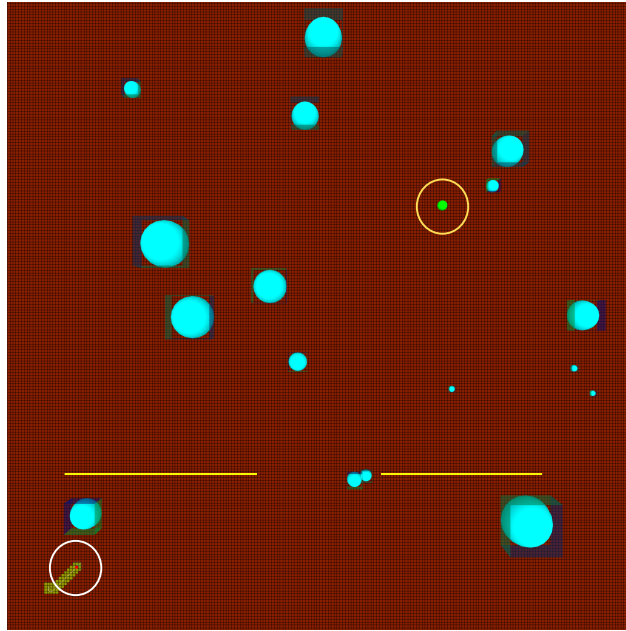


Figure 8.11: Hard room 2

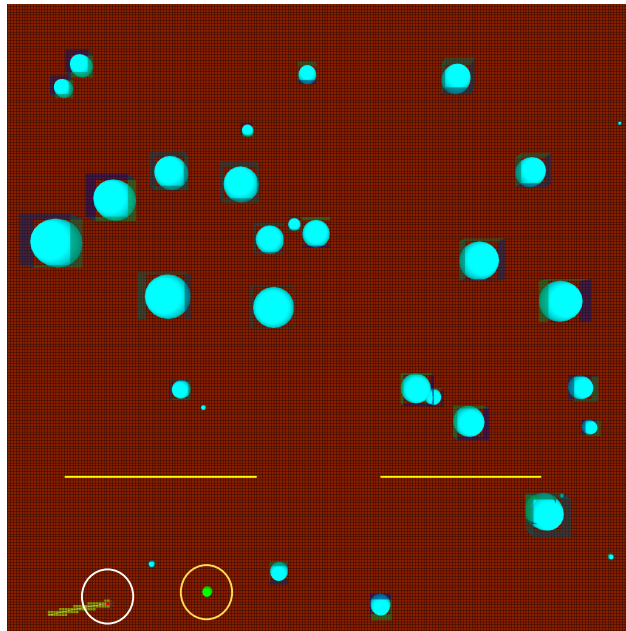


Figure 8.12: Hard room 3

8.4 Hardware and Software Complications

There were, of course, some struggles in the development of the robot. In particular, three primary areas became a significant source of difficulty that had to be overcome for this robot to be realized: Petalinux, wheels, and power delivery.

8.4.1 Petalinux

Petalinux, as a whole, is not widely used, and is thus not documented very well. There were significant difficulties that went into the installation of the program, and even more difficulties appeared when trying to work with it. In particular, the biggest challenge revolved around adding devices to the hardware device tree. Without doing this, the robot could never possibly function, as these devices included the encoders, motor PWM control, and the DMA for communication with the DANNA2 grid array.

To overcome this, a significant portion of research and experimentation was done to find out how to properly update this hardware device tree. Unfortunately, this required rebuilding the generated operating system each time, so the testing itself progressed rather slowly. Eventually, however, it was discovered how to add these devices properly, and once the kernel was configured with all of the proper drivers, enabled options, and software, the operating system was ready to be used on the robot.

8.4.2 Wheel Problems

Originally, a robot kit was purchased to use for the robot. This robot kit included two inch wide, rubber wheels capable of navigating a variety of terrains as seen in [Figure 8.13](#). While these wheels seemed ideal, they caused some undesirable behavior on the robot. In particular, because the robot itself was too heavy and the motors could not produce enough torque, the robot was incapable of turning with these wheels. This was due to the horizontal friction across the width of the wheel.

Because of the issues with the first set of wheels, a different set of wheels were tried. These wheels, while still rubber, were only 0.75 inches wide and may be seen in [Figure 8.14](#). It was believed that the reduced surface area in contact with the ground would reduce the



Figure 8.13: Original rubber wheel

frictional forces such that the wheels would turn, but this did not prove to be the case. The motors still could not produce enough torque to turn the robot appropriately.

Finally, a set of omni-directional wheels were tested on the robot as seen in [Figure 8.15](#). These wheels featured horizontal rollers perpendicular to the axis of rotation for the wheels. This meant that the robot did not have to overcome any additional friction in order to turn. By using these wheels on the robot, it was possible to perform precise movements that could then be used to perform targeting and obstacle avoidance.

8.4.3 Power Delivery

The last notable issue came in the form of power delivery to the various components of the robot. In particular, spikes in current draw during the robot's operation would occasionally cause the voltage of a fixed, five-volt regulator to drop lower than 4.8 volts on its output. This sudden voltage drop could, at times, cause the sensors to reset. When this happened, the sensors stopped collecting readings and the robot would, in turn, almost always crash into an obstacle. Because it was not originally known what caused the sensors to reset, a variety of potential issues were evaluated, including the USB hub, the SoC's USB port, and the sensors themselves. Unfortunately, this took a good deal of time to evaluate before finally determining the true cause of these issues.

In the end, the resolution of this problem was to simply use a regulator made by Pololu that supported a higher current draw [\[30\]](#). Using this regulator, the output voltage never



Figure 8.14: Banebot rubber wheel

suffered the sudden drop that had plagued the first one, and thus, the sensors ceased this resetting behavior.

8.5 Tuning Robot Parameters

In initial testing, parameters on the robot itself had to be adjusted to achieve the appropriate response with the system. In particular, parameters corresponding to how close the robot has to be to have reached the target, how close obstacles have to be to be considered obstructions, how large the target has to appear before it is believed to not be obstructed, and the amount of cycles in which the robot does not see the target must occur before it reverts to roaming. This set of parameters was tested to optimize traditional algorithms first, but it was found that after adjusting them for this purpose, the parameters relevant to the neuromorphic implementations did not have to be changed.

Additionally some fine-tuning of parameters had to be done for each network that was tested on the robot. After a network was trained, the motor speeds corresponding to turning and movement had to be adjusted such that the behavior on the physical system matched the behavior in the simulator. This was done by closely studying the behavior observed in the simulator and on the physical system with small adjustments in these speeds being made until the systems appeared to match.

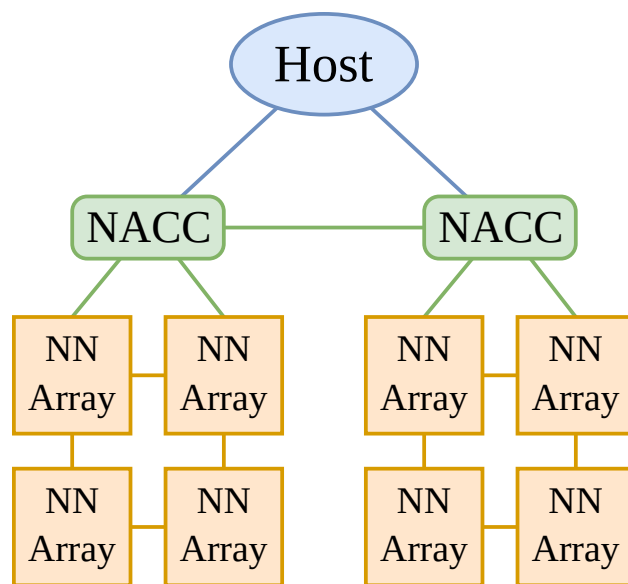


Figure 8.15: Omni-directional wheel

8.6 Removing Network Restrictions

In an effort to further explore the capabilities of GRANT and network training, experiments were run that would allow the robot to communicate with a more powerful host. The robot code was modified to either transmit the full packets or condensed input, simulate, and configuration commands across a socket to a remote host. This host would then evaluate the network with these packets or commands and respond with either output packets or a condensed output bundle that defined the activity over a given period. The robot could then, upon receiving a response, process the output and execute the proper decision. These tests were performed over ethernet and WiFi and supported running with the network simulator, a larger FPGA, and SNACC, a Scalable Neuromorphic Array Communication Controller [42]. A depiction of the structure of SNACC system may be seen in [Figure 8.16](#), where the communication controller allows multiple small, DANNA2 grid arrays to be treated as one large array. Through the use of SNACC and the larger FPGA, network size restrictions were eliminated, and by utilizing the network simulator, connectivity restraints could be removed as well.

The results of these tests, including network sizes, latencies, and performance, will be discussed in [Chapter 9](#).



SNACC

Figure 8.16: Model of the SNACC system [42]

Chapter 9

Results

The only good luck many great men ever had was being born with the ability and determination to overcome bad luck.

– Channing Pollock

Grid coverage was shown to behave much like that exhibited by NeoN [21]. The network proved to be cautious around obstacles, virtually never crashing. In general, it explored a space quite well covering approximately 60% of the room. This coverage does not account for areas of the room that were unreachable due to being occupied by obstacles or being blocked off by obstacles. An example of the coverage performance of this network may be seen in [Figure 9.1](#). The behavior exhibited by this network is mirrored on the physical system, as well.

The remainder of this section discusses the performance of the targeting implementations as these were the novel point of this work.

For the basis of these tests, five different implementations of the targeting algorithm were tested. This included a sparse network with 33 lines of LIDAR and one degree wide bins (`sparse_33`), a grid network with 33 lines of LIDAR and one degree wide bins (`grid_33`), a grid network with seven lines of LIDAR and nine degree wide bins (`grid_7`), traditional algorithms using 33 lines of LIDAR and one degree wide bins (`trad_33`), and traditional algorithms using seven lines of LIDAR and nine degree wide bins (`trad_7`). Tests in the simulator were ran on 144 unique rooms that had not been previously experienced by the

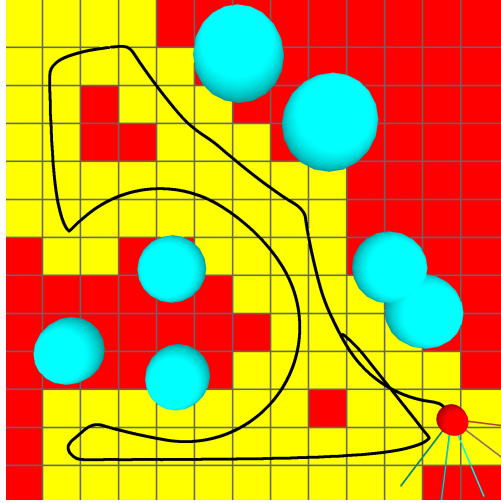


Figure 9.1: Grid coverage performance

networks in training. Also, the networks and traditional algorithms were given 430 seconds to reach the target in these simulated rooms.

9.1 Simulated Performance

By nature, the ability of a system to target an object is hard to quantify into metrics. In essence, what is desirable is a matter of behavior, especially when the simulated performance needs to be translated into an imperfect, physical system. For this reason, the behavior of each of the listed networks and the traditional implementations is discussed here before attempting to quantify the performance into measurables.

9.1.1 Behavioral Performance

The behavior for each network and each traditional algorithm was observed for each of the 144 rooms used for testing. Herein, the observed behavior exhibited by each is discussed. Also, for the behavioral path images presented in this section, the same random seed was used for each room, such that the motion of the target as well as the distribution of the obstacles is the same. It is worth noting that these path images are not indicative of the way

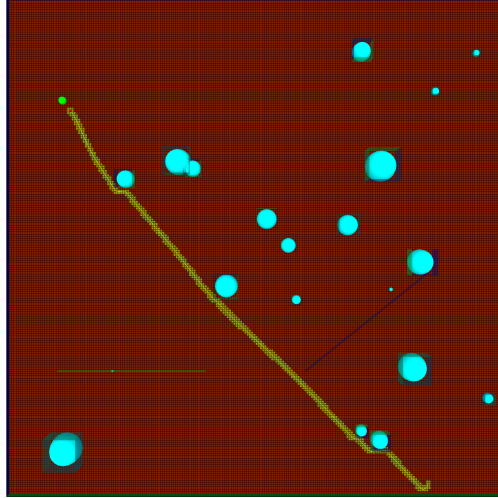


Figure 9.2: Grid_7 example path

the networks always perform, but they do represent an example of the performance that each may exhibit.

Grid_7

The `grid_7` network, in general, performs quite well in pursuit of the target. It turns slowly towards the target while in pursuit of it. This network does, however, perform some undesirable behavior in that the network will occasionally turn towards obstacles. In some of these cases, the network will eventually turn back towards the target and continue its pursuit. In the other cases, the network will either crash into the obstacle or remain stuck in front of it until it runs out of time. An example of its path can be seen in [Figure 9.2](#).

Grid_33

The `grid_33` network behaves nearly identical to the `grid_7` network. It exhibits the same behavior of turning slowly towards the target while pursuing it. It takes slightly more efficient paths on average than the `grid_7` network. This network also gets stuck on obstacles less often and will, in general, turn towards obstacles less. An example of its path can be seen in [Figure 9.3](#), where it performs nearly identically to `grid_7`.

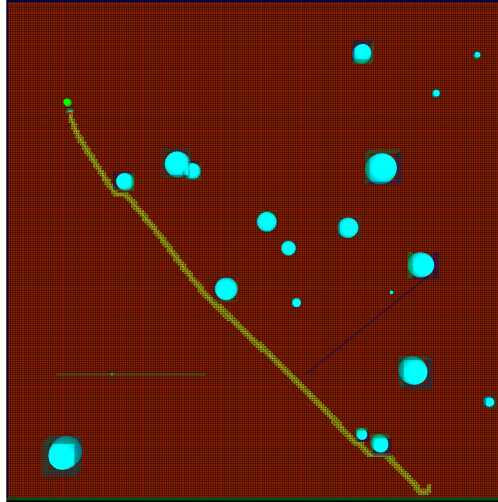


Figure 9.3: Grid_33 example path

Sparse_33

The `sparse_33` network is very hesitant to approach obstacles. It does so slowly and methodically, while also exhibiting a wiggling behavior as it progresses forward in the space. This leads to the network being rather slow to reach the target, but it reaches the target often. Also, because of this wiggling behavior, the network can also lose by running out of time without getting stuck on obstacles merely because of how slow it progresses. An example of its path can be seen in [Figure 9.4](#), in which the network ends up crashing into an obstacle near the completion of the room.

Trad_7

The traditional algorithm, when ran with seven lines of LIDAR and nine degree wide bins, exhibits decent behavior in simulation. The algorithm prioritizes facing the target, and will attempt to do so before progressing forward. In the event that turns must be made to avoid obstacles, the algorithm chooses to do so as outlined in [Chapter 7](#). This implementation does, however, have issues when using a small number of LIDAR lines. With this reduced input, the algorithm ends up crashing into obstacles often due to blindspots created by poor decision-making. An example of this crashing may be seen in [Figure 9.5](#).

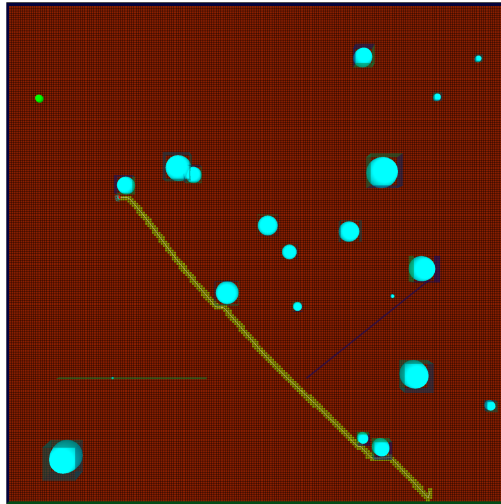


Figure 9.4: Sparse_33 example path

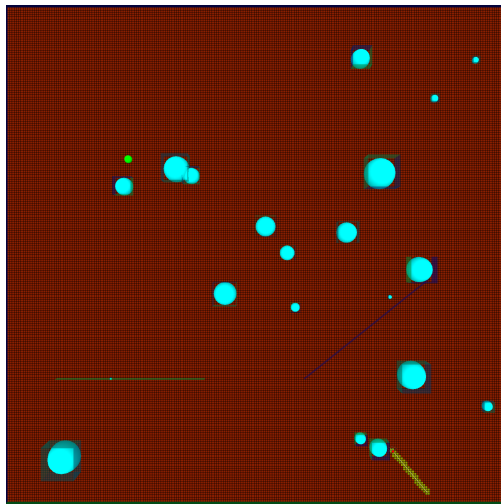


Figure 9.5: Trad_7 example path

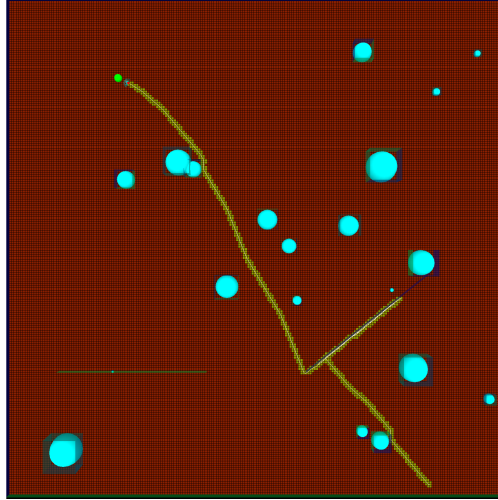


Figure 9.6: Trad_33 example path

Trad_33

Much like `trad_7`, this implementation behaves like the algorithm outlined in [Chapter 7](#). The increased input space of this implementation, however, allows the algorithm to avoid obstacles very well, such that it rarely crashes into one. In general, the algorithm has high efficiency in reaching the target quickly, but occasionally, the nature of the algorithm will cause it to take a less efficient path as seen in [Figure 9.6](#). In this path, the algorithm chose to patrol up and down the wall in an attempt to reach the target before finally being able to get around the wall. It is worth noting, however, that this test case is among those with the worst performance for this implementation.

9.1.2 Performance Metrics

When evaluating the results of a network or traditional algorithm traversing a room to reach the target, there are three potential outcomes: a timeout where the simulated robot runs out of time before reaching the target, a crash where the simulated robot collides with an obstacle, or a win where the simulated robot reaches the target within the time limit. From this, we may gather four important metrics: win rate, crash rate, timeout rate, and the average time to reach the target.

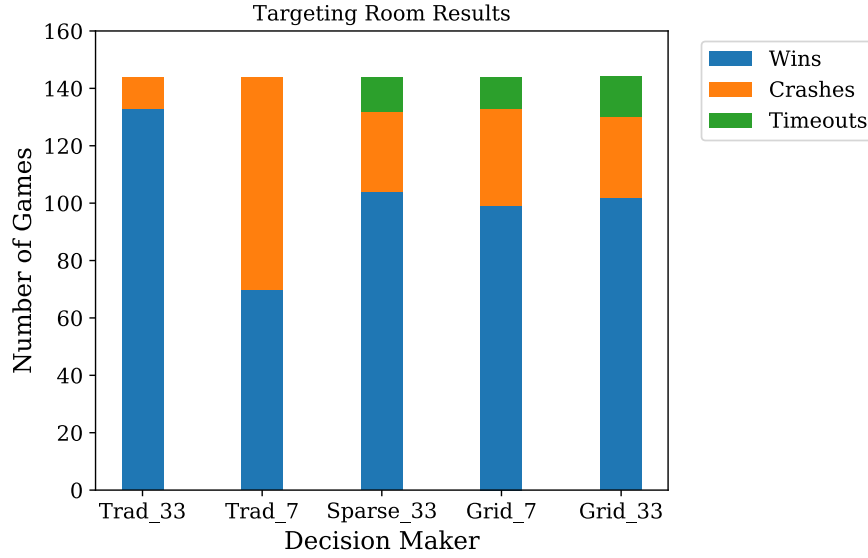


Figure 9.7: Implementation room results

Win rate represents how often the given methodology, traditional algorithms or a network, is able to reach the goal. Timeout rate is indicative of inefficiencies exhibited by the given methodology. These inefficiencies may be in the decision-making of how to approach the target or potentially in the way the methodology seeks to avoid obstacles. For instance, a methodology that is scared to turn away from the target may become stuck when an obstacle is directly between it and the target. The crash rate is often representative of the recklessness associated with a given methodology. In general, a methodology with a higher crash rate is more efficient at getting to the target but ends up hitting the corner of obstacles trying to navigate too tightly around them. The average time to reach the target shows the efficiency of a given methodology. A shorter time, on average, represents a more efficient strategy for approaching the target.

These metrics for the given tests are presented in [Figures 9.7](#) and [9.8](#).

One may see that while `trad_7` was the fastest at reaching the target, it was also the most reckless, having a very high crash rate comparatively. It is also worth noting that the random turning exhibited by the traditional algorithms keeps them from ever getting stuck, as neither `trad_33` nor `trad_7` ever suffer from running out of time. The traditional algorithms do exhibit the best path planning on average, as they both reach the target faster

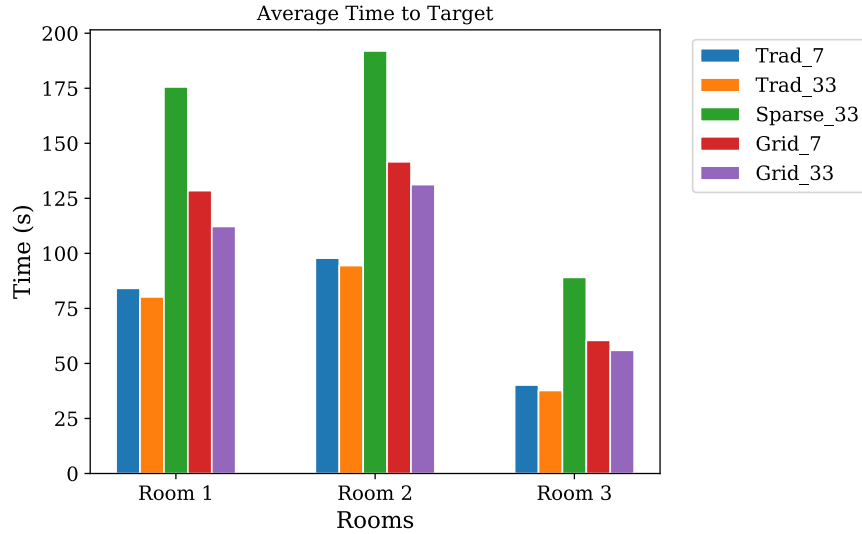


Figure 9.8: Implementation average time to reach target

than any of the approaches attempted by the neural networks. While the `sparse_33` network is the safest and reaches the target the most out of the networks, it does so inefficiently, as it takes the longest time to reach the target on average.

9.1.3 Network Size and Connectivity

As the networks had comparable results in terms of performance and one purpose of using these RSNNs is to create small networks, it is worthwhile to evaluate the size of the networks as a basis of comparison. In [Figure 9.9](#), one may see the total neuron and synapse count for each of the three networks. Furthermore, depictions of these networks may be seen in [Figures 9.10](#), [9.11](#), and [9.12](#) where the grey neurons represent unused elements in the grid array.

The size and configuration of these networks contain some interesting points to note. It is obvious that using grid networks greatly increases the size required for a given network. This can be seen by comparing the size of the `grid_33` and the `sparse_33` networks. While each network requires the same number of inputs and outputs, the `sparse_33` network found a direct mapping from the inputs to the outputs. In this way, this network conserved a considerable portion of resources. Even when comparing the `grid_7` network to the

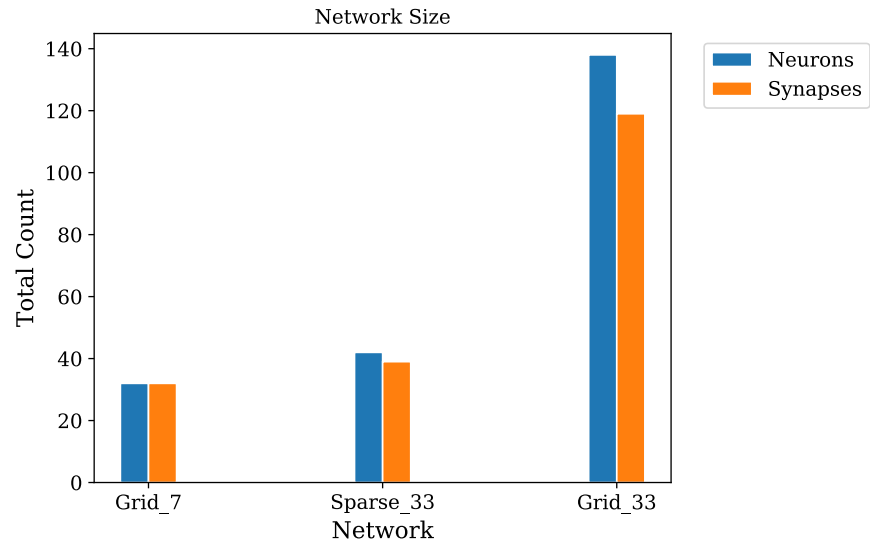


Figure 9.9: Network sizes for best networks

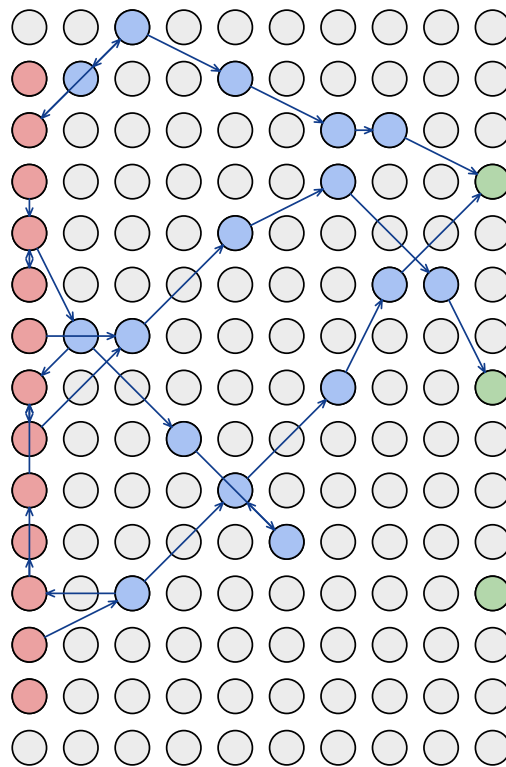


Figure 9.10: Grid_7 network

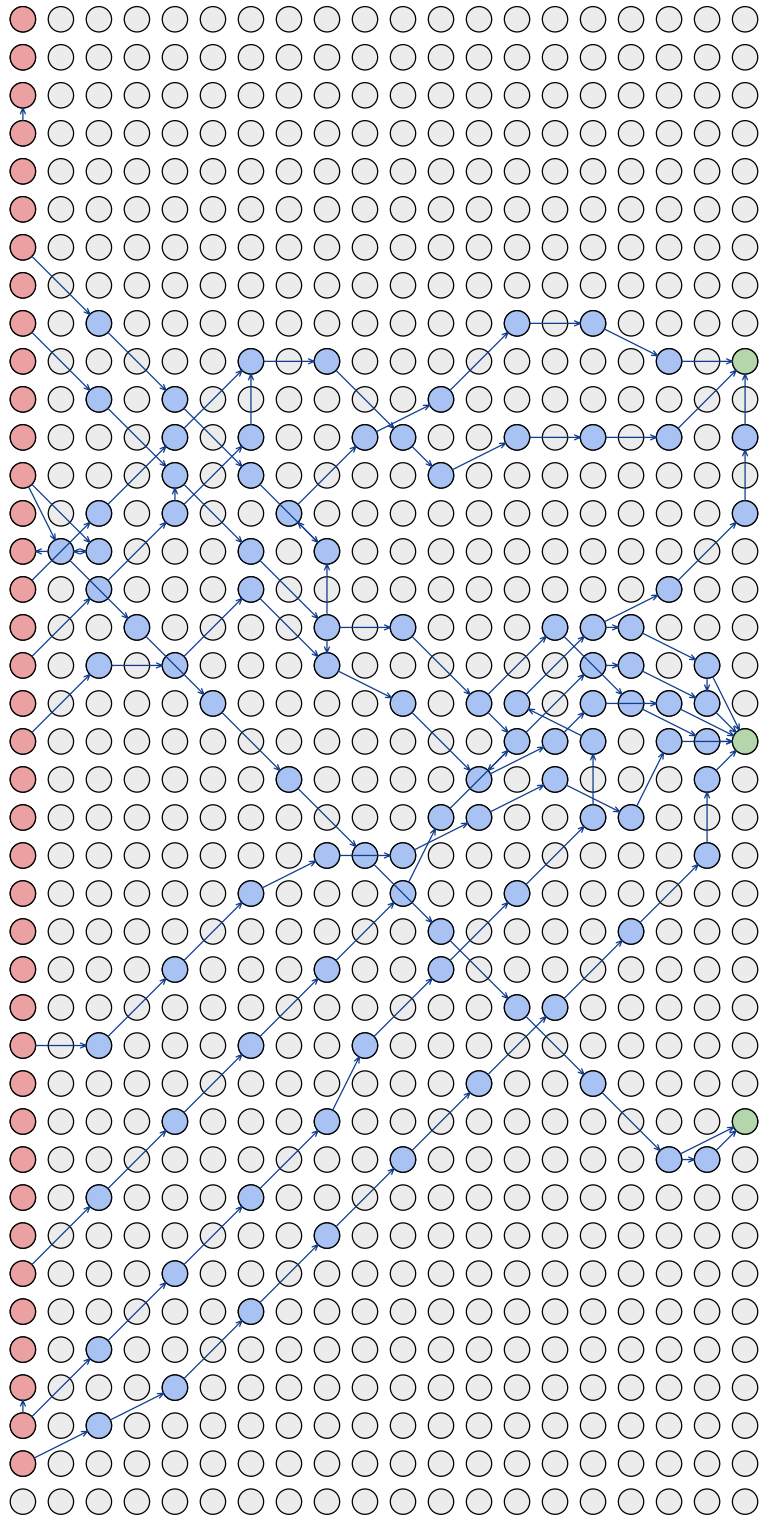


Figure 9.11: Grid_33 network

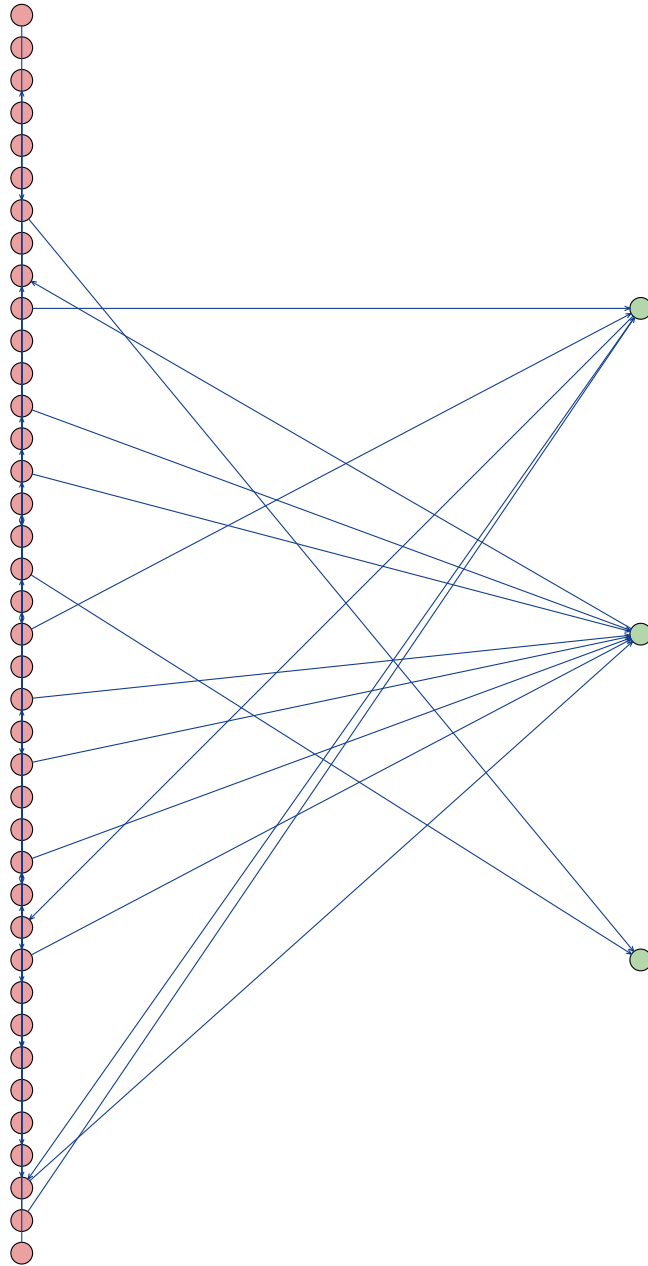


Figure 9.12: Sparse_33 network

`sparse_33` network, it is apparent that while `sparse_33` has substantially more inputs, it does not use many more resources than the `grid_7` network.

It is also apparent that too much information was given to those networks that have 33 lines of LIDAR. When looking at the configuration of these networks, one may see that many of the input neurons for the `grid_33` and `sparse_33` networks are left unconnected. If the networks have found a solution without these inputs, then it is reasonable to assume that they are unnecessary. For this reason, it is likely that the `sparse_33` network could be further reduced in size by reducing the number of input neurons it has.

Lastly, there is a very interesting quality to note about the `grid_7` network. This network has no connections to the output that chooses to move forwards. Because of this, the only way this network may ever move forwards is if the networks makes no votes to turn right or left. It is quite interesting that this network learned to not fire any output neurons if it should move forward instead of learning to fire the output neuron to take this action.

9.2 Robot Performance

In this section, how well the behavior exhibited in the simulator translated to the physical system is discussed. Furthermore, exploration is done to evaluate if networks may be ran on a more powerful host with communications occurring between this host and the robot.

9.2.1 Behavioral Performance

For each network, the observed behavior seemed to mostly match the behavior seen in the simulator. There were instances in which the robot seemed to have stronger responses in turning towards obstacles than those exhibited in the simulator, but other than this, the behavior seemed the same. Networks would alternate decisions of turning towards the target and progressing forward, such that the robot appeared to make sweeping turns in its pursuit of the target.

Furthermore, the behavior of the traditional algorithms nearly matched exactly. Using these algorithms, the robot would turn towards the target and then progress towards it, just as it did in simulation. These algorithms likely showed better matching behavior than the

behavior exhibited by the networks because when using the camera to estimate the position of the target, the accuracy of the estimation is increased by the target being directly in front of the robot. In fact, this estimation likely accounts for the majority of the discrepancies exhibited between the simulator and the physical system, with the only other contributing factor being the difference in motor responses.

9.2.2 Update Evaluation Time

In evaluating the ability of the robot to utilize a remote host, the most important factor is if the time it takes to communicate with the remote host and for that host to do the evaluation can fit within the real-time deadline enforced on the robot. For GRANT, this real-time deadline, as previously discussed, is 10 milliseconds. For these tests, 5000 update cycles were captured. During the 5000 update cycles, five network switches occurred, and, because sending the network to the remote host is the largest possible transmission that must occur, we evaluate the time it took to complete these cycles separately from the ordinary update cycles that simply transmit network fire information. These tests were ran with a high-level, shortened packet format that was translated by the remote host into full hardware packets.

In [Figure 9.13](#), one may see the performance of both Ethernet and WiFi communications running in conjunction with a remote host using either the network simulator, a large FPGA, or the SNACC system. The performance of the local system is also presented as a basis of comparison. Furthermore, one may see the time required during cycles in which a network reconfiguration occurred in [Figure 9.14](#). Upon inspecting these graphs, it becomes apparent that there is no real hindrance in using Ethernet instead of running the networks locally. Communication over Ethernet is just as fast as running locally for regular update cycles, and for those in which a network must be switched, running over Ethernet is actually faster. This is because the host can create the packets to send to the network much faster than the ARM core on the SoC can.

When attempting to run networks over WiFi, it can be seen that there may, in fact, be issues. While update cycles are performed fast enough on average, there are a sufficient number of outliers that take longer than the 10 milliseconds and raise some concern. This is exacerbated when looking at cycles in which the network must be switched, as these cycles

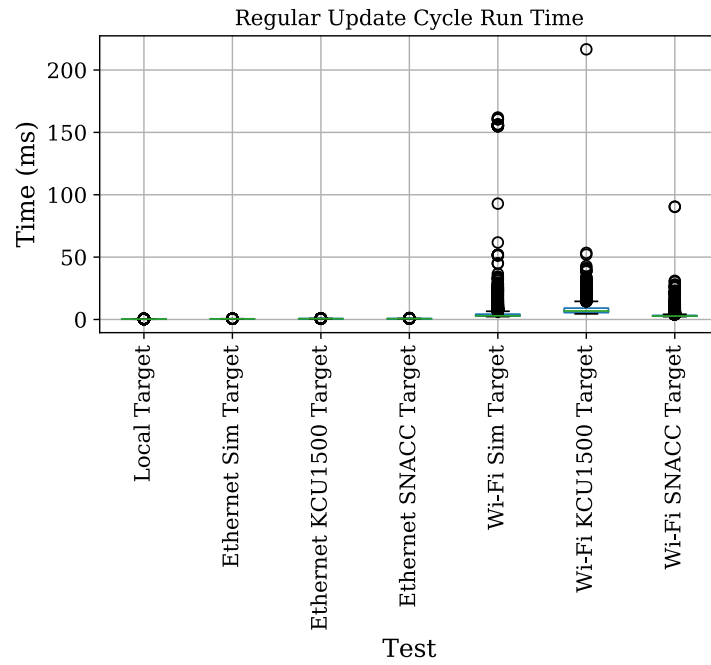


Figure 9.13: Update time for cycles without network switches or sensor updates

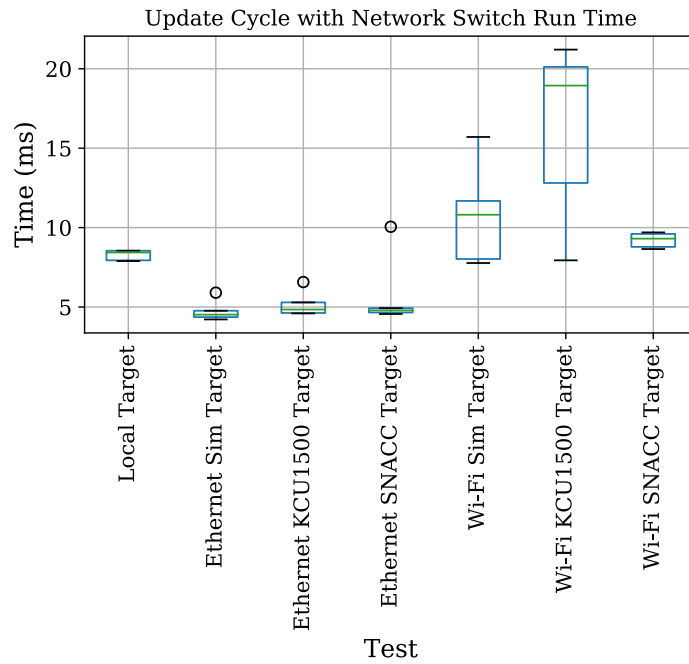


Figure 9.14: Update time for cycles with network switches

barely meet the requirement on average. This performance may be caused by issues with the wireless connection, as the outliers were generally grouped in short periods of time. Further testing would need to be done to determine if communication via WiFi is a feasible way to run the robot.

Chapter 10

Future Work

If you can't fly, then run. If you can't run, then walk. If you can't walk, then crawl, but by all means, keep moving.

– Martin Luther King Jr.

This robot, as a concept, was a logical progression from the implementation of NeoN, as it added the use of multiple networks to accomplish a single task and added the ability for the robot to pursue a target. Furthermore, GRANT also utilized the DANNA2 neuromorphic architecture, which was a logical progression from its predecessor, DANNA, that was used on NeoN. In this respect, there are multiple routes in which one could progress from GRANT, with these primarily involving the hardware and architecture, the usage of networks for multi-objective functions, and applications.

10.1 Hardware and Architecture

Perhaps the most obvious way in which one could progress from GRANT is to upgrade the hardware. One could certainly use a different set of sensors to increase scanning speed for the LIDAR and improve target recognition for the camera. Doing these things could potentially lead to better results and would certainly allow for more complex targets to be recognized. This could even be further extended to use a Dynamic Vision Sensor (DVS) camera. Such a camera could allow for the replacement of the LIDAR with a sensor that is constantly working

in real-time, although it would require modifications to the encoding of this information to the network.

Furthermore, improvement could also be made to the power requirements of GRANT. While using an FPGA is good for development, it is far from optimal in terms of a deployable, resource-constrained embedded system. In order to mitigate this issue, one could develop an Application Specific Integrated Circuit (ASIC) with a DANNA2 array. If the input encoding and output decoding were moved to being done in hardware, one could implement the entirety of the control circuitry on such an ASIC, such that the power consumption would be greatly reduced in comparison to a device with a general purpose CPU.

10.2 Running Networks Concurrently

In order to use different networks to perform different functions on the robot, the DANNA2 array is reconfigured when the robot believes the appropriate conditions are reached to switch. Instead of actually reconfiguring the array, one could instead run multiple networks concurrently within the hardware. For instance, utilizing a 30x10 DANNA2 array, one could run both the roaming and targeting networks stacked on top of one another with no interaction between the networks. In this way, one could simply select the desired network to receive output from without ever having to reconfigure the array. One could also utilize this as a method of behavior composition between multiple networks. By assigning a weight to the decision made by an individual network within a group of networks, one could compose the decisions and their corresponding weights to make the ultimate decision as to what action should be taken. For instance, in the above example with the roaming and targeting networks, one could weight the network outputs such that the roaming network's decision carries more weight and will thus avoid crashing into things regardless of the desire of the targeting network. These weightings could be treated as hyperparameters that are optimized with a grid search in the simulator itself, such that the composed behavior could be optimized for performance.

10.3 Target Interaction

On NeoN, the robot's goal was purely to explore an environment while maximizing its coverage. On GRANT, the robot's goal was to do this very same thing in order to find a target, and upon finding the target, pursue it. At the point at which GRANT has reached the target, the task has been completed. Just as GRANT added an additional component to the objective set by NeoN, a successor to GRANT may do the same. A robot could be trained to have some sort of interaction with the target upon reaching it. This could be anything, from moving the target to a new location to eliminating the target. Adding such an additional task would add another layer of complexity to the problem, and would be an effective test to further push the bounds of the application of these neuromorphic systems.

10.4 Navigating a 3D Environment

If one did not wish to increase the complexity through adding a task with target interaction, one may desire to increase the problem's complexity by adding an additional dimension to travel along. Through the creation of a drone, one could force the RSNN to navigate a 3D environment. The additional degrees of freedom would allow more room for exploration with evolutionary algorithms, but it could also cause the algorithms to never develop any promising results. Ideally a simplified control scheme would be used, such that the network could choose to go up, down, left, right, forwards, backwards, or some combination of these directions. In this way, the network would not have to learn how to control the flight itself, but could rather simply learn how to navigate the environment.

This, of course, would come with a myriad of challenges. A larger FPGA and more advanced sensors would be required in order to give enough information to a network and actually implement that network in hardware. Furthermore, failure cases could be substantially more devastating. In the event that GRANT fails to perform the intended action, the robot may run into a chair leg or a wall, causing nothing more than potential cosmetic damage; a drone crash, on the other hand, could be devastating, as it could cause

serious damage to the robot itself. Lastly, modeling the environment properly and accounting for all of the necessary physics could prove to be incredibly difficult.

Chapter 11

Conclusion

It always seems impossible until it's done.

– Nelson Mandela

This work serves as the first embedded neuromorphic system that utilizes the DANNA2 neuromorphic architecture. It also shows the ability of networks compatible with this architecture to pursue a target while avoiding obstacles, and that these networks are capable of operating with less information than the traditional algorithms. Furthermore, it shows the ability of these networks to severely outperform the traditional algorithms when the traditional algorithms are given the limited information offered to the networks. This work also shows the most successful iteration of a self-contained, targeting robot using neuromorphic computing to date.

With the use of these small, fast networks utilizing the DANNA2 architecture, a tremendous power-savings could also be achieved over traditional algorithms by using an ASIC for the DANNA2 grid array. Coupling this with the grid array's ability to allow real-time switching of networks for multi-objective functions means that a complex task could be completed using a minimal amount of resources. This dynamic switching of networks expands the horizons of what is achievable in a self-contained robotic platform, allowing small networks to achieve parts of a multi-objective function.

Bibliography

- [1] F. Akopyan et al. “TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (Oct. 2015), pp. 1537–1557. ISSN: 0278-0070. DOI: [10.1109/TCAD.2015.2474396](https://doi.org/10.1109/TCAD.2015.2474396) (cit. on p. 6).
- [2] George A Bekey and Kenneth Y Goldberg. *Neural networks in robotics*. Vol. 202. Springer Science & Business Media, 2012 (cit. on p. 1).
- [3] Buddy Bland. “Titan-early experience with the titan system at oak ridge national laboratory”. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE. 2012, pp. 2189–2211 (cit. on p. 13).
- [4] Sam Bobrowicz et al. *Arty-Z7-20-base-linux*. 2018. URL: <https://github.com/Digilent/Arty-Z7-20-base-linux> (cit. on p. 33).
- [5] Mariusz Bojarski et al. “End to End Learning for Self-Driving Cars”. In: *CoRR* abs/1604.07316 (2016). arXiv: [1604.07316](https://arxiv.org/abs/1604.07316). URL: <http://arxiv.org/abs/1604.07316> (cit. on p. 1).
- [6] Alberto Elfes. “Sonar-based real-world mapping and navigation”. In: *IEEE Journal on Robotics and Automation* 3.3 (1987), pp. 249–265 (cit. on pp. 3–5).
- [7] K. D. Fischl et al. “Neuromorphic self-driving robot with retinomorphic vision and spike-based processing/closed-loop control”. In: *2017 51st Annual Conference on Information Sciences and Systems (CISS)*. Mar. 2017, pp. 1–6. DOI: [10.1109/CISS.2017.7926179](https://doi.org/10.1109/CISS.2017.7926179) (cit. on p. 1).
- [8] Tony Huang et al. *rplidar_sdk*. 2019. URL: https://github.com/slamtec/rplidar_sdk (cit. on p. 39).
- [9] Tiffany Hwu et al. “A self-driving robot using deep convolutional neural networks on neuromorphic hardware”. In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 635–641 (cit. on pp. 7, 8).
- [10] Scott Larson. *Quadrature Decoder (VHDL)*. 2017. URL: <https://www.digikey.com/eewiki/pages/viewpage.action?pageId=62259228> (cit. on p. 34).

- [11] Rich LeGrand. *pixy2*. 2019. URL: <https://github.com/charmedlabs/pixy2> (cit. on p. 39).
- [12] J. J. Leonard and H. F. Durrant-Whyte. “Simultaneous map building and localization for an autonomous mobile robot”. In: *Proceedings IROS '91:IEEE/RSJ International Workshop on Intelligent Robots and Systems '91*. 1991, 1442–1447 vol.3 (cit. on p. 2).
- [13] Rongbing Li et al. “LIDAR/MEMS IMU integrated navigation (SLAM) method for a small UAV in indoor environments”. In: *2014 DGON Inertial Sensors and Systems (ISS)*. IEEE. 2014, pp. 1–15 (cit. on pp. 5, 6).
- [14] Weibo Liu et al. “A survey of deep neural network architectures and their applications”. In: *Neurocomputing* 234 (2017), pp. 11–26 (cit. on p. 1).
- [15] Niels Lohmann. *JSON for Modern C++*. 2019. URL: <https://github.com/nlohmann/json> (cit. on p. 39).
- [16] *Lynxmotion Aluminum A4WD1 Chassis - RobotShop*. URL: <https://www.robotshop.com/en/lynxmotion-aluminum-chassis-a4wd1.html> (cit. on p. 25).
- [17] P. Merolla et al. “A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm”. In: *2011 IEEE Custom Integrated Circuits Conference (CICC)*. Sept. 2011, pp. 1–4. DOI: [10.1109/CICC.2011.6055294](https://doi.org/10.1109/CICC.2011.6055294) (cit. on p. 15).
- [18] *mfoo/Math-Library-Test/mother.cpp at master*. URL: <https://github.com/mfoo/Math-Library-Test/blob/master/src/mother.cpp> (cit. on p. 39).
- [19] Moritz B Milde et al. “Obstacle avoidance and target acquisition for robot navigation using a mixed signal analog/digital neuromorphic processing system”. In: *Frontiers in neurorobotics* 11 (2017), p. 28 (cit. on pp. 8, 9).
- [20] J. P. Mitchell et al. “DANNA 2: Dynamic Adaptive Neural Network Arrays”. In: *International Conference on Neuromorphic Computing Systems*. Knoxville, TN: ACM, July 2018. DOI: [10.1145/3229884.3229894](https://doi.org/10.1145/3229884.3229894) (cit. on pp. 14, 16, 36, 39, 46, 47).
- [21] J. P. Mitchell et al. “NeoN: Neuromorphic Control for Autonomous Robotic Navigation”. In: *IEEE 5th International Symposium on Robotics and Intelligent Sensors*. Ottawa, Canada, Oct. 2017, pp. 136–142. DOI: [test](https://doi.org/10.1109/RSIS.2017.8000000) (cit. on pp. 2, 12–14, 23, 45, 47, 71).

- [22] John Parker Mitchell. “DANNA2: Dynamic Adaptive Neural Network Arrays”. MA thesis. The University of Tennessee, Aug. 2018 (cit. on p. 17).
- [23] Mitchell Orsucci et al. *Petalinux-Arty-Z7-20*. 2018. URL: <https://github.com/Digilent/Petalinux-Arty-Z7-20> (cit. on p. 37).
- [24] Brandon Perez et al. *xilinx_axidma*. 2018. URL: https://github.com/bperez77/xilinx_axidma (cit. on p. 39).
- [25] *PetaLinux Tools*. URL: <https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html> (cit. on p. 37).
- [26] *Pixy2 – PixyCam*. URL: <https://pixycam.com/pixy2/> (cit. on p. 24).
- [27] J. S. Plank et al. “A Unified Hardware/Software Co-Design Framework for Neuromorphic Computing Devices and Applications”. In: *IEEE International Conference on Rebooting Computing (ICRC 2017)*. Washington, DC, Nov. 2017 (cit. on p. 11).
- [28] J. S. Plank et al. “The TENNLab Exploratory Neuromorphic Computing Framework”. In: *IEEE Letters of the Computer Society* 1.2 (July 2018), pp. 17–20. DOI: [10.1109/LOCS.2018.2885976](https://doi.ieeecomputersociety.org/10.1109/LOCS.2018.2885976). URL: <https://doi.ieeecomputersociety.org/10.1109/LOCS.2018.2885976> (cit. on pp. 10, 11).
- [29] *Pololu - 30:1 Metal Gearmotor 37Dx68L mm 12V with 64 CPR Encoder (Spur Pinion)*. URL: <https://www.pololu.com/product/2823> (cit. on p. 25).
- [30] *Pololu 5V, 9A Step-Down Voltage Regulator D24V90F5*. URL: <https://www.pololu.com/product/2866> (cit. on p. 67).
- [31] *PYNQ-Z1 [Reference.Digilentinc]*. URL: <https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start> (cit. on p. 21).
- [32] J. J. M. Reynolds, J. S. Plank, and C. D. Schuman. “Intelligent Reservoir Generation for Liquid State Machines using Evolutionary Optimization”. In: *IJCNN: The International Joint Conference on Neural Networks*. Budapest, 2019, pp. 1–8. DOI: [10.1109/IJCNN.2019.8852472](https://doi.org/10.1109/IJCNN.2019.8852472) (cit. on p. 11).
- [33] *RPLIDAR-A2 Laser Range Scanner_ Solid Laser Range Scanner/SLAMTEC*. URL: <https://www.slamtec.com/en/Lidar/A2> (cit. on p. 23).

- [34] C. D. Schuman et al. “An Evolutionary Optimization Framework for Neural Networks and Neuromorphic Architectures”. In: *International Joint Conference on Neural Networks*. Vancouver, July 2016 (cit. on pp. 12, 51).
- [35] W. Severa et al. “Training Deep Neural Networks for Binary Communication with the Whetstone Method”. In: *Nature Machine Intelligence* 1 (Jan. 2019), pp. 86–94. URL: <https://doi.org/10.1038/s42256-018-0015-y> (cit. on p. 11).
- [36] S. B. Shrestha and G. Orchard. “SLAYER: Spike Layer Error Reassignment in Time”. In: *Advances in Neural Information Processing Systems* 31. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 1412–1421. URL: <http://papers.nips.cc/paper/7415-slayer-spike-layer-error-reassignment-in-time.pdf> (cit. on p. 11).
- [37] Richard B Stein. “Some models of neuronal variability”. In: *Biophysical journal* 7.1 (1967), pp. 37–68 (cit. on p. 17).
- [38] *Traxxas 7600mAh 2S 7.4V 25C iD LiPo Battery Pack*. URL: <https://www.rcsuperstore.com/traxxas-7600mah-2s-7-4v-25c-id-lipo-battery-pack/> (cit. on p. 26).
- [39] *Valgrind Home*. URL: <https://valgrind.org/> (cit. on p. 37).
- [40] Mark CW Van Rossum, Guo Qiang Bi, and Gina G Turrigiano. “Stable Hebbian learning from spike timing-dependent plasticity”. In: *Journal of neuroscience* 20.23 (2000), pp. 8812–8821 (cit. on p. 17).
- [41] Jilles Vreeken. *Spiking neural networks, an introduction*. 2003 (cit. on p. 1).
- [42] Aaron Reed Young. “SNACC: The Scaled-up Neuromorphic Array Communications Controller”. PhD thesis. University of Tennessee, 2020 (cit. on pp. 69, 70).

Appendix

A Wiring Schematic

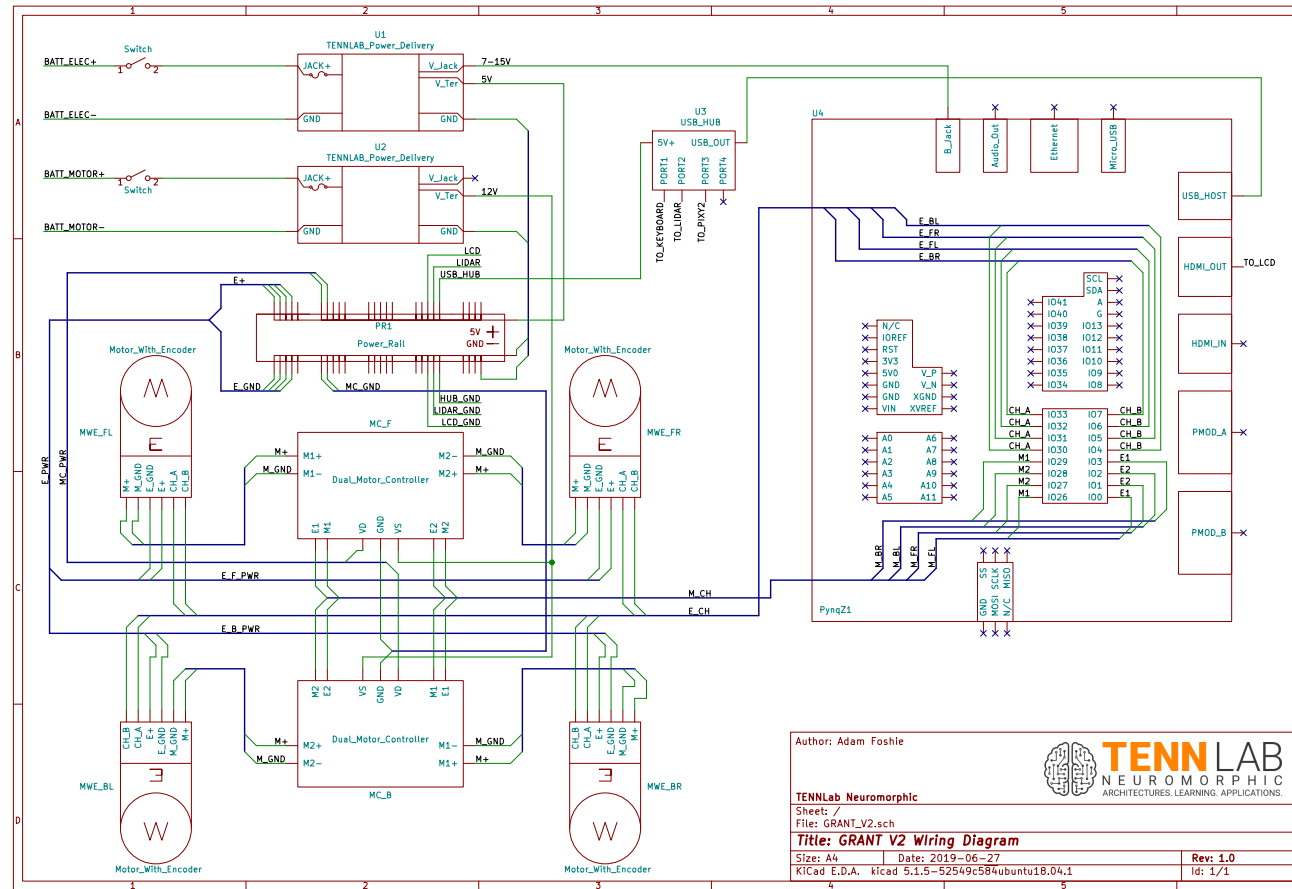


Figure A.1: Wiring schematic for a GRANT robot

B Bill of Materials

Table B.1: Bill of Materials

Item	Website	Part #	Total Price
Dual Motor Controller	https://www.robotshop.com/en/dfrobot-4-8-46v-2a-dual-motor-controller.html	RB-Dfr-19	\$34.10
7.4V 5800 mAH Battery Pack	https://www.rcsuperstore.com/Traxxas-5800mAh-2S-2-Cell-iD-LiPo-Battery-p/tra2843x.htm	TRA2843X	\$119.90
HDMI Screen	https://www.adafruit.com/product/2406	2406	\$79.95
Pynq Z1 FPGA	https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/	6003-410-017	\$199.99
Robot Platform	https://www.robotshop.com/en/lynxmotion-aluminum-chassis-a4wd1.html	RB-Lyn-399	\$87.15
CMU Pixy2	https://www.amazon.com/Pixy2-Smart-Vision-Sensor-BeagleBone/dp/B07D1CLYD2/	B07D1CLYD2	\$59.90
Pololu 12V, 30:1 Gear Motor w/ 64 CPR Encoder	https://www.robotshop.com/en/pololu-12v-301-gear-motor-64-cpr-encoder.html	RB-Pol-336	\$159.80
Sweeping Lidar	https://www.robotshop.com/en/rplidar-a2m8-360-laser-scanner.html	RB-Rpk-02	\$319.00
Omniwheels	https://www.robotshop.com/en/vex-robotics-large-omni-directional-wheel-kit.html	RB-Inn-25	\$49.98

Vita

Jonathan Daniel Ambrose is from Clinton, Tennessee. He graduated from Clinton High School May 2015 with a 4.0 GPA. He enrolled at the University of Tennessee - Knoxville in August 2015 seeking a Bachelors of Science in Computer Engineering from the Department of Electrical Engineering and Computer Science. He obtained this degree in December 2018, and went on to pursue his Masters of Science in Computer Engineering. During his time as an undergraduate, he interned as a Help Desk Triage Specialist with Mesa Associates Inc. and worked as an undergraduate research assistant for Dr. Mark Dean. He continued with his research position into graduate school working for Dr. Dean. Jonathan received his Masters of Science in Computer Engineering in May 2020.