

To the Graduate Council:

I am submitting herewith a thesis written by Andrew Friend entitled "Creation of a Neuromorphic Collective." I have examined the final paper copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dr. Catherine Schuman, Major

Professor

We have read this thesis
and recommend its acceptance:

Committee Member 1

Committee Member 2

Committee Member 3

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

To the Graduate Council:

I am submitting herewith a thesis written by Andrew Friend entitled "Creation of a Neuromorphic Collective." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dr. Catherine Schuman, Major Professor

We have read this thesis
and recommend its acceptance:

Committee Member 1

Committee Member 2

Committee Member 3

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Creation of a Neuromorphic Collective

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Andrew Friend

August 2025

© by Andrew Friend, 2025
All Rights Reserved.

Abstract

The focus of my research has been in emergent behavior, especially as it relates to robotic systems. Within this work I discuss three aspects of this research, simulation, hardware, and embedded systems, each of which had use in swarm behavior, and the latter two have applications outside of this niche as well.

The simulation was a swarming application on the TENNlab framework. The best networks were manually reviewed to determine how accurately they completed the task and therefore how faithfully the fitness function captured the intent of the program. Testing those functions was one of the largest parts of this project. Due to being such an open-ended project the performance is difficult to summarize, but the networks developed well in simulation and translated fairly easily into hardware.

The hardware began as a dedicated platform for the simulation but became its own project with the goal of being easy to use and repurpose. Its success was measured primarily in binaries of whether or not certain features worked as intended. The robot was found capable of precise movement and easily applicable to a range of projects.

The embedded system was not integrated into the other two projects but came about as a result of working with the tools available in the lab. It is a variation of the neuromorphic kit where the encoding and decoding units are combined onto a single processor. I measured the speed, power, energy, and variability of use when scoring the results of this variation. In each of these categories my kit matched or exceeded the baseline set by the original. Some of the changes could also be implemented into the original with little effort.

Table of Contents

1	Introduction	1
1.1	Swarm Application	2
1.2	SALMON Platform	2
1.3	Two Board Kit	2
2	Related Works	4
2.1	Neuromorphic Swarm	4
2.1.1	Swarm Optimization	4
2.1.2	Milling	5
2.2	TENNlab Framework	5
2.2.1	RISP	5
2.2.2	EONS	6
2.3	Neuromorphic Starter Kit	7
2.4	Fishschool Project	7
2.5	X-Drive Project	9
3	Swarm Application	11
3.1	Setup	11
3.1.1	Goals	11
3.1.2	Constraints	11
3.2	Methods	12
3.2.1	Application Details	12

3.2.2	Fitness Functions	17
3.2.3	Network Evaluation	22
3.2.4	Hardware	23
3.3	Results	26
3.3.1	Application Details	26
3.3.2	Fitness Functions	29
3.3.3	Network Evaluation	36
3.3.4	Hardware	36
4	Swarming And Learning Machine Operating on Neuroprocessors	40
4.1	Setup	40
4.1.1	Goals	40
4.1.2	Constraints	40
4.2	Methods	42
4.2.1	Locomotion	42
4.2.2	Chassis	46
4.2.3	Sensor Suite	49
4.2.4	Power Distribution Board	50
4.3	Results	54
4.3.1	Locomotion	55
4.3.2	Chassis	58
4.3.3	Sensor Suite	59
4.3.4	Power Distribution Board	60
5	Two Board Neuromorphic Kit	63
5.1	Setup	63
5.1.1	Goals	63
5.1.2	Constraints	64
5.2	Methods	64
5.2.1	Code Combination	64

5.2.2	Communications	65
5.2.3	Repository Refactor	67
5.2.4	Hardware	70
5.3	Results	73
5.3.1	Code Combination	73
5.3.2	Communications	76
5.3.3	Repository Refactor	77
5.3.4	Hardware	77
6	Conclusions	79
6.1	Summary of Collaborations	79
6.2	Conclusions and Contribution	80
6.3	Discussion	81
6.3.1	Swarm Application	81
6.3.2	SALMON	81
6.3.3	Two Board Neuromorphic Kit	82
6.4	Future Work	82
6.4.1	Swarm Application	82
6.4.2	SALMON	83
6.4.3	Two Board Neuromorphic Kit	84
	Bibliography	86
	Vita	88

List of Tables

4.1	SALMON Component Prices	56
4.2	SALMON Print Weights	56
4.3	Truth Table of Motor Drivers	62
5.1	Electrical Specification Comparison of Kits	74
5.2	File Sizes	74

List of Figures

2.1	Neuromorphic Kit with Three Raspberry Pi Picos	8
2.2	Fishschool Visualization	8
2.3	VEX X-drive Chassis	10
3.1	Example of Swarm Application Input	15
3.2	Example of Goal Region	20
3.3	Example of Exploration Zones	21
3.4	Physical Testbed for IR Friend Detection	25
3.5	Networks Trained with Movement Counting	31
3.6	Networks Trained with Naive Multi-Objective	33
3.7	Networks Trained with Complex Multi-Objective	33
3.8	Networks Trained with Exploration Separation	35
3.9	Networks Trained with Coverage	37
3.10	Robots in Testing Space	39
4.1	Components of Motor Assemblies	44
4.2	View of Robot Drive Train	45
4.3	Alternate Angles of Robot Chassis	48
4.4	Circuit Diagram of Power Distribution Board	51
5.1	File Tree of Each Kit Version	68
5.2	Two Board Kit Stacks	72
6.1	PCB Design of Power Distribution Board with Two Board Kit	85

Chapter 1

Introduction

Swarm intelligence is the concept of taking many simple actors, all operating on relatively a simple set of rules, and observing complex collective behavior when viewing them at scale. This behavior is called emergent behavior, because it emerges as a result of the interactions between the individuals, rather than being hardwired into them. Swarming robotics is the practice of applying those principles to robots that are also individually simple. Neuromorphic computing is interesting for the field of swarming robotics due to both being based in biological systems, as mentioned in Section [2.1.1](#), but also because of the focus on event-driven behavior, which limits the power and processing needs of the system in a way that is helpful for robotics.

This combination was also appealing due to both fields distinct focus on minimalism and interaction. A spiking neural network performs activity only when interacted with, and swarming agents have minimal individual ability and processing, which become relevant when interacting with each other at scale. To accomplish this project there were three primary sections of work to be done. The first is a method to create a network that could be placed on the agents to allow them to process their surroundings. The second section is a hardware platform that lets the agent observe and act within its room. To combine these sections there must be a processor that can

be deployed onto the hardware that is capable of hosting one of the trained networks, this is the third section of the project.

1.1 Swarm Application

For the software side, this project prompted the creation of the swarm application category for the framework by Dr. Plank, and was the first, and so far only, non-example project to use it. The application created identical agents within a room with the goal of exploring the room together. Though the application itself was relatively simple to program, it required much testing to find ways of accurately evaluating the fitness of networks and to find an environment that was simple enough to learn in but complex enough to prepare agents for hardware translation.

1.2 SALMON Platform

Our lab has a history of creating dedicated hardware for projects that can be showcased in a demo. This leads to specialized robots that are hard to reuse and difficult to perform maintenance on once the students that were part of the original project have left. Requiring new robots to be built for each project that wants to have a hardware aspect can discourage people from pursuing this type of project. For that reason, one of the goals of this project was to make a hardware platform that is easy to reuse and recreate. This turned into the project in [Chapter 4](#).

1.3 Two Board Kit

One aspect of an easy-to-use robot is to standardize the interface. Many of the lab's previous robots use Single Board Computers, such as SABR, or FPGAs, such as GRANT and NeoN. However, the lab has recently begun pushing for the use of the neuromorphic kit detailed in [Section 2.3](#) for hardware applications. As of

the beginning of this project, the kit had not been utilized much outside of the example projects. It was used for this project because it presents a clear workflow for exporting networks from the framework to hardware, as opposed to everyone using their own method on a project-to-project basis. While using the kit for the development of the robot, I began work on a side project of developing an alternative kit that became Chapter 5. It holds promise as an alternative option for projects that want a condensed version of the kit, but some of the changes implemented could be used in the primary workflow to better the experience of users.

Chapter 2

Related Works

2.1 Neuromorphic Swarm

Combining swarm intelligence with neuromorphic computing is not a new concept. In the field of neuromorphic computing as a whole, there have been a few projects that take on challenges similar to what will be done in this paper. Two examples of this are listed below.

2.1.1 Swarm Optimization

A swarm of SNNs was used in [Fang et al. \(2019\)](#) to optimize the traveling salesman problem, among other common optimization problems. Both aspects of the SI-SNN(Swarm Intelligence-Spiking Neural Networks) models, as they call them, were chosen for their roots in biological systems. The goal was just to prove the feasibility of connecting the two systems, which they accomplished by showing that the traveling salesman problem was solved to near-optimal solutions. The authors found two tradeoffs in their hyperparameters, one between spacial and temporal size, and the another between speed and accuracy. This project also included deployment to hardware, granted in a very different context than we will be doing.

2.1.2 Milling

A more clear comparison for the project in Chapter 3 is [Zhu et al. \(2024\)](#). In that study, TENNlab’s EONS was applied to a problem similar to our own. This experiment utilized a binary sensor to inform the network, whose goal was to create a milling circle. This paper was successful in its goal at the simulation level, with hopes of moving to hardware in the future. While this project set the goal of a simpler and more well-defined task, their success at creating emergent robotic behavior from a neuromorphic swarm gave us confidence for our foray into the space.

2.2 TENNlab Framework

In this work, we use the TENNlab neuromorphic computing software framework [Plank et al. \(2018\)](#). This framework defines a network object and a processor object that provide common interfaces that can be used with different applications, algorithms, or neuromorphic hardware backends. The TENNlab framework is used for the development of a custom swarm application in Chapter 3 as well as the benchmark networks in Section 5.3.1. The TENNlab framework provides many options for spike encoding and decoding [Schuman et al. \(2019\)](#). The exact encoding and decoding schemes used in the swarming application are discussed in Section 3.2.1.

2.2.1 RISP

The neuromorphic hardware implementation used for all networks of this paper is the Reduced Instruction Spiking Processor, or RISP, implementation [Plank et al. \(2022b\)](#). RISP implements simple leaky integrate and fire neurons as well as synapses. The former’s leak and threshold and the latter’s weight and delay are all programmable. Synapse delays are always defined in positive integer values, whereas weights and thresholds can be either discrete or continuous. Neuron leak can be only turned on or off. If leak is enabled, then at the end of each integration time step, if the neuron

does not fire, the neuron’s charge is reset to zero. If leak is disabled, the neuron retains its charge for the next integration time step. In this work, we use the discrete version of RISP, so all weights and thresholds are integers.

2.2.2 EONS

Baseline EONS

To design spiking neural networks for RISP for the swarm application, we use Evolutionary Optimization for Neuromorphic Systems (EONS) [Schuman et al. \(2020\)](#). EONS uses an evolutionary algorithm to design the structure and parameters of a spiking neural network to fit the requirements of a task. After a population is randomly generated, with the only fixed items being the number of input and output neurons, they are placed in the simulation and evaluated using a fitness function. In the baseline version of EONS, this function returns a single value. Using tournament selection, networks are chosen from among the population to serve as the parents of the next generation.

To create the next generation, EONS first places the parents into the generation, then uses cloning, mutation, and crossover to create networks that are combinations of and alterations on the parents. This fills out the generation, which repeats the process of being tested, graded, and replaced by the next generation. The parents being placed into the next generation unchanged also helps to prevent backsliding, as there will be at least a few networks that are as capable as the last generation, even if all of the children are worse.

Multiobjective EONS

Recently, EONS has been further extended to support multiobjective optimization. This extension is based on the Adaptive Geometric Estimation Multi Objective Evolutionary Algorithm, AGE-MOEA-II [Panichella \(2022\)](#). In that work, as well as multiobjective EONS, networks are first chosen based on their non-dominated

rank. Ties are broken by choosing networks that are measured to be more dispersed. As with AGE-MOEA-II, dispersion is measured via the geodesic distance on a unit ball created by normalizing the fitnesses within $[0, 1]$ and finding the L_2 distance. By utilizing dispersion as a selection characteristic more varied approaches continue to propagate through generations instead of converging early.

Swarm EONS

Swarm applications in EONS work in much the same way as an ordinary application, with the caveat that each member of the population in an epoch is a single network copied to each of N agents that exist in the same environment, interacting with each other. As such, the fitness function for these applications must take into account all choices made in the environment and all effects of those choices. This variation of EONS is what is used in Chapter 3.

2.3 Neuromorphic Starter Kit

The hardware implementation for each of the projects was done in the context of the TENNLab neuromorphic starter kit [Plank et al. \(2022a\)](#) (Fig. 2.1). This kit is made of a custom printed circuit board, PCB, that acts as a carrier for three components; an input-to-spike encoding unit, a neuromorphic processor unit, and a spike-to-output decoding unit. While the neuroprocessor can be implemented using FPGAs, SBCs, or custom neuromorphic hardware, we utilized the base case of a microcontroller, specifically the Raspberry Pi Pico, two of which are also used for the encoding and decoding units.

2.4 Fishschool Project

For a class project, Samara Buchanan and Louise Fry created an early version of the swarming app that will be discussed in Chapter 3.



Figure 2.1: Neuromorphic Kit with Three Raspberry Pi Picos

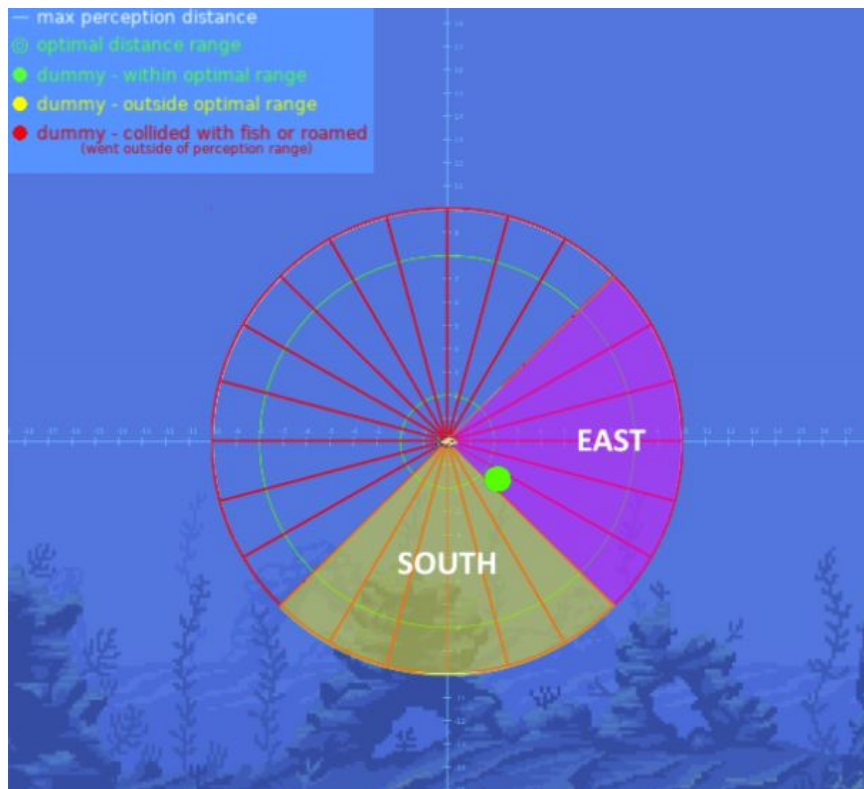


Figure 2.2: Fishschool Visualization

This project used the TENNlab framework with a single agent trained to move with a “dummy” that followed pre-programmed rules to move semi-randomly. The network agent received LiDAR inputs, downsampled to the nearest per quadrant, and could choose to move in the four cardinal directions at either a low or high speed. In this simulation, the dummy body is the only visible object in the space, and the view of the visualization is centered on the agent, eliminating the need for boundary walls. This can be seen in Fig. 2.2.

This project found high success rates in terms of the agent following the dummy at an optimal distance. Hyperparameter tuning also revealed that the agents were much more competent and suited for their tasks at slow speeds, when each movement only slightly affected the positions of the bodies. They also found that the sparse reward environment found when penalizing unwanted behavior harshly was worse for training than a smaller penalty that allowed room for slow incremental growth.

2.5 X-Drive Project

In the same class as Section 2.4 was developed, I was part of another project focused on teaching a robot to explore a room via holonomic driving. The robot used was very similar to what the goal would become for Chapter 4.

The project consisted of using NeuroEvolution of Augmenting Topologies, or NEAT, to train a neural network. This network would be given a distance and angle from a desired goal location as its input, and give four individual motor powers as its output, which were then taken and converted into movement for the timestep. On the hardware side, we used a simple X-drive chassis, Fig. 2.3, that was developed by the members of my project as part of team YNOT for the VEXU robotics competition.

This project was a complete success at the simulation level, but the units used in simulation did not translate well to the speed at which the hardware could process sensory input. The robot did move in the correct direction but it rarely moved the correct distance.

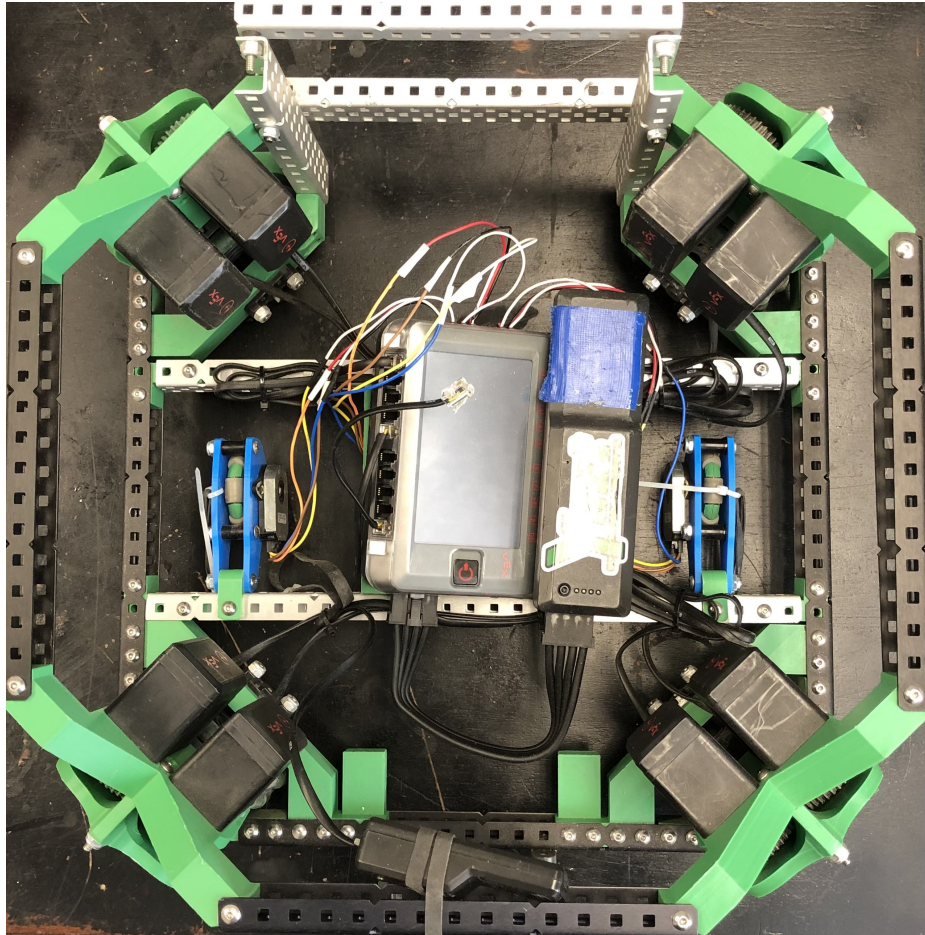


Figure 2.3: VEX X-drive Chassis

Chapter 3

Swarm Application

3.1 Setup

As mentioned in Chapter 1, much of the content of this chapter comes from a paper that is being published at IJCNN. That paper had a more focused dissection of the fitness functions and did not discuss as heavily the aspects of the project, especially in hardware, that didn't make it to the finished product.

3.1.1 Goals

The primary goal of this project was to continue the work performed in Louise's class project from Section 2.4 by expanding the training to multiple interacting agents and by exporting the networks to a physical robot that can interact in the real world. This plan lead into the development of the mostly independent project from Chapter 4 and the general goal of the application, to create a network that is able to work collaboratively with copies of itself to explore a room.

3.1.2 Constraints

Before any programming was performed we had decided on certain things that should be true of our project. The first is that the agents only have local information. We

didn't want any communication between agents or any globally true information such as an agreed upon "forward" that is agnostic of the direction a robot is facing. This was decided in order to keep agents simple and to increase the versatility of the system. In a situation where robots do not need to communicate, the swarm is not bounded by a size limit determined by the communications protocol limitations, and by not including any objective information they could be deployed to a wider variety of environments with less thought given to the starting conditions.

The second constraint was to ensure that anything we programmed could be replicated in hardware. We knew that movement would be covered under the robot that was made in Chapter 4, but the sensors were deemed a part of this project. That's because the sensor array is application specific, rather than part of the core robot, which is the focus of that project.

3.2 Methods

3.2.1 Application Details

Environment

The environment for this application is a rectangle representing a room viewed from the ceiling, looking at the floor. For all of the primary testing and evaluation, the room was square with the dimensions of twenty-five by twenty-five units, but our goal for when we discovered a fitness function that could handle it was to have the room generate with random dimensions between fifteen and fifty units. A unit in the simulation is equivalent to the radius of the robot, for simplicity of programming, which equates to approximately three and a half inches.

Within the room are numerous solid objects. In this chapter, solid will refer to an object that is visible to the LiDAR used by the agents, will prevent an agent from attempting to move into space already occupied by it, and will note a collision in that

scenario. The relevance of what a collision means to the simulation is discussed in Section 3.2.2.

The room is bounded by four retainer walls that completely envelope the space. These walls are solid and were implemented to train agents to be used to existing in a closed space, meaning that in order to explore more space they would have to turn and move in directions other than that which they were created facing. They also serve the secondary purpose of confining the agents to an easily observed area for the purposes of evaluating their behavior.

Some testing of the application involved creating a more complex environment, which took the form of obstacles and the “treadmill”. For the former, at the beginning of an episode we generated between one and three random objects within the room. Each object was solid and had a footprint of three to five contiguous square units. The second complication, the “treadmill”, was not an object in the room. Instead, at the beginning of each episode a random direction would be chosen from the list of “up, down, left, right, none”. Each timestep thereafter, all agents would be moved a single unit in that direction in addition to their normal movement. The motivation behind these additions was to increase the resiliency of the networks by introducing them to more varied environments.

Of course, the room would be incomplete without agents to act within it. Each agent is a solid circle with a radius of one unit. It has a LiDAR located at its center that can see solid objects. The agents are created within the room with a randomized angle and a pseudo-randomized position that held to the following constraints

1. Agents could not be created within the outer margin of the room, which occupied ten percent of its area on each side.
2. Agents could not be created overlapping with another solid object.

Observations

The input space for these agents was confined to something very simple because of the end goal of the project, to run it on the neuromorphic kit. The limited processing power of the Raspberry Pi Pico meant that a camera would likely be far too computationally expensive, so we decided to use a basic two-dimensional, 360° LiDAR. The input space for these networks is a list of distances, each corresponding to the minimum distance seen by the LiDAR in a “slice” of the agent’s field of view, see Fig. 3.1. The down sampling to only one distance per slice was a concession that we made due to the limitations of EONS, which can normally only handle approximately twenty inputs. This is reminiscent of the work in Section 2.4. We wanted to minimize preprocessing to really put emphasis on the idea of the agents being extremely simple individually. There were a few variations of this input space that we experimented, listed below.

1. 360° field of view
2. 90° field of view centered on the front of the robot
3. Slices split with a division at 0°
4. Slices split with the center of a slice at 0°
5. A subset of five consecutive readings for each slice, centered on the nearest reading seen within that slice, in place of a single distance

The LiDAR in the simulation works inverse of how one would expect, where a higher value corresponds to a closer object. This is because it minimizes the amount of activity from the input neurons when in the expected state of being relatively far from all objects, and increases activity when the agent is close to something, which should prompt action as the neuron lights up more and more. You can see how this is calculated via the equation below. The max distance of the simulation and hardware is forty five units, or about four meters.

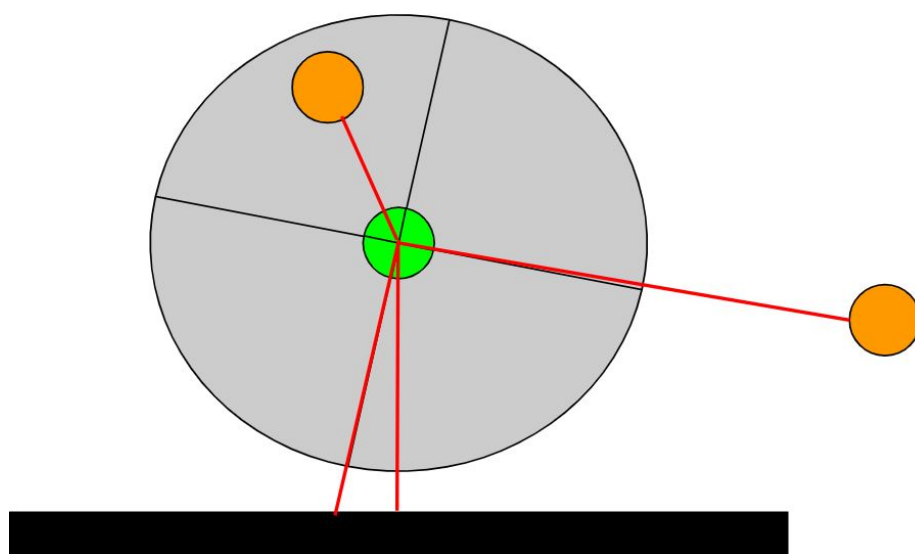


Figure 3.1: Example of Swarm Application Input

$$output = 1 - \frac{dist_{observed}}{dist_{max}}$$

Additionally, we experimented with something we called “friend detection”. This would allow agents to tell where the other agents were by adding a boolean input neuron for each slice that would be active when an agent was present in that direction and inactive when not.

Actions

In the simulation, each timestep agents choose an action out of forward, left, right, backward and a speed of either one or two. The former list is based on the agent’s angle, not in regard to the “top” of the room as observed by users. There were two versions of how this output would be utilized in the simulation, which have to do with the handling of the action. For both versions of the output scheme, the speed is a multiplier for the action, and if high speed, or two, was selected, it would have the same effect as if the agent chose the low speed version of that action twice. This could improve the amount that an agent could accomplish during an episode and gave them the choice between safely traveling slow and risking faster movement to maximize reward. A single driving action would move the robot by one unit.

The first type of output was holonomic driving. Holonomic driving allows for movement in any direction and, if we were to add complexity to the experiment, spin while driving. This was chosen initially because it is the simplest to implement from a top-down perspective. This method was exclusively used with a 360° field of view, as with holonomic movement there is no reason for a specific angle of the agent to be favored as the “front” for reasons other than communicating direction.

The second method of moving the robot was a more traditional driving scheme that involves driving forward and backward, and pivoting left and right. We moved to this as the primary development front because we found that the holonomic all-seeing agents seemed to get overwhelmed with the surplus of information. Turns in

this output scheme rotate either fifteen or thirty degrees depending on the speed. We tested this method of control with a variety of action subsets. All variations could drive forward and turn left, but also we tested versions that could not drive in reverse and/or turn right. These limitations were both instituted after we observed that the agents seem to regularly choose two opposite commands and alternate between them to avoid ever colliding.

Encoding and Decoding

The encoding style for this application was a spike train encoder with each input neuron’s maximum and minimum input values set to one and zero respectively and a time interval of fifty. This type of encoding takes the input of the LiDAR, a value from zero to one where zero is the edge of visibility and one means that the objects occupy the same space, and spikes $interval * input$ times. In tests with friend detection, there would be an extra encoder in this style for each section that functioned exactly the same, but instead of the LiDAR input it would have either a zero or a one depending on if an agent was detected.

Both output schemes used the same decoding technique of winner-takes-all voting. This decoding style consists of a group of neurons in a category that each represent an action that can be taken, and whichever neuron within that category spikes the most is chosen to represent the output of the network. In this application there are the two categories that can broadly be described as a direction and a speed. The speed category always has two bins. The direction category had between two and four bins, depending on the test, as described above.

3.2.2 Fitness Functions

We evaluated several different fitness functions for this application, the results of which are detailed in Section 3.3.2. Each of the fitness functions were normalized by the number of agents present and the number of timesteps in the simulation, to

help with simple comparison between experiments with these values. We evaluated both single-objective and multi-objective fitness functions using combinations of the following metrics.

Swarm Cohesion

The cohesion score for the swarm was used to measure how well they were traveling as a cohesive unit. The cohesion score for an individual agent was found via calculating the difference between a given ideal distance for agents to be from each other and the average distance from the agent to the closest N other agents. This function was discontinuous in that distances outside a certain distance from the ideal are ignored and treated as a zero instead. The cohesion score can be found with the following equation where *observed* is equal to the average of the closest N neighbors within the reward band.

$$reward_c = 1 - \left(\frac{|ideal - observed|}{ideal} \right)$$

Collisions

The collision penalty for an agent is a negative value applied if an agent tried to occupy the same space as another solid object. This is the main event that agents want to avoid. In our final set of experiments a collision was treated as an early termination condition, giving a harsh negative value in every category.

Goal Region

The goal region was an invisible and randomly placed area somewhere in the room. Each timestep agents in this region would be given a large reward. Additionally, if all agents were inside of the area at the same time, the episode would end and the agents would receive the full reward as though they had acted perfectly for every remaining timestep. The goal of this was to encourage exploring the room to try to discover

the goal. An example of the goal region can be seen in Fig. 3.2, where agent one is within the goal and the others are not.

Exploration Zones

The exploration zones calculation replaced the goal region calculation for some fitness functions. This consisted of a grid of congruent sections that completely covered the room. Agents in this model received a reward for entering a zone they hadn't personally entered previously. If all agents entered all zones, the same early termination would take place as described in the goal region behavior. Agents are always counted as being in the zone their center is in, so in the example of Fig. 3.3 agent one is in zone A, two is in zone D, and three is in zone N.

Forward Progression

The forward progression reward was used to encourage forward movement for the agent. It was found by tallying the number of drive forward actions taken by all agents across an episode. At the end of each timestep, if this was found to be ninety percent of the total actions taken or greater, the swarm would be rewarded proportionally to this tally. Otherwise, it would be penalized. Each agent also received a reward each time it chose to drive forward. The equation to find the progression score for a single timestep is given below, where $p_forward$ is the number of times any agent in the swarm chose to move forward divided by the number of choices made by all agents.

$$tally = \begin{cases} 10 * (p_forward - 1) & p_forward < 0.1 \\ 10 * p_forward & 0.1 \leq p_forward \end{cases}$$

$$reward_{fp} = count(forward) + tally$$

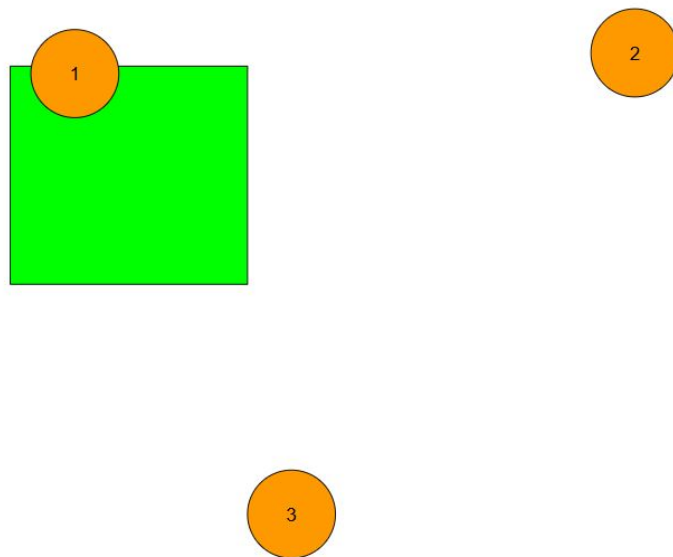


Figure 3.2: Example of Goal Region

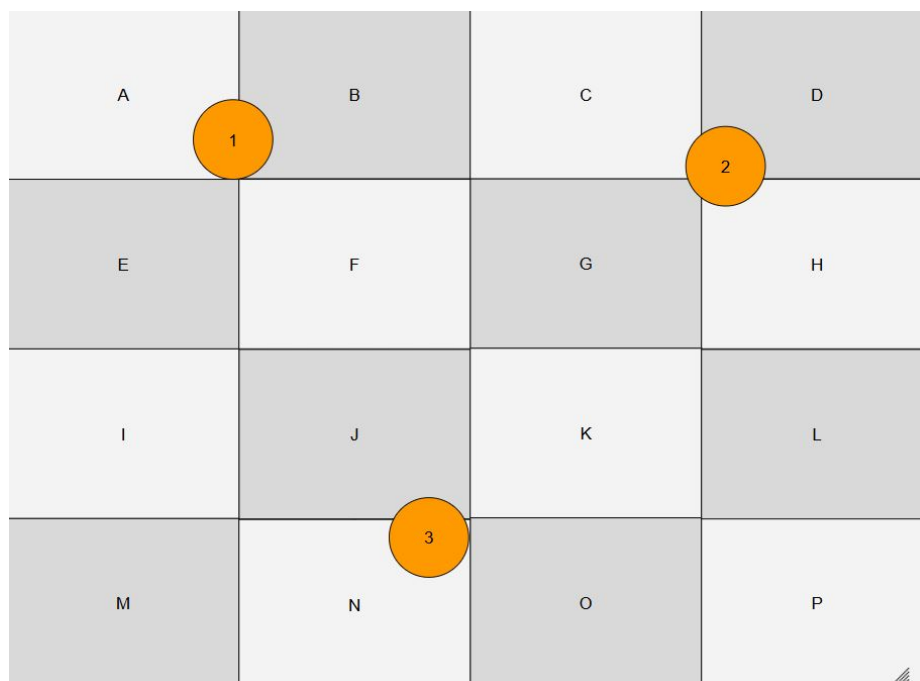


Figure 3.3: Example of Exploration Zones

Decision Streak

The decision streak reward was used to encourage the robots to make the same choice multiple times in a row while discouraging both undoing their previous action and completing rotations in place. This value can be obtained using the equations below where C_{cur} , C_{streak} , and C_{streak}^{-1} correspond to the current decision, the last decision made, and the inverse of the last decision made, respectively.

$$streak = \begin{cases} streak + 1 * speed & C_{cur} == C_{streak} \\ -15 & C_{cur} == C_{streak}^{-1} \\ 1 & C_{streak} \neq C_{cur} \neq C_{streak}^{-1} \end{cases}$$

$$reward_s = \begin{cases} -1 * streak & C_{cur} == turn \& \& streak \geq 24 \\ streak & C_{cur} \neq turn \& \& streak < 24 \end{cases}$$

The constants exist as they do to attempt to teach the agents that completing a full rotation is worse than doubling back on themselves, but if they double back on themselves and then continue turning in that direction, it will incur more penalty than having completed the rotation.

3.2.3 Network Evaluation

Visualization

A large amount of time was spent determining the best fitness function to use for training agents. This meant that the fitness scores achieved in different training sessions meant very little in relation to each other and could often have wildly different magnitudes. Since the goal of the project was more qualitative than quantitative, there was also no value that could be logged and tell us the best network across a series of sessions.

The solution that we came to to evaluate networks, and by proxy their fitness functions, was to run a series of independent training sessions, and save the best network of each. Each of these best networks would be tested in a standard environment of twenty-five by twenty-five units and the movements of the agent were saved in a log file. All of the logs associated with a particular fitness function would then be visualized together so that we could manually review the types of behavior that commonly occurred while training under that function.

Multi-Objective Selection

Training sessions in the default, single-objective, version of EONS produce a “final” network that is determined to be the best agent from the last training epoch. The multi-objective version of EONS, however, does not. Instead it produces a checkpoint that includes all networks in a population and the fitness score array associated with each of them.

To retrieve a single network from this log I applied a series of masks to the array, down-sampling it to only the networks that performed better than a certain standard in each category. From this smaller array I picked the network with the maximum score in the cohesion category, with the goal of encouraging agents to first ensure they are exploring and being productive, and secondarily stick together as well as possible.

3.2.4 Hardware

Observations

To physically mimic the observations we use in Section 3.2.1, we multi-threaded the LiDAR that the robot uses for its vision on the encoding unit to be continuously fetching new values into a list. When the network is ready for inputs for a new timestep, the encoder checks this list for the closest reading within it’s section and relays those as the inputs. In order to remove values that are no longer accurate, all values are removed every tenth of a second, or one complete rotation.

Many sensors were tested or considered for friend detection and were prototyped without the printed robot.

1. Infrared (IR) LEDs and sensors
2. Ultra-Wideband(UWB) communications
3. Hue sensor with unique chassis color
4. LiDAR shape detection
5. Microphone array and speakers

For the IR sensors and microphone arrays, the robots would require unique identification numbers that could be flashed in order to ignore their own LEDs or vocalizations respectively. The proposed solution for this was to include a jumper array on the power distribution board of the robots and jump the connections corresponding to a one in the binary representation of the identification number while leaving the zeros disconnected. This was early in the development cycle when changes could be made to the ongoing project of robot design. We used the prototype in Fig. 3.4 for testing this functionality.

Actions

As discussed in Section 4.3, the robot never had functional odometry, or positional tracking. Without this we were not going to be able to replicate the output of the simulation one to one. Instead, we opted for the most simple implementation of the output. At the end of each timestep, the robot would decode the network’s decision, stop its previous command, and begin its new command. This process repeated for as long as the robot was operational, meaning that it moved continuously, and each decision lasted as long as a timestep on the kit, which is about fourteen milliseconds. In order to minimize the error and jostling that could occur when the robot drove



Figure 3.4: Physical Testbed for IR Friend Detection

quickly, low speed was set to the lowest power that could be provided to the robot and still provide enough torque to drive.

This is also the appropriate place to bring up the largest difference between the simulation and the real-world implementation. The robot cannot travel as fast in a single timestep in the real world as it can in the simulation. If we decreased the simulated speed of the agent to mirror the capability of the robot, lucky agents could drive forward without worry of colliding with the bounds of the room. We also could not increase the duration of episodes due to training time constraints, and we worried that decreasing the size of the room to the size needed would prevent agents from forming interesting behavior. If this was discovered to be a problem, we planned to implement a timeout on the decoding unit of the neuromorphic kit to lengthen the real world timesteps.

3.3 Results

3.3.1 Application Details

Environment

The treadmill and obstacles were ultimately not used for the application. We found that they created agents that were much more timid and simply hoped for an advantageous seed. Additionally, there were persistent bugs that an agent could move through objects or walls if they aligned their movement to the direction of the treadmill. This likely could have been remedied given enough time, but because we were not seeing the benefits of these additional objects, we decided not to devote the additional time to following up on this. One interesting behavior we observed in a small number of early obstacle tests was an amoeba-like ability to split around an obstacle and reform. This was very interesting but only worked in the version of the application using holonomic drive and consistent angles, which was problematic for reasons listed below.

Randomizing the agent angles was not originally part of the simulation, but it became a huge boon to our progress. In early iterations, when utilizing holonomic drive, we seemed to be seeing great success with agents traveling in a flock as a single unit. On closer inspection, however, we noticed that at the beginning of a training session a preference for a random direction would be slightly more prevalent and lucky. As the session continued, we would see more and more networks that just chose that direction repeatedly with little to no regard for their inputs. This is because, with all agents being copies facing the same way, if they all agreed to only travel left, for example, they would appear to move together and be awarded points for cohesion.

Observations

Our LiDAR readings worked best with a 360° field of view and only one reading per section. We were surprised by the range preference exhibited. A complete field of view reduces the resolution of that view by a factor of four compared to looking at only one quadrant, from 22.5° per reading to 90° . It was very clear however that the agents trained with this larger field of view performed much better than those with limited peripheral vision.

The preference of one reading per section can likely be chalked up to the inconsistency of our hardware. In the software, the adjacent readings given with the smallest distance were always a consistent angular distance from the relevant reading. In hardware, however, the LiDAR was not rotating a consistent amount between readings. Especially if there were errors, causing some readings to be discarded, there could be massive differences in the distance between readings that were logged as “adjacent”. This lead to this version of the code confusing the robots more than it helped them, which was not worth investigating because the benefits of this input structure were relatively small in the runs where they were observed at all.

We were not able to make friend detection work. The only solutions that didn’t require active call and response were IR and hue sensors, but we were unable to make

these sensors function consistently. Since we could not find a way to make it work on hardware we left it out of the simulation as well.

Actions

Holonomic driving was ultimately a dead end for our purposes. I don't have a good intuition as to why, but we had much more trouble trying to make those versions of the agents behave intelligently. In comparison, the traditional driving scheme of moving forward and pivoting was much more promising early in our trials, and is thus the scheme that we tested our fitness functions using.

As mentioned in Section 3.2.1, we observed many agents choosing to move back and forth in a manner that contradicted themselves. This lead us to removing the ability to drive backward for agents, and then removing the ability to turn right when we saw that that worked to an extent but did not fully remove the problem. Right was chosen arbitrarily, as the goal was simply to force agents to not undo their own actions repeatedly. This did not fully solve the issue of agents choosing inaction, and their new solution was to spin in place. While still not ideal, it was easier to penalize this behavior than needing to account for every way of undoing themselves. Spinning in place was relatively prevalent, as is presented in Section ??, but these restrictions greatly improved the variety of behavior.

An interesting pattern that we noticed across both output schemes and all fitness functions was that agents almost exclusively chose to move at low speed. When moving at high speeds the likelihood of a collision skyrockets. This is even more true in cases of agent-to-agent collisions, where now both bodies may be moving at twice the speed they would be otherwise. It appears to us that the agents saw this as a risk that was not worth taking, nearly universally.

3.3.2 Fitness Functions

The figures in this section are visualizations of the results of the simulations. A single plot in the figure represents three agents navigating an episode, their paths being marked by triangles of increasing opacity that point toward the front of the agent. The four plots within a figure are the end result of four different training sessions, in order to demonstrate different activity patterns that the agents could learn given the circumstances. Circles or other points that are darker than those around them represent timesteps that an agent stayed at the same coordinates and chose to rotate.

Naive Single Objective

The first fitness function that we applied to the simulation, where only the coherency and collisions of the agents were tracked, showed little growth across many training scenarios. The agents seemed largely incapable of determining the best course of action, and most networks under this fitness function would either only drive forward, quickly hitting a wall or another agent, turn in place and hope to be close to other agents for a reward, or alternate choosing actions that were exact opposites, resulting in no net movement. This is what prompted the host of exploration and movement rewards that we attempted in the following trials, and what motivated the previously mentioned movement restrictions. Once the restrictions were in place only the first two options were observed.

Random Goal

The next variation of the experiment was largely the same, but featured a goal region. This proved initially successful, as we saw more networks that would drive and risk collision rather than spinning in place, but these networks were just driving forward without regard for their surroundings. The random nature of the goal region seemed to allow lucky networks to prevail over good ones, and in most cases we would once

again end up with unmoving agents. Seeing this, we chose to remove the goal region from reward calculations, despite its ability to motivate movement.

Movement Counting

The next attempt to motivate movement was notably more direct, and worked much better. We implemented the forward progression reward, and it was moderately successful and saw agents that moved most of the time and covered a notable portion of the room. Unfortunately, they primarily found a small track of space in the room and went back and forth on it without much effort to explore the rest of the area. This fitness function is the first time that we see behavior we will refer to as “orbiting”, wherein the agents drive circles around each other while staying in the same general area. See Fig. 3.5 for examples of this behavior. This is a good example of early versions of the type of cohesion we would like to see, but we would like to see them move through the room rather than spiraling in place.

Naive Multi-Objective

We then moved to the multi-objective functions that were described in Section 2.2.2. This change was made because we found that the problem of “explore an area together” was vague and could be better refined by breaking it up into separate criteria for the agents to experiment with.

The naive multi-objective function, however, was largely a failure. By completely separating the collision, coherency, and movement fitness scores, the network would learn within the first epoch to completely maximize either movement or collision and choose only drive forward or only turn. We found that these ravines were impossible for agents to escape from, as the hyperparameters we had set penalized agents for changing in a way that would allow them to find a better middle ground. The hyperparameters of this function had issues because the penalty for excessive spinning was static, not proportional to the amount of spinning. Agents would not see a notable

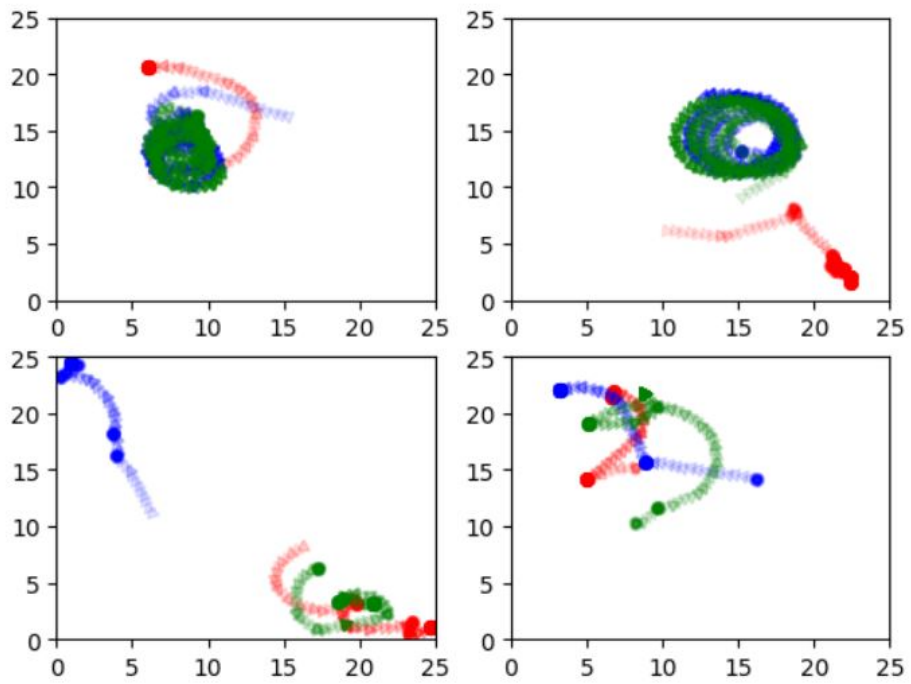


Figure 3.5: Networks Trained with Movement Counting

uptick in their movement category score until they chose to move forward more than ten percent of the time, and while they tried to reach that point the collisions category would become much less stable, because the difference in always spinning and spinning most of the time changes the probability of a collision from zero to a positive value that's functionally infinitely more likely. As can be seen in Fig. 3.6, there were some runs that chose to move a small amount, but they explored so little of the space that it was hardly an improvement on only spinning, and networks never got past the point seen in the bottom right of the figure.

Complex Multi-Objective

Our complex multi-objective function worked much better. By rewarding the coherency category only when moving forward and rewarding the collision category when moving, at a weight of one third the absolute value of the collision penalty, these categories became much less simple for agents to perfect and worked in tandem with the movement category. The movement category was also hard to perfect, as the “obvious” choice of going forward even when facing a wall would not allow an agent to collect the bonus reward of entering new exploration zones. This function faced similar issues to the naive multi-objective function, but to a much lesser extent. Early agents would spin over ninety percent of the time, but around the halfway point of training would reliably jump to driving forward approximately a third of the time. This is a marked improvement but clearly still not ideal. This fitness function sees the return of the orbiting behavior observed with the movement counting function. While basic versions of this can be seen in all four graphs of Fig. 3.7, the break in this behavior in the top left is interesting. Here, agents began by driving the loops we would expect, but upon spacing themselves out across the room they began to spin in place. Overall, this fitness function produced promising behavior, but the agents still tended to find a pattern to move in that didn't explore the complete area, making these runs an improved but similar variation of the behavior observed in the final single objective function.

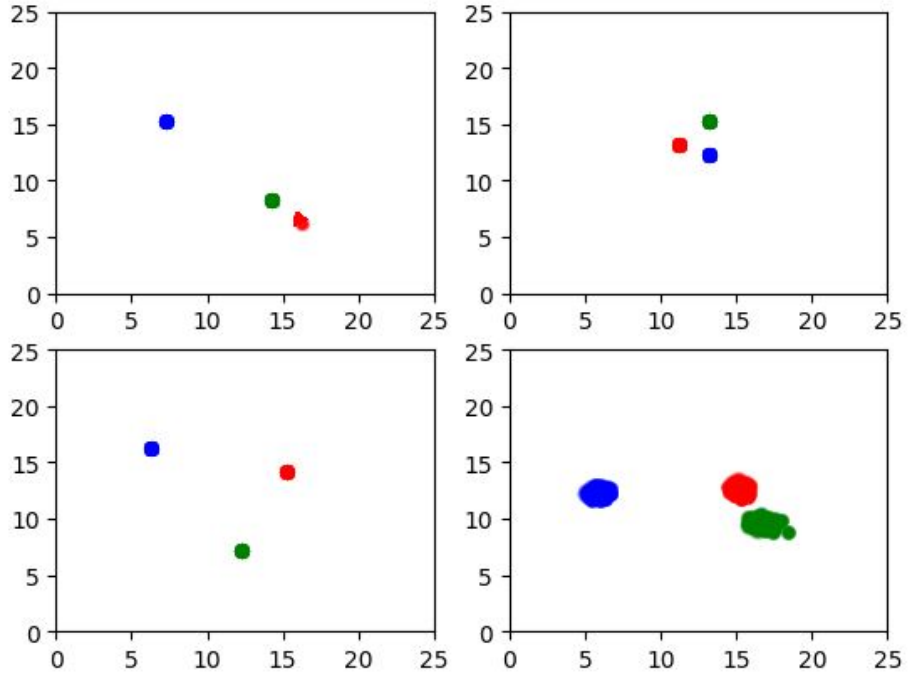


Figure 3.6: Networks Trained with Naive Multi-Objective

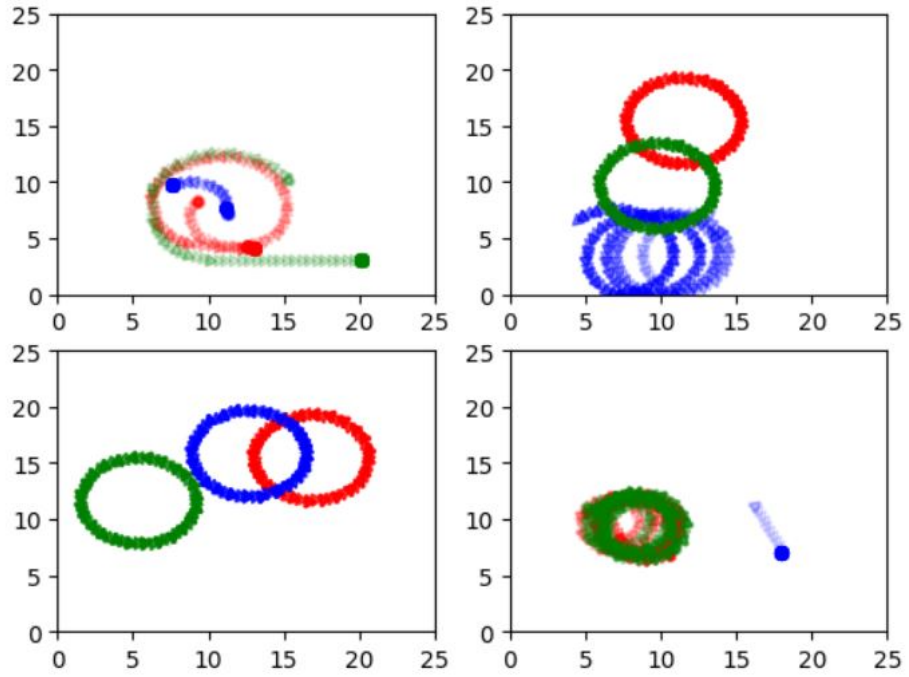


Figure 3.7: Networks Trained with Complex Multi-Objective

Exploration Separation

We next evaluated a fitness function that broke the exploration zones into their own category. This was aimed at fixing the problem of incomplete exploration, but it resulted in unexpected behaviors. By making this change, we made it too easy once again for the agents to maximize one of the other categories, in this case the movement category. This led to the same behavior that we had previously seen in the naive multi-objective function, but as we can see in Fig. 3.8, there were stable patterns seen here that aren't found in the naive multi-objective, namely the larger loops and orbiting found in the complex multi-objective and movement counting fitness functions. This function also debuted the decision streak reward. Adding this component allowed us to give the agents the ability to turn in both directions again without becoming stagnant. The behavior observed did not change much with this introduction but allowing the agents more choice brought this simulation more in line with our goals.

Coverage Percentage

Our most promising fitness function prioritizes the percent of the room covered by the swarm as a whole, as calculated with the exploration zones of Section ???. To work with this increased importance, we used one hundred zones laid out in a ten by ten grid. This approach also removes cohesion from the fitness scores entirely, encouraging a slightly different task of exploring the room cooperatively instead of directly together. We can see from Fig. 3.9, there is not a single run that landed in the pattern of only spinning in place or doing the safe loops that we had been seeing in our previous runs. In fact, the behavior here looks much closer to the best runs observed in Fig. 3.5, which makes sense because both of these fitness functions place high priority on encouraging agents to move. Despite individual agents seeming less intelligent than in the previous trials, much more of the area of these rooms are explored. This shows that this function better utilizes the core of emergent behavior.

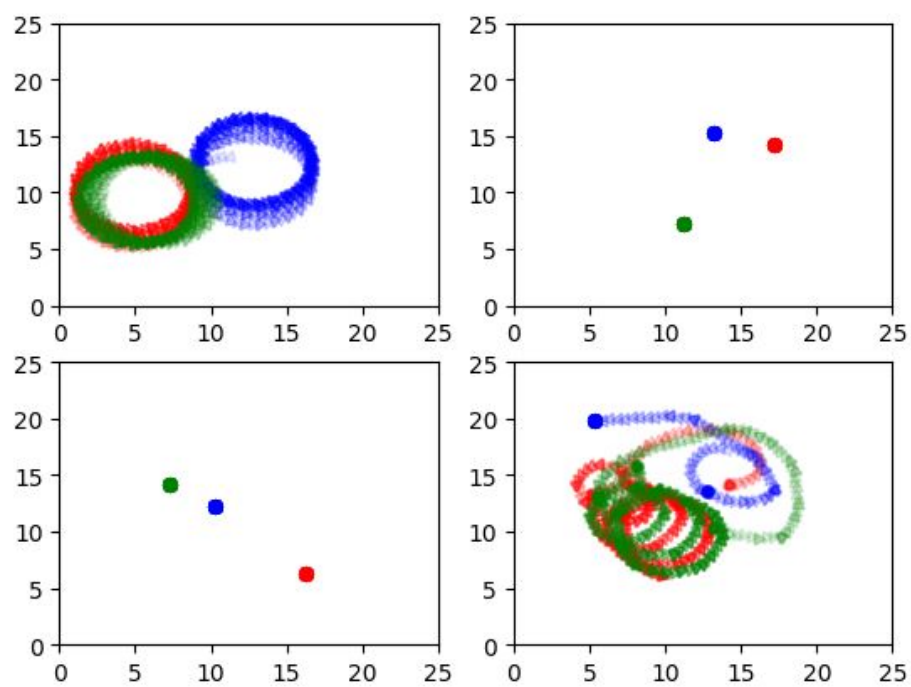


Figure 3.8: Networks Trained with Exploration Separation

These agents were also able to accomplish a new task that we had not previously evaluated, namely to randomize the room size for each training episode. All networks were visualized in a twenty-five unit square room, but the training runs could have rectangular rooms with dimensions between fifteen and fifty units. The robustness demonstrated in these networks was great to see, but we are not convinced that we have necessarily found the best option for fitness.

3.3.3 Network Evaluation

The sampling masks used to determine a best network from the checkpoints produced by multi-objective EONS worked well. They were capable of saving networks emblematic of the training session that created them. The main issue is that they created an entirely new category of hyperparameters that had to be tracked, in order to keep the minimum scores of the masks at a place that made sense for the behavior we were trying to encourage.

For the final fitness function listed above, we treated the final selection as though it were a single-objective problem. Agents were trained, selected, and evolved in the normal way, utilizing all of the fitness values, but at the end of the run the only consideration was the coverage score. This does preclude any episodes where a collision occurred as long as at least one didn't have a collision, because all categories are penalized immensely when that happens. This worked well, and we may investigate using it for future training.

3.3.4 Hardware

Challenges Moving from Simulation

The methods that we mentioned in Section 3.2.4 did not perfectly mirror those in the simulation, which resulted in issues when converting between the two. The LiDAR readings not being taken synchronously complicates things drastically. Flushing the values like discussed in an earlier section can lead to either missing or imagining

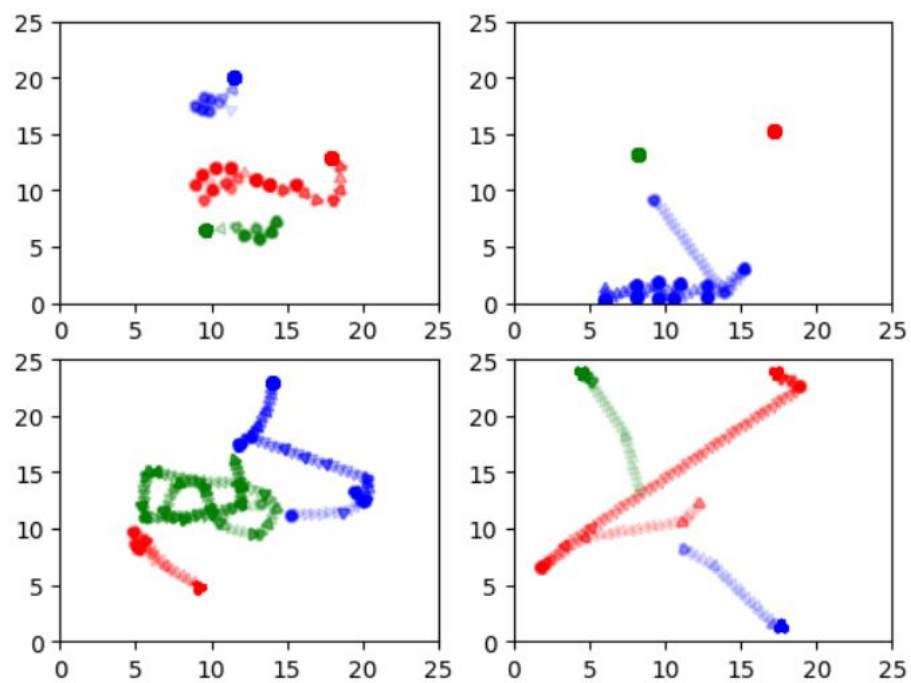


Figure 3.9: Networks Trained with Coverage

objects, depending on if the threshold to discard is too low or high respectively. The angle is also an issue. The current implementation of the robot does not have a gyroscope, and as such it assumes that the sensor readings are all in line with the value that they were saved at. This has resulted in behavior we’ve dubbed the “death spin”, where most physical tests of the robot end with the agent driving up to a wall, beginning to turn, and now thinking that there is that wall in two of it’s quadrants. This continues until all inputs show a very close object and the robot spins indefinitely, believing itself to be in a closed box, when it is really just next to a flat surface.

Successful Transition from Simulation

Aside from the errors mentioned in the previous section, our translation to physical hardware went extremely well. We had fears early in the process that the instability of the printed frame and the vastly different timestep sizes would make simulated networks extremely difficult to use on our robots, but the translation largely worked out. The lack of global information makes the agents very resilient and adaptable to the drift incurred by their shuttering frames. With networks that are hand selected using the visualization tool seen throughout this chapter, we are able to observe very similar behavior between a simulated agent and its physical counterpart. You can see the robots in a physical testing space in Fig. 3.10. The physical robots also completely avoided collisions under normal conditions. In our testing, the only wall an agent actively ran into was a reflective doorstop, which we theorize gave the LiDAR an incorrect reading.



Figure 3.10: Robots in Testing Space

Chapter 4

Swarming And Learning Machine Operating on Neuroprocessors

4.1 Setup

4.1.1 Goals

The Swarming And Learning Machine Operating on Neuroprocessors (SALMON) robots were conceived with the swarming application of Chapter 3 in mind. We quickly realized while designing them that the constraints of this project also lent themselves to the chassis being very adaptable for a wide range of projects. We decided to lean into this and maximize the reusability of the robots, turning them into a project in their own right.

4.1.2 Constraints

Below are the primary guiding principles of the robot design, including the reason that they were part of the swarm constraints and how it helps a general use platform to operate.

Replicable

With the goal in mind of creating a swarm of robots, we needed the design to be easy to construct repeatedly. This would allow for easy construction of a swarm that would all act exactly the same when given the same code.

Easy replicability means that future users can create and get acquainted with the platform more easily. It also meant that while choosing components we looked for those that could easily be found in large quantities. Straying away from niche items decreases the likelihood that future user will need to deviate from our original design.

Inexpensive

We were aware that all costs would be multiplicative when making a group of identical robots. This lead us to look for inexpensive components whenever possible.

Like replicability, the lost cost of the robot will hopefully make it a great testbed for many projects, even if they later move to more specialized hardware. The goal of this section was to keep the barrier of use low so that people will be interested in trying it.

Adaptable

Intending the minimize the latency between a trained network from the swarm application and its deployment to a robot, we developed the first iterations of the robot while still designing the simulation. At this point, it was unclear what the app would look like. This led to a design where parts could be used in multiple ways or be easily replaced without recreating the entire robot.

This is clearly useful for the general case. Being able to quickly change components and layouts would be a great boon to other projects looking for a hardware platform, because the odds of another project needing exactly the same input and output space are relatively low compared to projects that just need a driving robot.

4.2 Methods

4.2.1 Locomotion

The drive scheme that we decided on was an X-drive configuration. This is a holonomic drive, meaning that the controllable degrees of freedom are equal to the total degrees of freedom. This means that, with independent control of each wheel, the robot can move in any direction in the two-dimensional plane, rotate in place, and even rotate while driving in a straight line. We knew that holonomic drive would be the best option for the reusability of the robot, because it allows for any drive scheme to be mapped onto it. We also considered Kiwi drive, a type of holonomic drive including only three wheels as opposed to the four of X-drive, but we decided on the latter because of my familiarity with it from previous robotics experience.

Motors

The motors we originally used were TT motors because they were easy to come by. They also have a turning axle that could be accessed on either side of the motor body, so we could put the wheel on one side and the encoder could read off of the other. Wires had to be soldered to the pads of the motor and connected to the motor driver.

During development there was a complete overhaul of the drivetrain. The reasons for the change can be found in Section 4.3. During this process we switched from the TT motors we had been using previously to Adafruit DC motor in micro servo body. This was a completely different motor in a very different form factor, and so a change in the printed components of the wheel assembly was needed as well. Additionally, since these motors were much smaller than the previously used TT motors, the chassis had to be redesigned to move the mounting holes. This was required to keep non-adjacent wheels in line and keep the intersection of those two lines in the center of the robot, as is required for a functional X-drive.

The components of both versions of the motor assembly can be seen in Fig. 4.1, with the original on top and the second iteration on the bottom.

Wheels

To use X-drive, the robot must have omni-directional wheels. These function similarly to normal wheels, but they are covered in rollers that rotate perpendicular to the wheel's axis of rotation. This allows for the wheels to be moved along the axis of rotation without imparting any drag or other force onto the body.

Our original wheels are 3D printed based on a file found on the website thingiverse, with slight modifications made in order to work with our motors, which were TT motors at the time. The rollers were printed using a traditional extrusion printer, but we found this to not be precise enough for the hub of the wheel. For the hubs, I used my personal resin printer.

When we switched our motors, we decided to switch to purchasable wheels as well. The two inch Dualie Omni Wheels from Andymark were approximately the same size as the printed wheels we had been using, and I had prior experience with them, so I chose to use these wheels going forward. The purchased wheels had a hard plastic hub and 35A durometer rollers.

Motor Controllers

We used L928 Dual H Bridge motor drivers to control the motors. The dual H bridge design lets us control the speed of the wheels going forward and backward. Two of these drivers were used so that all four of the motors could be controlled independently. Control of a motor involves two pins that give direction and one pin that gives speed.



Figure 4.1: Components of Motor Assemblies

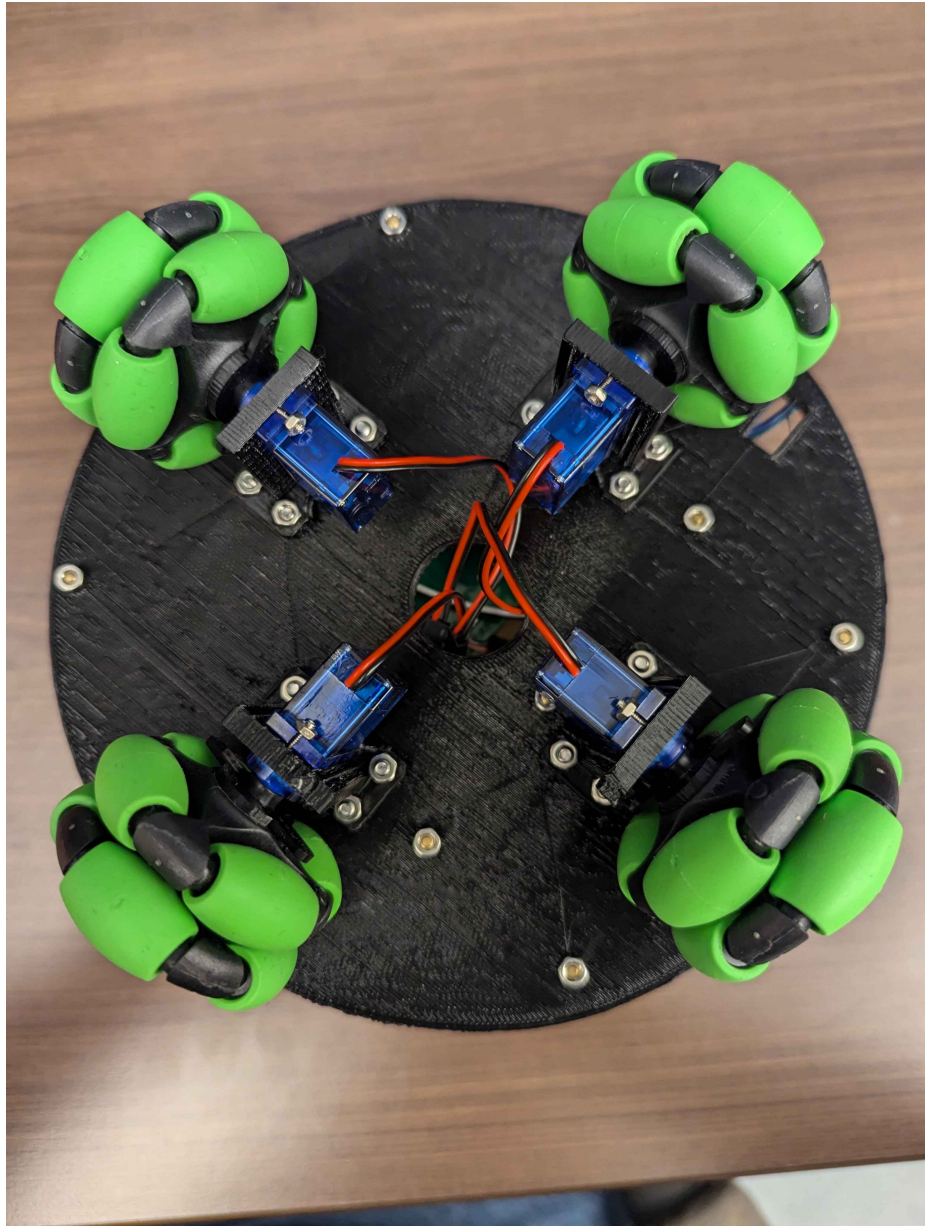


Figure 4.2: View of Robot Drive Train

Printed Components

Both versions of the wheel assembly included a few printed components to connect the pieces together.

The first is the motor bracket, which was completely redesigned but one was used in each assembly. This piece connects the motor to the chassis via the mounting holes that were already present on both motor types. On the TT motor this also had a slot that held the motor encoder in place.

The TT motor assembly included an adapter from the axle of the motor to the motor encoder. This piece was able to be printed on an extrusion printer, but it was certainly pushing the limits of how small we could get. Obviously due to the removal of the motor encoder in later revisions, this piece was also discontinued.

The micro servo body assembly included an adapter from the horns included with the motor to the wheels. This was necessary as the motor as a very small plastic gear as its output shaft, the teeth of which are too small to accurately print a meshing counterpart to. The horns, by virtue of coming with the motor, already mesh with it. This adapter went through many iterations, and the main two versions were to either have either two sub assemblies of horn-motor and wheel-adapter that connect and are screwed together or a subassembly of horn-adapter that attaches to the wheel on one side and the motor on the other. Listing the adapter as a component of only one assembly is misleading, however. While there was no dedicated part for this job on the TT motor assembly, that's only because we were able to add it to the hub of the wheel. The modifications still had to be made, they were just able to be joined with the wheel instead of created standalone.

4.2.2 Chassis

The chassis is the frame of the robot on which the rest of the components are mounted. It consists of two levels, the body and the sensor hat, both of which are 3D printed.

The levels of the chassis are stacked on each other via pegs and are not adhered in any way, meaning the top level can be easily removed to reach the interior components.

Hood

The hood is an optional additional piece of the chassis. It is mounted on top of the sensor hat via pegs, similarly to how the sensor hat rests on the body. The hood was created to give the robot a solid body and was conceived of when we were not sure how accurately the LiDAR would stay on a two dimensional plane. The hood is largely an aesthetic piece that can also protect internal components, but it is ultimately not a necessity for the robot. As such, it will be primarily ignored. There are variations for each version of the sensor hat both with and without the pegs to mount the hood, so that projects not desiring to use it can ignore it wholesale, without dedicating filament and print time to the mounting pegs.

Body

The body is the main level of the robot. It consists of a flat disc and three pillars that hold the sensor hat above it. The disc has mounting holes for the motor brackets, motor controllers, and PCB stack. This piece went through several revisions as those components changed and moved, but those changes will be covered in their respective sections. The disc also has a rectangular hole that is intended to be used to access the battery's power switch.

Sensor Hat

The sensor hat takes inspiration from the Raspberry Pi “Hardware Attached on Top”, or HAT, system. The name was clearly a product of this, and the functionality is also similar. This layer of the robot is designed to host the sensors and other aspects of the robot that we predict are likely to change from project to project. For the development of the robot we designed two versions of the hat. Both include mounting

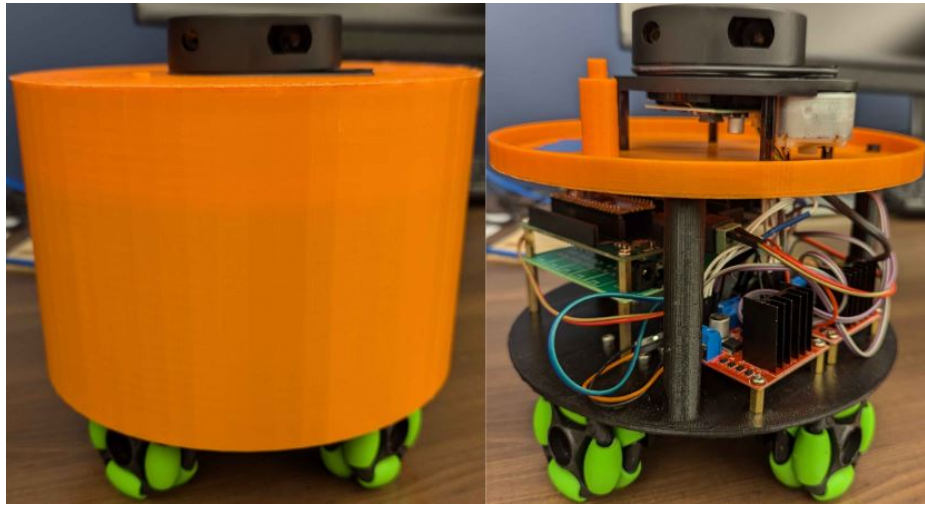


Figure 4.3: Alternate Angles of Robot Chassis

holes for the LiDAR used in the swarming robot project, and one additionally has holes for an inertial measurement unit, or IMU, to be mounted beneath the LiDAR.

The sensor hat has holes for pegs from the body, rather than vice versa, because the intended use case of the robot is to be able to use the exact same chassis for most projects, and only need to reprint the hat when changing the functionality of the robot. While putting the columns on the hat, rather than the body, would have made it easier to access interior components of the robot, it also would have meant that this piece that is intended to be reprinted over and over with slight modification would be using a much higher amount of filament.

4.2.3 Sensor Suite

The sensor suite, like the sensor hat, is intended to be changed from project to project. This section will focus on the sensors that we have included in our testing so far, which we anticipate will also be some of the most commonly used sensors in other projects.

LiDAR

The LiDAR, which stands for Light Detection And Ranging, sensor is the primary input device for the swarming application. This means that we worked with it a lot compared to other sensing options. However, we don't think that this is unrealistic for the theoretical use cases of the lab. Most of the robot projects that are in the lab already have a LiDAR of some kind in their sensor array.

The LiDAR used is an RPLiDAR A1, because it was the least expensive option we found for a 360° two dimensional LiDAR. This sensor can take eight thousand readings per second and complete ten rotations per second, giving approximately eight hundred readings per rotation. We condensed the seven pin plug to four wires by combining both ground pins into one, and splicing together two 5v pins with the PWM wire. This ties the PWM, or Pulse Width Modulation, signal to always be on full duty cycle, setting the LiDAR to full speed. The issue we anticipated having was

getting a wide enough field of view each timestep, not the fidelity of that view, so there is not a time we planned to ever not have the motor at full speed.

Inertial Measurement Unit

While not originally included in the sensor suite for the swarming application, we added an Inertial Measurement Unit (IMU) to the sensor hat, anticipating that many projects would be likely to use one. The IMU that we used is a GY-521 breakout board that uses an MPU-6050 sensor. The IMU is mounted below the LiDAR, centered in the disc of the sensor hat, in order to minimize error accumulation.

Motor Encoders

Early versions of the robot included motor encoders that could track the movement of each of the robot's wheels in order to drive for specific distances and adjust for error in the system. When switching from the TT motors to the mircoservo motors, see Section 4.2.1, we were unable to find a suitable mounting solution and thus removed the encoders altogether. When there were encoders present, we used Alps Alpine EC10E1220501 mouse wheel encoders for our motor encoders.

4.2.4 Power Distribution Board

The core of the robot is the Printed Circuit Board (PCB) stack. The stack includes the neuromorphic kit from Section 2.3 and a Power Distribution Board (PDB) that I designed specifically for this project, Fig. 4.4. The purpose of the PDB was to make powering devices easier. The kit essentially has the available ports of the encoder and decoder microcontrollers, which is enough for small projects such as the examples, but in order to control a robot I would need to split all of the voltage and ground ports many times over. The PDB exists to solve that problem and keep the robot cleaner in terms of wiring messes.

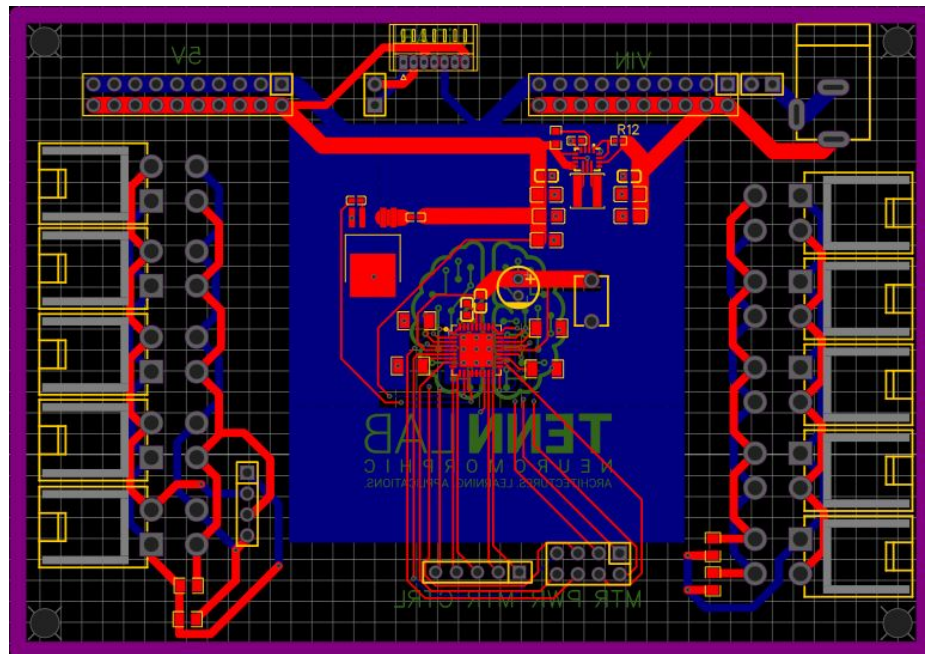


Figure 4.4: Circuit Diagram of Power Distribution Board

Board Details

A core feature of the PDB is that it has the exact same footprint as the kit, including the placement of the mounting holes. This means that the PDB doesn't impose any extra constraints on the design of the robot or placement of components than the kit already would have. This tenant lead to a board with a much larger footprint than it needed, and as such a lot of space that could be used for extra features.

Kit compatibility also lead me to use the same battery and power port that the kit already used, placed so that it would be directly under the power jack of the kit. I knew that the barrel jack would stick out from the board a notable amount, by stacking both instances of it, this limited how much of the stack's perimeter breached the rectangle of the boards. This could allow future projects to take the stack and use it with entirely different chassis structures if they had a project that required it, but they would not need to redesign the electrical components.

Voltage Regulation

The voltage regulation circuit seems very redundant at first glance. It was largely inspired by the same circuit on the kit, which sits a mere inch above it and performs the same task. This brings to light a challenge of this robot's design; I did not design the kit and I cannot alter it. The kit does not grant user's access to the unregulated battery voltage, about 12v. Instead it has ports for 5v and 3.3v. These are the voltages given by the Raspberry Pi Pico microcontrollers and the voltage levels they expect to use for logic. By not including the battery voltage the kit likely will save many users from accidentally burning out their peripherals. However, the motor drivers used in this project require 12v, so this board needed to provide access to that, meaning that the battery must be plugged into the PDB first, which can then power the kit off of said battery line.

It would be possible to then connect a wire from the 5v line on the kit back to the PDB, but ultimately I decided that that would complicate the assembly more than

having a second regulator. This gives the additional benefit of allowing the PDB to be a standalone board if a user has an alternate controller that they wish to use. As long as it can accept either five or twelve volts, it can be used with this board.

Power Rails

The power rails are what this board provides in place of splitting wires from the kit. There are two twelve by two pin rails that take up the rest of the edge of the board that the battery port is on. The headers take ninety degree pins so that the row closer to the edge of the board is on the bottom and the other row the top. The top row of pins on each is the positive voltage and the bottom row is ground. The rail closer to the battery provides unregulated battery voltage, which is used by the motor drivers and the kit, and the farther rail provides a regulated 5V, which is used by the LiDAR before the rail starts and an IMU on the rail. The rails are almost entirely unpopulated so that future projects are should not run out of power pins before they run out of control and communication pins from the kit.

I2C Buses

The short edges of the board have an inter-integrated circuit, or i2c, bus each. I2c is a communication protocol that lets users use four pins of a microcontroller to power and communicate with many peripheral devices. Many times these devices can be plugged directly into the microcontroller, but depending on the specific device, pull up resistors could be needed. The busses on the board have these resistors preinstalled, because if they aren't needed users can just use the ports on the kit to interface directly with the device. I included two buses on this board because the Pico can support both 3.3v i2c devices and 5v i2c devices, so if necessary one side could be used for either voltage. This aspect of the board is superfluous. It took very few extra components and utilized empty space on the board in a way that future projects can hopefully use, but there has not been any need for it thus far.

Extra Components

There are some additional components on the PDB that can be simply described without a dedicated subsection. These include

1. A port for a power switch.
2. A port for the LiDAR. Despite being a sensor, we expect the LiDAR to be used in a lot of projects. This, combined with the connector it uses not being compatible with the ports of the kit, led us to add a designated port to the PDB. This also includes a two-pin port for the communication pins to be routed back out to the kit.
3. An integrated motor driver. This would allow for the removal of the motor drivers in Section. [4.2.1](#), decreasing the component list and allowing for a smaller footprint if desired. Pins are present on the remaining long edge to plug in 4 motors and control them with 5 pins.

4.3 Results

The robots are as cheap as we can feasibly achieve. The breakdown of costs of major components is available in Table [4.1](#), and the examination of the printed components is in Table [4.2](#). This table does not include the price of wires and bolts because they are both cheap and likely available to any user building a hardware platform. The quantity of horns is not a flat number because they come in a pack of three, so for every three robots you must buy four. The PCBs include shipping because that's the largest part of their cost, and come in a minimum of five. The standout prices are the battery and LiDAR. The former was chosen for its compatibility with the kit and availability in the lab, and a version of this robot with a cheaper battery would be very easy to accomplish. The latter is due to a lack of options. This was the cheapest 360°LiDAR we could find, and given the prevalence of LiDARs in applications on the

framework and other robots of the lab, we deemed this a necessity. The total cost of this product is \$271.87 per robot, or \$168.87 without any sensors. We did not have a set goal for the product, but we are satisfied with the result.

4.3.1 Locomotion

I will begin with the results of the locomotion category because they were discovered very early in the process, and there are effectively two sets of results, for the two wheel assemblies we evaluated.

There is an unfortunate result to note before that split. We failed at creating a fully holonomic drive. With the motor drivers used on the robot, each motor needs three pins to fully control, two for direction and one for speed, for a total of twelve. Unfortunately, the decoder on the kit only has eleven available pins with which to output. We are using splitters to reduce the number of required pins to eight, but in doing so we also lose full control. The compromise that we came to is that the robot can either drive forward, backward, and spin, or it can drive forward, backward, left, or right, and you can switch between these modes by changing which pins are wired together. This will cover most use cases but is ultimately not what the goal of the project was.

TT Assembly

The TT motor assembly worked well as a prototype, but had several issues that quickly became obvious. The first is that the shape of the motor, where the larger mounting holes are not in line and only on one side of the axle, created a lever out of our bracket. This left a lot of movement in the motor and allowed for inconsistent pressure between the ground and the wheels.

The second issue with these motors was the solder pads. The motors didn't come with attached wires, and the solder pads that we had to attach the wires to were

Table 4.1: SALMON Component Prices

Type	Product	Price
Printed Components	0.293kg PLA	\$8.20
LiDAR	RPLiDAR A1	\$99
IMU	GY-521	\$4
Battery	LBP-124500	\$62
PDB	Custom	\$6
Kit	Custom	\$6
Motor Driver	Dual H-Bridge	\$2.50(x2)
Motors	Microservo Body	\$3.50(x4)
Horns	Adafruit horns	\$1.25(x4/3)
Wheels	Andymark Omni Wheels	\$16.50(x4)
Total Cost		\$271.87

Table 4.2: SALMON Print Weights

Component	Weight
Hood	0.115kg
Body	0.091kg
Sensor Hat	0.063kg
Motor Adapters	0.006kg
Motor Bracket	0.018kg

incredibly frail. There were multiple instances of the pad breaking off, even to the point where we had nothing left to solder to.

These issues together made us re-evaluate our motor solution, and since we were changing part of the assembly, we opted to stop printing wheels as well, because they had their own set of issues.

Our wheels had issues keeping traction on the ground. We used carpet squares to try to combat this, and while this performed better than the floor of the lab, it was still regularly not able to turn or drive correctly, likely due to a combination of inaccurate roller printing and inconsistent pressure. It didn't help that many of the wheels were slightly broken by the time they were put on the robot.

3D printed resin, like we used for the hubs of these wheels, is extremely brittle. This is a very bad trait to have when we have very thin areas of the wheel that are meant to be tensioned with a bolt. To assemble a wheel, there are eight rollers on bolts and a set screw, each of which is held by a thin piece of resin. Many of the wheels would break during assembly.

Speaking of assembly, due to the tight tolerances of the head of one bolt and the nut of the next, assembling one wheel would easily take almost an hour, and the nuts would still occasionally come loose while driving. While not included for most of our decisions, we looked at the realistic side of robot construction in this area. Doing this we saw that, accounting for the labor cost of a researcher assembling two wheels, since the odds were that one would break, it was as expensive to produce our own wheels as to buy them from a supplier, and the latter would have a much higher quality.

Microservo Assembly

The dc motor in the microservo case solved all of our problems with the TT motor. The motor was small and the two mounting holes were in line with and on either side of its output shaft. The motor came with wires already attached, and the cohesion point was inside of the protective case, ensuring that the pads wouldn't be jostled. It did create new problems, such as displacing the motor encoder mentioned in [Section 4.2.1](#)

and forcing us to design an adapter, but these issues were much less of a hassle than those of the previous generation.

The first horn to wheel adapter style we tried to create worked by creating a hole for the horn to come down from the same direction as the wheel, sticking slightly through the end of the adapter, and connecting to the motor. Then you could put the wheel on, locking the horn in place. This ran into some issues because the design required most sections of the adapter to be very slim. This regularly caused bending in the prints, breaks when trying to assemble, or allowed the wheel to spin freely because the adapter couldn't hold the wheel in place relative to itself.

The adapter that we settled on is used by creating two sub-assemblies of motor-horn and adapter-wheel, pressing them together because they are keyed to spin together, and screwing the section together to keep the adapter and horn from separating. This solution works very well and had minimal unwanted movement, despite having multiple points of failure from axle to wheel.

The change in wheels worked out even better than that of the motor. The wheels that we purchased not only worked much better on our carpet squares, but performed just as well without them, freeing us to test in many more locations. We still notice some drifting while the robot is supposed to drive in a straight line, about 5° per meter, which we blame primarily on possible warping of the printed chassis. However, given the constraints of the build, we are very happy with the accuracy at which it drives.

4.3.2 Chassis

The changes to other components did not majorly affect the efficacy of the chassis. It properly does its job of hosting all of the parts of the robot, however there were some aspects of the design that we thought were worth noting for the future. We found that centering the weight of the robot was very important, and because the battery is a sizable percent of that weight, it is better to place it beneath the PCB stack. This renders the hole for its power switch obsolete and vestigial in current usage.

We did not consider during the design of the chassis that one of the columns supporting the sensor hat is located squarely in front of the micro-USB port for the network processor microcontroller. This means that to deploy new code to that board you must remove it from the kit or remove the kit from the robot, the former usually being the easier of the two. This is not a large issue and doesn't impact the functionality of the robot, but as a tool meant to be given out to others I think that the quality of life changes like this are something to keep in mind.

4.3.3 Sensor Suite

LiDAR

The LiDAR works very well for what we need. It is able to consistently make a complete rotation in the time it takes the kit to run approximately seven timesteps.

There was extensive work that needed to be done to communicate with the LiDAR, as there is not a native C library for the sensor, and the C++ library that existed made heavy use of concepts like classes and default parameters that don't exist in C. However, once this was completed, the sensor was able to reliably communicate quickly enough to cause no problems with the kit's cycles. In order to maximize the data seen from the LiDAR, I multi-threaded the kit's encoding unit so that one core is constantly receiving new data from the LiDAR and storing it, and the other core is waiting for the decoder to tell it to start the next timestep and transmit the relevant information to the network processor. This allows for much quicker timesteps than waiting for the LiDAR to make a rotation before progressing.

Inertial Measurement Unit

The IMU worked out of the box as anticipated. However, because there is no magnetometer, there is drift in the yaw calculation, which is the relevant axis for the projects we have been testing. To calculate yaw, the heading of the robot, without a reference to "true north" like a magnetometer would give you requires that you

integrate the angular velocity of the robot with respect to time. Unfortunately, the Raspberry Pi Pico has an inaccurate onboard clock when looking at the microsecond scale. This causes the integration to be inaccurate and thus the heading to follow. The error accumulation is small when turning at speeds up to about ten RPM, but when the instantaneous velocity gets higher than that the error compounds and becomes offset for the remainder of the run.

Motor Encoders

As discussed in prior sections, the motor encoders are not present on the current iteration of the robot. We deemed that it was not relevant enough to justify putting work into figuring out how to use them with the second generation of motor assemblies. This is because they were not very helpful while they were present. The encoders used had such a small hole for the axle they were supposed to measure, that our printed adapters couldn't ensure that they actually caught all of the movement in the motors. We found that due to this, the robot could actually create more error by trying to correct itself with an encoder than it originally had while driving. If the print dislodged, then the motor it was attached to would spin until we manually stopped it, because the encoder didn't ever observe it finish the desired distance. This, coupled with the relatively small amount of error we actually observed in the motors, led to us ceasing use of the sensor while it was still part of the robot, long before we changed motors and saw an even smaller magnitude of error.

4.3.4 Power Distribution Board

The primary goal of the PDB was accomplished easily. The power rails are in use and cut down on the mess of wires that would otherwise have existed by a significant margin. The i2c buses are not in use but are functional in case they are called upon. However, some of the additional features that I attempted to add did not work out. The power switch pins, likewise, are functional but unused. The battery we used has

an integrated power switch and made this feature redundant. It is bridged open on the boards currently in use.

The LiDAR port is the more interesting failure. If the LiDAR is plugged into its port on the PDB when the board is turned on, everything on the board that is supposed to be regulated to 5V is instead held at a steady 2.1V. If, instead, the board is turned on and then the LiDAR is plugged in, everything works as intended. Additionally, if you power the LiDAR from the exposed Pico pins on the kit, as is our current solution, everything works as intended once again. I believe that the issue is that the startup current draw of the LiDAR we used stops the capacitors of the voltage regulator from charging correctly. This could likely be fixed by untying the PWM wire from permanently high, but doing so would need to involve changes to the PCB, it is not done in the wiring.

The integrated motor driver unfortunately does not work either. The chip that it was designed for went out of production between the design of the PDB and when I could have assembled it. Another company, at the same time, began releasing a very similar chip with the same SMD footprint and a very similar datasheet, and I believe that they bought the die used to make the chip. However, this chip has a slightly altered internal truth table, Table 4.3, which means that the way my board was designed locks each motor into coast mode whenever the chip has power. This is frustrating and, once again, can be solved, but would have to be done with a new PCB. If this had worked, it would have also eliminated the problem of not having enough output pins to fully control the motors for complete holonomic drive.

Table 4.3: Truth Table of Motor Drivers

Enable	Phasex	Old Action	New Action
L	L	Reverse	Coast
L	H	Forward	Coast
H	L	Coast	Reverse
H	H	Coast	Forward

Chapter 5

Two Board Neuromorphic Kit

5.1 Setup

5.1.1 Goals

The original goal of this project is clear from the title, to condense the neuromorphic kit from Section 2.3 from the current three microcontrollers to two. This was to be accomplished by putting the encoder and decoder on a single controller, called the CPU for this chapter, while leaving the neuroprocessor separate. The kit had always been envisioned as the three separate boards, though, so combining these cleanly would take more than putting the code from both programs together.

One of the biggest benefits of combining the encoder and decoder units was to grant more freedom to the pin declarations. On the kit as it exists, there are twelve I/O pins available for the encoder and eleven available to the decoder. The Raspberry Pi Pico has twenty-six general purpose input/output, or GPIO, pins total, and although some of those must be used for inter-processor communication, allowing all free pins to be used for inputs or outputs grants much more freedom to the user. For instance, many sensors can be bussed, meaning only two inputs are used. This leaves twenty-four pins available for communications and output. Alternatively, a

network that only needs to output to the onboard LED now has access to twenty-five pins for a sensor array.

While working with the kit, I found that the workflow was unintuitive. To upload all new code to the kit, you need to go two to three sub-folders deep to reach each build location, and the same applies to the code folders. Although they can all be built using the same commands, the locating each file that a user needs to change is a task in and of itself. Because the code structure had to be overhauled, I thought that addressing this could also be helpful to future users.

5.1.2 Constraints

The constraints of this project are all centered around the idea of keeping this kit as close to the original as possible. I want it to be an easy process for users to transition from the original kit to my kit. The user-facing functions that are common between both versions must have the same functionality. The observation loop will continue to be called once per input per timestep and the action loop will be called once per timestep. This also applies to the command line arguments, aside from those mentioned in [Section 5.2.3](#).

5.2 Methods

5.2.1 Code Combination

The main part of the project is combining the code of the encoder and decoder into a single CPU. To start this I simply copied the preexisting code for the two units into a single program file. This worked, but left a lot of redundancy in the code for items that were in both programs but only needed to be included once. By combining both while true loops into one section a lot of the code duplication was avoided. Doing this, I also found internal redundancies.

On the decoding unit the code checks for SPI use and reads from the bus if required, then goes to decoding and within that section checks for SPI use again and reads voltage spikes if SPI is not being used. On the encoder something similar is done, if SPI is used everything is written to SPI, then the code enters another statement, and within that section there is another SPI check. Overall, the encoder loop checked for SPI usage three times and the decoder loop checked twice.

To eliminate these repetitive checks I used a macro for SPI use, like in the original, but I utilized preprocessor directives to overload all of the functions needed by the CPU loop. By doing this, the code would either build for SPI use or for voltage spike use, and no checks should need to be performed during the loop.

Overloading the code in this manner required abstracting much of the program into a header file, and I decided to double down on this with the goal of improving user experience. A user of the kit does not need to open the core code file, expounded on in Section 5.2.3, but if they wanted to explore the underlying logic of the kit, abstraction makes it easier for new people to understand what the code does at a high level, and they can look at the header to dive deeper into any individual functions.

5.2.2 Communications

The communication on the kit, when I began this project, had the options of either digital voltage spikes or SPI bus. I wanted to keep both options available for users because I think it is important to maintain the ability to use spikes when possible for a spiking network, but some networks will have too many input and output nodes for this to feasibly work. This had the added benefit of keeping this kit as in-line with the original kit as possible.

SPI

The SPI bus is very simple to physically connect, as will be detailed in Section 5.2.4. On the software side, I based the packets off of the existing ability for the kit, meaning

two sixteen-bit values per packet. The first value categorized the signal and the second elaborated with the intended value.

By incorporating a command signal that means to stop reading, and leaving the main loop of each board to read for packets until they find it, we also eliminated the need for separate synchronization pins. This would not have been possible for the three board kit, since the encoder needs a signal from the decoder to begin, and they do not have any other communication. This means that SPI communication removes only four pins from the total, leaving twenty-two for the I/O.

Voltage Spikes

Voltage spikes are, on the surface, even easier. Each pin turns on if it represents a spike and off if it doesn't, a synchronization pin tells the other board to record the current state as one piece of the interval, and then repeat. It is essentially that simple, but to increase the connectivity of this edition of the kit I had an idea to reduce the width of pin count required.

The system I used was to find the pins not being used by the user, discussed in the next paragraph, and treat them as buckets. From there, the input neurons and output neurons would be distributed across the pins in order to minimize the number of neurons that each pin represented. Now, instead of looking for *Interval* signals, the second board would look for $Interval * Uses$, where *Uses* is the maximum number of neurons that a pin represents. Pins need to be assigned a purpose of either in or out for a board, meaning that no pin can represent both input or output neurons, so there needs to be at least one for each direction.

To assign pins, we allow for users to declare what pins they want to save for either their input or output and use the rest for communications. Timesteps can run faster when *Uses* is smaller, and thus we make use of as many pins as possible for communication via voltage spikes. In the most extreme case, when only the minimum of two pins are used for communication and two for synchronization, this method uses the same number of pins as SPI. Due to this, we restrict four pins from user access

during their declaration that can be used for either scenario, and if there are additional free pins and neurons, we distribute them appropriately.

5.2.3 Repository Refactor

My main goal with this section of the project was to make the repository structure more user-focused than board-focused and to handle the added complexity on the backend. See Fig. 5.1 for the file tree of the original kit and my iteration. In both versions, folders that the user needs to use for a project are blue, along with the paths to get there from the root folder. I only highlighted the paths that would be necessary for a single project, i.e. only one of the two network subfolders is used, in each project.

On the left, the original kit, I made a category of folders called “homes” and colored them orange. Each of these folders contains the broken out list of subfolders as well as anything directly attached to it in the image. On the right, for my kit, I did something similar to show what files exist in the project folders. Each project folder includes the same four files.

App Compile

This is unchanged from the original kit in this regard. The code of the app_compile files is different to reflect the changes in the rest of the repository, but this folder still consists of everything needed to compile C code and networks into UF2 files than can be uploaded to the Pico.

To make it clear to users, all files that are auto-generated, either from copying user code or otherwise, have their filename proceeded by an underscore. It is unlikely that most users will be confused, since the only code that needs to be used in the creation of new projects is the project folder, but I found working with the original kit to occasionally be confusing to remember which file is generated and which is used as the source.

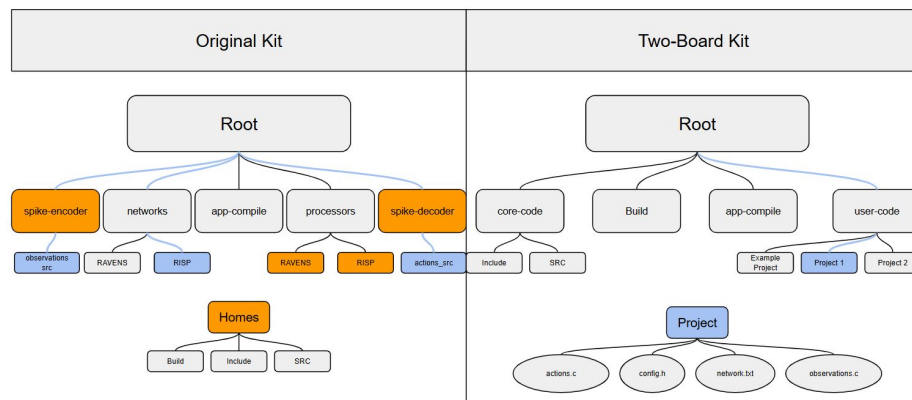


Figure 5.1: File Tree of Each Kit Version

User Code

This is the only folder that most users will need to program within. I provide an example project folder, and the intended use case is that the user copies this folder for their own project and renames it, while maintaining all of the file names inside of the folder. Within the directory there is an actions file and an observations file, which corresponds to the functionality of a decoding unit and encoding unit respectively. Although these functions are now both performed on the CPU, they are different tasks and I thought it would help users to be able to separate them in their minds. This folder also has a config file for the user to specify things that were previously command line flags. The flag options are still present, but I wanted to provide another way of saving parameters for the file. The only exception to this is the *USER_PINS* variable, which cannot be set in the command line and must be declared in this file. When a user goes to build their project, rather than specifying the encoder, decoder, and network individually, they instead only need to list their project folder.

In the original version of the kit the available functions to run were a setup and a loop for both the encoder and decoder, similar to an Arduino sketch, which I assume was the intention. For the encoder, the loop would run once per input at the beginning of the timestep and for the decoder it would run once at the end of the timestep. I maintained this functionality for my iteration, to help with easy translation from one to the other, but I also added asynchronous loops that can be used between timesteps.

Core Code

This is the central location of all code written to run the kit that does not change from project to project. The general flow of the CPU is to run setup code, and begin an infinite loop consisting of:

1. Collect synchronous observations before processing.
2. Perform the mini-loop below this one until encoding and decoding have each been run *SIM_TIME* times.

3. Perform synchronous actions after processing.

The mini-loop mentioned above is the following two steps.

1. If the neuroprocessor says it is ready for the next step, encode and decode for the current step and signal to the neuroprocessor to continue.
2. Regardless, run the asynchronous user functions.

The neuroprocessor runs once per iteration, between reading spikes and writing them back to the CPU. It changed very little from the original kit, other than in communications, because it has little to no direct interfacing with the user. The main change is only in hardware, that it now sends its results to the same board it received input from rather than a different board.

Build

This folder is where the files built by the compiler are placed, both for the neuroprocessor and the CPU. To make the repository cleaner, I also made the compiler remove all build files that aren't UF2. This could easily be changed if any other format is desired, but in daily usage the UF2 files are the only type needed by users, and thus keeping this folder clean was deemed more important than maintaining all output. This takes the place of the four different build folders of the original kit that were spread across the “homes” from Fig. [5.1](#).

5.2.4 Hardware

Two Board PCB

The original plan was to create a carrier PCB to hold and connect the boards, like what the current kit implementation uses. This ideally would have given this kit the same ability to change its neuroprocessor from a microcontroller to a Field Programmable Gate Array as its predecessor. The I/O flexibility we're looking for

made this a complicated request though. There are forty pins on a Raspberry Pi Pico, and because I don't know how many will be used for any purpose, they all must be routed between the boards and to externally accessible pins. This means, in the cleanest case, twenty traces that would have to go around both boards to connect the far rows of pins. This isn't even beginning on the added complication of ensuring that an SPI bus is functionally connected between the two.

Independent Stack

During early prototyping on the breadboard, it was easiest to jump the boards one-to-one, where each pin connected to the same pin on the other board. This had to be changed for the SPI bus, swapping pins sixteen and nineteen which are SPI RX and TX pins respectively. Trying to do this on a PCB would be very complicated, involving a lot of trace overlaps, but fortunately there is an easier solution; stacking the boards on top of each other. This connects the boards one-to-one and reduces the footprint drastically. I also removed the pins on sixteen and nineteen and instead jumped them to each other. The pads connected to the BOOTSEL button were also connected during the assembly to make for easier resets when the button was difficult to reach. There are two versions of this that I made. The first uses a Pico with soldered on standard headers missing the aforementioned pins, and solders an additional Pico to the same pins, below the plastic spacer. The second uses stacker headers on the bottom board and a completely normal top board. Both of these stacks can be seen in Fig. 5.2. In the stacker variation, I included a short pin on one of the synchronization pins so that, if desired, a user could record timesteps on an oscilloscope or similar device.

Expansion PCB

I was worried about how my version of the kit, if using the independent stack, would work in projects like that of Chapter 4 without a better mounting solution. This

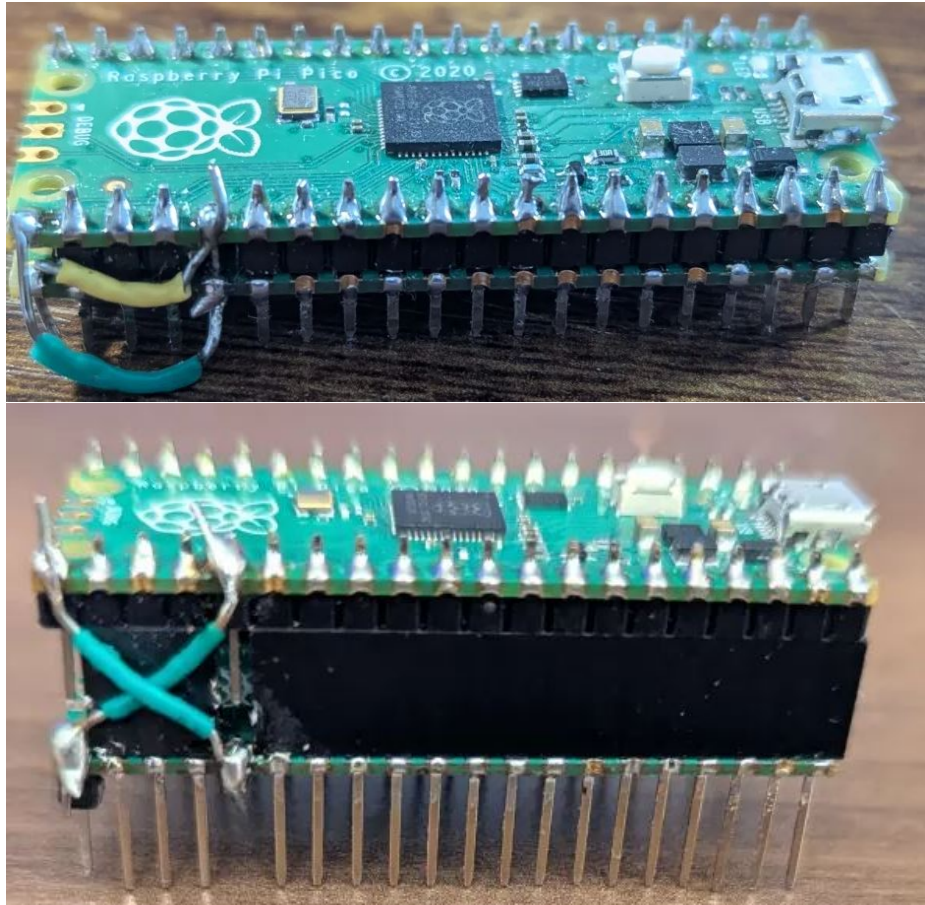


Figure 5.2: Two Board Kit Stacks

led me to the design of a new board. This board isn’t part of the kit, so to speak, but is an expansion to ease in the use of my kit stack. This PCB is essentially a custom breadboard, meant to make connecting peripherals to the kit easier as well as providing a mounting solution for projects that would prefer not to have a floating processor. Because of the nature of the stack, this board works equally well for projects that utilize just a Pico instead of a kit.

5.3 Results

It is necessary to make clear before diving into the results that this project, while finished, is stripped down when compared to the original kit. There is no hardware to connect the CPU to an FPGA and it can currently only run RISP networks. Because it seems like these restrictions apply to very little of the current use of the kit I am confident presenting this as a finished product, but it would be dishonest to promote this project as one that could fully supplant the use of the current neuromorphic kit.

5.3.1 Code Combination

As will be the trend of this section, the summary of the results for combining the encoder and decoder in software is that it is functional and the goal was accomplished. Both units now run off of one Raspberry Pi Pico and all GPIO pins can be used for either input or output, aside from those reserved for communications as mentioned in Section 5.2.2. For the following analyses, I used the binary decimal converter as it exists in the embedded neuromorphic repository, and pressed no buttons. The results are summarized in Table 5.1, where the “Time” column means time per timestep and the “Energy” column means energy to complete one timestep. The first three data columns were measured, and the final two were derived.

Table 5.1: Electrical Specification Comparison of Kits

Kit Type	Voltage	Current	Time	Power	Energy
Original Kit	10.7V	40mA	15ms	0.428W	6.42mJ
Two Board Kit	5V	30mA	3ms	0.15W	0.45mJ

Table 5.2: File Sizes

Kit Type	File	Size
Original Kit	Encoder	61KB
	Decoder	58KB
	Neuroprocessor	53KB
Two board kit without macros	CPU	80KB
	Neuroprocessor	72KB
Two board kit with macros	CPU	66KB
	Neuroprocessor	52KB

I measured the power consumption of the original neuromorphic kit to be 0.428W. In contrast, my kit used 0.15W. just over a third of the power. I have a number of thoughts as to why this is even lower than the two thirds expected.

1. Macros reduce the number of computations
2. Two board communication is simpler, requiring fewer computations and pulses
3. The original kit includes a voltage regulation module

The last point makes it feel like a slightly unfair comparison, but given that the battery voltage isn't accessible to users and thus provides no additional functionality while draining power, I think it is reasonable to include, rather than breadboarding three Picos together to test the efficiency of purely the boards. For the sake of argument, it can be seen in Table 5.1 that even if they used the same input voltage, my kit drew less current and therefore utilized less power.

I tested the speed of the kits by measuring the time that it took them to process ten thousand timesteps. Across five trials, the original kit took an average of fifteen seconds, and my two board variation took approximately three. The timing was very internally consistent, there was not more than a fifth of a second difference between any two readings for the same board. I attribute this to being able to simplify the communication protocols since every message communicates with the whole system. This compounds on the efficiency of the last paragraph because it means that, to accomplish the same fixed-timestep task, my kit will take one fifth the time and one third of the power, leading to a fifteen times decrease in energy.

As demonstrated in Table 5.2, combining the encoder and decoder made the CPU a much larger file than either of them independently, but because of the removed redundancy it is smaller than the combination of both. My neuroprocessor, without macros, is also notably larger than the original kit's neuroprocessor, which I believe is the fault of my code consolidation. Both the CPU and neuroprocessor include some code only used by each other, because I wanted both boards to use the exact

same functions so that the communication could be more easily understood by new users. The size disparity would have been a tradeoff, but by including the macros in my program I actually reduce the neuroprocessor code to a smaller size than the original and make the CPU comparable to either the encoding or decoding unit of the original.

5.3.2 Communications

The voltage spike communication protocol was the first that I developed. It functions as intended for any number of pins all the way down to the minimum of two spike pins and two synchronization pins. The results for the reuse of pins were exactly as expected, a *Uses* of three took three times as long as a *Uses* of one to communicate with the next board. Given the results of Section 5.3.1, and the original kits having eight designated voltage spike pins for the input, my kit can use the minimum of two spiking pins to communicate the same maximum amount of information in just 1.6x the time, while leaving the rest of the pins available for peripherals.

I expected SPI to be much faster, given that it was a dedicated communication interface. I had not taken into account, however, that while voltage spikes may only need to deliver one bit, the SPI interface I took from the the kit transmits packets of thirty-two bits. This makes sense, then, that to complete the same ten thousand steps that I saw in Section 5.3.1 it took the SPI version of my kit thirty-seven seconds, over twelve times the time. The communication is faster, but not enough to outpace the packet sizes. Where the original kit is limited by pins, and has to use an alternate communication scheme for that reason, my pin reuse code means that there is no reason to use the SPI bus unless you have more than twelve times as many input and output neurons as you do available pins. This is a possible scenario but feels like a very niche case.

Since both communication protocols can be used with just four total pins, there is a total of twenty-two GPIO pins that can be used by either the encoder or decoder.

This is only one less than the current number of pins available on the kit and I believe the flexibility offered is worth this reduction, because I think that few projects on the kit would use exactly twelve inputs and eleven outputs, which is the only case in which the original surpasses the capacity of this version.

5.3.3 Repository Refactor

This section is difficult to categorize the results of. The code is functional, requires less navigation to change which aspect of the kit you are working on, and prioritizes user experience over emulating the structure of the hardware. However, the goal of this section of the project, to make a more intuitive system, is an inherently subjective one, and so evaluating it cannot be accomplished objectively.

5.3.4 Hardware

The lab as a whole seems to have issues with FPGA functionality on the kit, and thus the added benefit of having that option wasn't worth the additional development time required. Once this was no longer part of the goal, a PCB began to lose significant favor in comparison to the stack. There is an additional factor to consider, which is that my kit is not the primary development track, and I am trying to work within the confines of what the main kit does. The alternate communication methodology for the main kit changed since I began my project, from SPI to UART, which would require an redesign of the PCB to make it work.

In comparison, a stack could just use another header layout and save the time of ordering new PCBs. The first stack option, with normal headers, works fine. It takes up the same space as a normal Pico and grants access to everything a user may use. The bottom Pico does not have an accessible BOOTSEL button, but since it is connected to the top's button, it can still be easily reprogrammed. This method does make reprogramming clunky though. When reprogramming one, both enter boot select mode. This means that the whole unit must be rebooted afterward to run.

In comparison, for the current kit, when you upload new code to one unit, it will restart and the others are already ready. The other issue, although slightly outside the purview of the project, is disassembly. This is never necessary when using the kit, but if a user wanted to scavenge the Picos by separating them it would be difficult, as there are thirty-eight soldered pins connecting them and three jumpers. The biggest issue that I found with this implementation of the stack is that the pins coming from the bottom are too short. They must go through a Pico and the connecting solder joints on the long side of the plastic header, and as such many connectors can't find purchase and the pins cannot be used with a breadboard.

The stackable headers work as well as hoped and solve all of the issues of the basic stack. The bottom Pico's button is accessible using a paperclip or similar item, or can be bridged to the top, whichever is preferred. The top Pico has normal headers and is only soldered to the bottom via two external jumpers. These are easily removed, which means one Pico can be salvaged with little issue, and although the bottom uses nonstandard headers, they can be used just as well for most purposes. Those nonstandard headers are also designed expressly for this purpose, so there is extra height added that allows for this stack to connect better to peripherals.

In both stacks, both boards can be powered from the micro-USB port of either. In fact, the boards are completely interchangeable, either can be programmed to be the CPU or the neuroprocessor. I advise using the top board as the CPU so that the status LED is more easily visible, but they are functionally interchangeable. The largest issue with the stack is that power must be routed through the Picos in the stacks. This makes the use of 12V peripherals much more difficult.

The expansion board is uninteresting. It works like a breadboard, breaking out the pins of the kit and leaving power rails of 5V and 3.3V. This makes it useful for mounting, giving the user drilled holes on the corners to put bolts through, but only slightly more useful than a breadboard. Ultimately the time investment to have these boards made was not worth it, and I stand by that decision, though I think it could be helpful in some circumstances.

Chapter 6

Conclusions

6.1 Summary of Collaborations

As discussed in Section 2.4, the basic app of Chapter 3 was heavily based on previous work, and Louise Fry was responsible for translating her previous work into the swarm application. She also made the prototype IR bodies in Fig. 3.4 and did preliminary hyperparameter tuning.

Will Buziak did the majority of the CAD work for the robot. This includes the original chassis and motor assembly, along with altering the file for the printed omnidirectional wheel. Will helped with assembling the first version of the robot.

The motor assembly overhaul happened as Will was leaving UT, and Daniel Archer stepped up to help me with the new CAD requirements. He designed and fabricated all currently used pieces of the new motor assembly, and altered Will's chassis file to work with the new motor placement.

Luke McCombs and Bryson Gullet both helped with debugging the integration of their work into these projects. Luke with adding multiobjective optimization to the swarm application and Bryson with using the kit for the SALMON robots and deploying networks to it. Bryson additionally shared the kit circuit diagrams to help

me make the power distribution board and answered all of my questions about the inner workings of his kit code while I was creating the two board kit.

6.2 Conclusions and Contribution

In Chapter 3 I expanded on the depth and abilities of the simulation made by Louise Fry. I changed the application from a single objective fitness function that measured only cohesion and collisions that used holonomic agents with fixed angles to a multiobjective fitness function utilizing many different fitness metrics and fully randomized agent starting positions. I also added much more flexibility to the application, allowing for higher input density, smaller field of view, and more tunable fitness weighting. I reformatted the ASCII visualization, added a GIF creator for that, based on an example in the repository, and created a logging function to watch the paths of the networks as a still image and across multiple training runs. The GIF generator and path logging make it much easier to manually review agent behavior than the original viewing solution.

In Chapter 4 I helped to design and map out the robot, created the PDB, wrote all of user-given code for the kit. I helped choose components for the original drivetrain of the robot and did choose the components for the second version of the drivetrain. I did the majority of assembly and wiring for the robots, including the Y-split wires to control the motor encoders and allow them to switch from planar driving to traditional driving. I created a minimal C library for the LiDAR used on the robot.

In Chapter 5 I took the code used by the original kit and performed a large overhaul. I refactored the repository to be more user-friendly and combined the encoding and decoding units into a single board. I also standardized the functions that the two boards used for communication, creating an easier environment for new users to become accustomed to. I created a hardware solution for my kit that requires no additional PCB, very few wires, and minimal soldering to be reversed in the case of disassembly.

6.3 Discussion

6.3.1 Swarm Application

This component of the project has the most opportunity for future work. I will elaborate in Section 6.4.1, but essentially this project is incredibly open, and thus it feels like more could always be done. The last fitness function we investigated in Section 3.3.2 opened a lot of doors for future work, with a different understanding of the goal and a different method of picking the best agent, but it also gave us a very good stopping point for the project, where we saw interesting emergent behavior from the agents, who acted vastly differently to one another just because of their starting conditions, even though they all have the same brain.

Applying this simulation to the hardware has also gone well. The specifics are discussed in Section 6.3.2 due to their applicability to various hardware deployments, but overall I am happy with the results. The robots clearly imitated their simulated counterparts, and I consider that a success.

6.3.2 SALMON

In terms of a physical robot, the SALMON platform represents an important and previously unrepresented space for our lab of a quick and easy robot testbed. We have other robots that people can use for their projects, but they all enforce more restrictions on the project at hand, such as a drive scheme and sensor suite, than SALMON does. SALMON's production process makes it easier to fix if something happens than robots in the lab where the entire chassis was purchased as a single item. I believe that filling this role is still important, even if it is not a perfectly precise machine, occasionally having drift issues.

This robot uses the kit as opposed to a SBC, which should make the transition to hardware less daunting than needing to devise a new embedding system as has

previously been required. I hope that this encourages more students in the future to try to bring applicable control applications to hardware with this system.

6.3.3 Two Board Neuromorphic Kit

Everything that I tried to add to this kit worked, and it accomplished all of the goals that I set for it. In voltage spike mode it performs five times faster and with three times more power efficiency than the original kit, which is much better than I had hoped for in both respects. Making the SPI bus work was the most arduous task of the process, but given that this was enough of an issue to leave it in favor of the much slower communication protocol, UART, on the main kit, I am proud to have finished it on my end. I am very happy with the idea of the independent stack, because I think that making the board act more similarly to a microcontroller from the outside is a boon, and with this it can work anywhere that is made for a single Pico.

I hope that some of the changes I made, such as the repository refactor and asynchronous loops, are considered for implementation on the main kit, because I believe they are general improvements that don't need the rest of my kit to work.

6.4 Future Work

6.4.1 Swarm Application

Because the swarm application is so open ended, it still has a lot that I would be interested in adding if given the time. I want to work on the robustness of the simulation and its resilience to change. This includes a wider variety of rooms with obstacles or non-rectangular boundaries. I believe that adding this complexity will help to further refine the fitness evaluations, as there is promise in our current iteration but it can be improved.

The largest gap in the current simulation and experimentation is the lack of alternative encoding and decoding methods. I'd like to run a large batch of

experiments on different commonly used encoders and decoders to see if any of them can improve performance or maintain performance while reducing activity.

Ultimately this app is very open ended and I could experiment with different parameters for a long time. There is a lot left that could be done but not necessarily anything that needs to be done.

Additionally, the IMU integration into the input space is still underway, but should be complete relatively soon after the completion of this paper.

6.4.2 SALMON

All work that I believe could or should be done on this project can be divided into two camps. The former category is all built on revising the Power Distribution Board, and the latter is quality of life improvements via 3D printing.

A revised PDB would fix the two errors mentioned in Section 4.3.4, the nonfunctional LiDAR port and integrated motor driver. This would simplify the wiring to make robot maintenance more intuitive, reduce the number of parts needed to assemble the robot and allow for a smaller footprint if desired. I would also add a port for an IMU, because I think that it is another sensor that is likely to be used by many projects, and adding it to the PDB increases its stability compared to having it mounted to the printed components directly. This was not accomplished during the span of the project because the shipping time for new boards is prohibitively long and the motor driver inclusion would have required a redesign of the chassis to use.

There are a few small changes that could be done on the printed components but were not because they are largely non-functional and I am not very familiar with 3D modeling. The changes would include rotating the body pillars 180° to not block the network microcontroller, removing the battery power switch, and adding an indicator on the sensor hat to show which direction is the LiDAR's 0°.

It would also be interesting to make a version of the PDB that uses the two board kit from Chapter 5, with or without the expansion board, likely without. This

could also decrease the minimum robot size, allow for more peripherals, and allow for minimal wiring by implementing the required traces directly into the port on the PDB. This is not the path that I would recommend the SALMON project take, because in its current state there are limitations of the two board kit, and the default kit is still being worked on, but I do think there is promise to this, so I made a simple version of what this circuit would look like, seen in Fig. [6.1](#).

6.4.3 Two Board Neuromorphic Kit

Everything that I think could be improved about this project is to regain some of the functionality of the original kit that I sacrificed to accelerate my design process. I would like to add RAVENS functionality and recreate the UART communication that was added to the kit after I split off. I also would like to see if the recent work in the lab of making the framework embedder is compatible or warrants adding to my version of the kit.

I think that it would be interesting to port this, and the default kit for that matter, onto the recently released Raspberry Pi Pico 2. It is faster, has a floating point unit unlike the first, and if we used it for an embedded system, like the one listed in the next paragraph, its chip, the RP2350, has more available pins than the RP2040.

At one point early in the process I had toyed with the idea of an embedded version of this kit, where RP2040 chips were used in place of Picos. I think this is very interesting . My plan for this was to make a PCB with two chips on it and a single USB port with a bus controller on the board. I would also make a script that would copy one UF2 onto each chip. I think this would be a very cool board for the lab to have, but it has the same pitfalls as other PCBs, with a long production and shipping time and becoming obsolete when the code needs change. I think this would be a great future project if the lab is interested in continuing work on this branch of the kit's evolution.

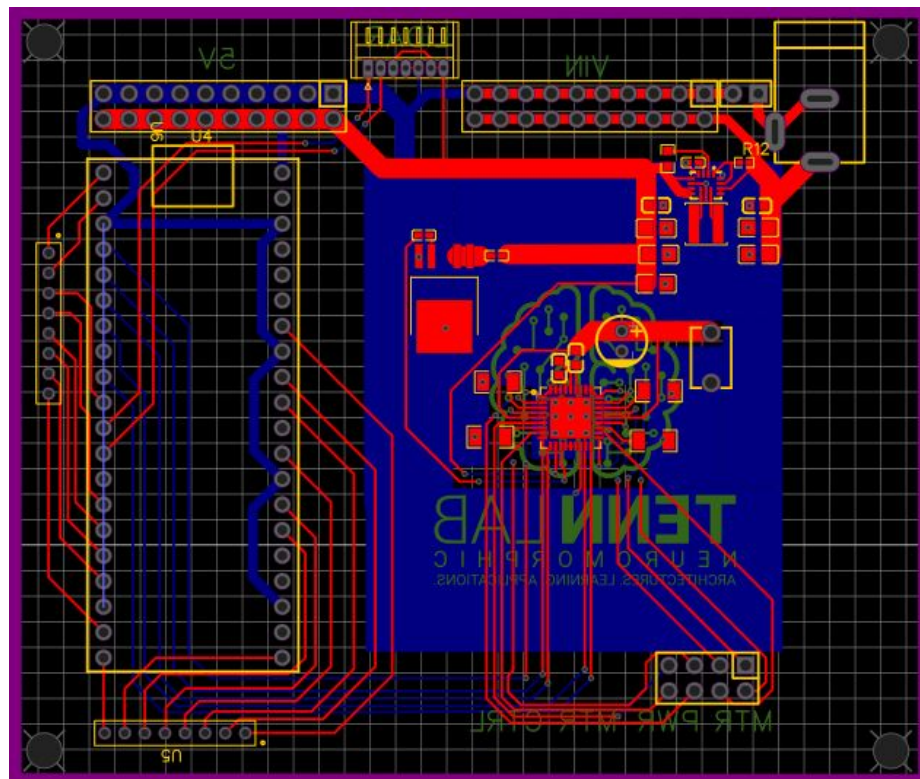


Figure 6.1: PCB Design of Power Distribution Board with Two Board Kit

Bibliography

- Fang, Y., Wang, Z., Gomez, J., Datta, S., Khan, A. I., and Raychowdhury, A. (2019). A swarm optimization solver based on ferroelectric spiking neural networks. *Frontiers in neuroscience*, 13:855. [4](#)
- Panichella, A. (2022). An improved pareto front modeling algorithm for large-scale many-objective optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '22*, page 565–573, New York, NY, USA. Association for Computing Machinery. [6](#)
- Plank, J. S., Gullett, B., Foshie, A. Z., Rose, G. S., and Schuman, C. D. (2022a). Disclosure of a neuromorphic starter kit. *arXiv preprint arXiv:2211.04526*. [7](#)
- Plank, J. S., Schuman, C. D., Bruer, G., Dean, M. E., and Rose, G. S. (2018). The tennlab exploratory neuromorphic computing framework. *IEEE Letters of the Computer Society*, 1(2):17–20. [5](#)
- Plank, J. S., Zheng, C., Gullett, B., Skuda, N., Rizzo, C., Schuman, C. D., and Rose, G. S. (2022b). The case for risp: A reduced instruction spiking processor. *arXiv preprint arXiv:2206.14016*. [5](#)
- Schuman, C. D., Mitchell, J. P., Patton, R. M., Potok, T. E., and Plank, J. S. (2020). Evolutionary optimization for neuromorphic systems. In *Proceedings of the 2020 Annual Neuro-Inspired Computational Elements Workshop*, pages 1–9. [6](#)

- Schuman, C. D., Plank, J. S., Bruer, G., and Anantharaj, J. (2019). Non-traditional input encoding schemes for spiking neuromorphic systems. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10. IEEE. [5](#)
- Zhu, K., Mattson, C., Snyder, S., Vega, R., Brown, D. S., Parsa, M., and Nowzari, C. (2024). Spiking neural networks as a controller for emergent swarm agents. In *2024 International Conference on Neuromorphic Systems (ICONS)*, pages 319–326. [5](#)

Vita

Drew Friend grew up in Monroe, Ohio. After high school he attended the University of Tennessee and received a Bachelor of Science degree in Computer Science. During his undergraduate studies he began work at TENNlab and began the accelerated Master of Science degree program. Upon graduation he swiftly transitioned into his graduate program, pursuing a Master of Science degree in Computer Science, with a concentration in Intelligent Systems and Machine Learning. His research interest includes robotics and spiking neural networks, specifically with a focus on their use as an alternative to the Von Neumann architecture.