# Autonomous Directional Rotary Artificial Intelligence Navigational System (ADRAINS)

**David Drew djd36, Joanna Dai jxd2**

## Introduction

Our final project in ECE 476 is a mobile robot with a developed neural network such that it evolves to avoid collisions into a circular vertical white wall while traveling at the fastest speed and straightest line possible without human intervention or external computers.

The completion of this project required extensive capacity and application on both hardware and software ends. In constructing the robot, we needed to build the custom prototype board, apply infrared sensors as neural inputs, implement stepper motors for robot motion, and provide a mobile power supply to the MCU. The purpose of these design factors is to allow the autonomous movement of the robot while minimizing the size of our robot, to accurately sense distance and collisions into the white wall of our arena, and to calculate the velocity precisely while providing sustainable torque to move our robot. On the software end, we needed to execute an evolutionary spiking neurons algorithm that interfaced with our hardware. The purpose of this was to integrate a spiking neural model with infrared sensors as inputs and motor speeds as outputs to determine robot velocity and direction. We also implemented the evolutionary model based on assessing random individuals of a randomly generated population through a fitness equation and improving the population by discarding the worst individual in the population with the worst fitness. The fitness equation measured by the velocity of the robot, the direction change, and the amount of activity from sensors.

## High Level Design

### Rationale and Sources

While our partnership coalesced from our strong interest in projectile devices and their implementation through use of microcontrollers, which is why we signed up the class initially, we were unfortunately denied of such opportunity as a final project.

Nonetheless, we wanted our final project to be a valuable challenge that tested our comprehensive knowledge of electrical and computer engineering. We wanted a final product that was significant to society, unique to the course, beyond the scope of previous labs, unprecedented by previous final projects, and realistic to our abilities.

Therefore, after consulting with TA Idan Beck as well as previous and current students of this course, we decided to build this robot with a genetic evolutionary algorithm.After speaking with Professor Land and previous students of the course regarding the feasibility of our project and workload involved, we formed the following objective:

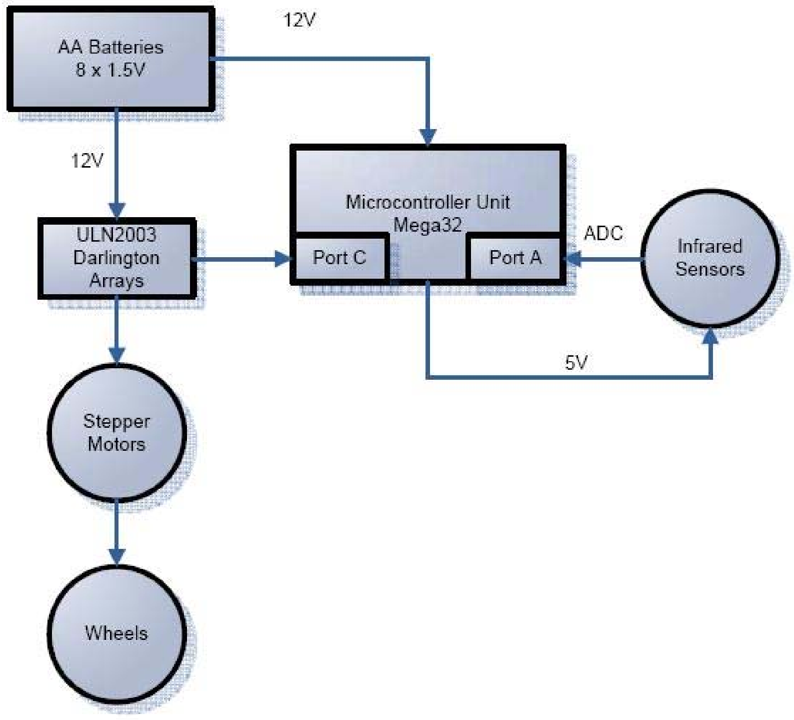
- To produce a mobile robot with the neural network that autonomously avoids collisions into the white wall of a circular arena.

## Background Math

We had no complicated derivations or integrations were involved in the calculations of this project. The math background needed for this project was geometry for stepper motor calculations, conversions to calculate velocity, timers, analog to digital, measurement units, algebra for analysis of the fitness equation and characterization of polynomials for infrared sensor behavior, and probability to analyze infrared sensor data, their polynomial characteristic equations, and corresponding least squares approximation and residuals.

## Logical Structure

We used 8 AA batteries at 1.5V each as our mobile power supply. The 12V powers the Mega32, the Darlington arrays, and the stepper motors. The stepper motors are controlled by the Mega32 through Port C. The infrared sensors are powered at 5V and go through the ADC through Port A.



## Hardware/Software Tradeoffs

After consulting with Bruce on possible approaches of neural theory, hardware, and software considerations for our neural network robot, we came across one significant hardware-to-software tradeoffs. Bruce introduced to us Hopfield and Hebbian neuron spiking networks, both of which require nonlinear amplifiers to simulate the analog inputs and spiking. He also referred us to Dario Floreano's paper, "Evolutionary Bits'n'Spikes" (see References), that described a spiking neuron model algorithm based on incoming spike contributions to the connectivity of the network and an evolutionary model improving its population based on a simple fitness equation of speed, amount of collisions, and amount of infrared sensor activity.

Between implementing the spiking neural network on hardware or software, we chose software. We felt that hardware would take more tedious manual labor, be harder to debug, and create extra weight on our robot. With software, we can compile and debugged efficiently on CodeVisionAVR and have flexibility in adding and modifying our code.

## Standards and Patents

We had no patents to adhere to in this project.

In terms of patents, lab TA Idan Beck gave us "Artificial Life IV", a book with compilations of published papers on artificial intelligence and genetic algorithms. We got our idea of building an autonomous evolutionary robot from there, particularly the copyrighted Khepera robot. However, Floreano's experiment and paper provided such substantial background on neural and evolutionary models for our project plan that we significantly based our final project off of his work. In adhereing to intellectual property, we exchanged emails for Floreano regarding permission to use the content described in his paper and contacted him for help when we came across a road block regarding initialization of our robot (see Intellectual Property Considerations).

## Hardware

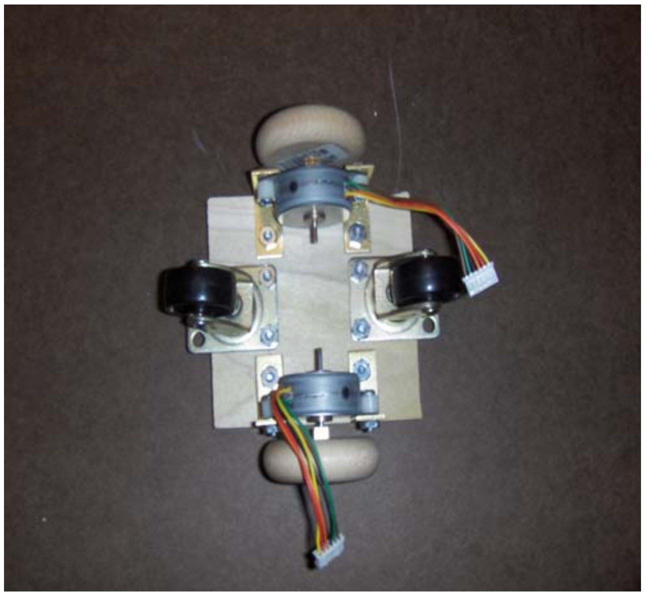
Components required for our hardware design:

- Structural design
    - Wood base, wooden wheels, free rotating wheels, foam core
- Custom prototype board and Atmel Mega32 microcontroller unit
- Stepper motors and Darlington arrays
- Infrared light-emitting diode and phototransistor
- Power supply

### Structural Design

The design of the robot was circular for the purpose of better interaction with our software in developing the neural network. The symmetry of this design allows the robot to detect infrared sensor inputs with more accuracy when approaching the wall and more balance between the left and right sides, in comparison to square or rectangular. The circular shape gives better turning radius in our arena, especially when the robot is in close proximity to the wall and needs to make a quick turn.

A mechanical challenge for the design was finding all the suitable parts to assemble the rectangular base of our robot. We used two wooden wheels that attached perfectly to the stepper motors. Two more free rotating wheels were used to keep balance and permit mobility of the robot. We used L-brackets, nuts, bolts, and spacers to install the stepper motors onto the wood base.

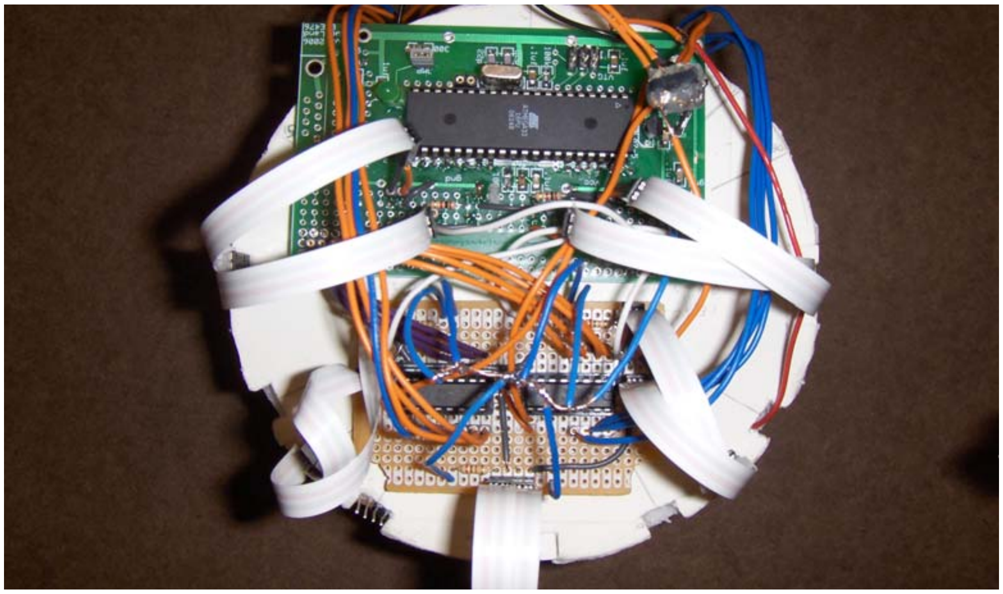


To minimize the robot’s size, we created two layers: one for the power supply and one for the circuit boards. Thus, the foam core circles were no larger in diameter than the protoboard and solder board side by side.

### Prototype board and Atmel Mega 32

The protoboard was assembled by following the instructions on the protoboard design page. Because the robot was running off batteris and not the AC power supply, we connected our 12V battery supply directly to the power ports on the protoboard. To program the Mega32, the 6-pin programming port was used throughout this project. Pins D.0 and D.1 of our Mega32 were used to display values through hyperterm.

The Port C pins were used to connect the Mega32 to the input of the Darlington arrays for the stepper motors. Port A was used to converting the analog voltage from the infrared sensors to the digital signals as inputs for our program. The remaining available pints on the protoboard were used to lay out the infrared sensor circuits.

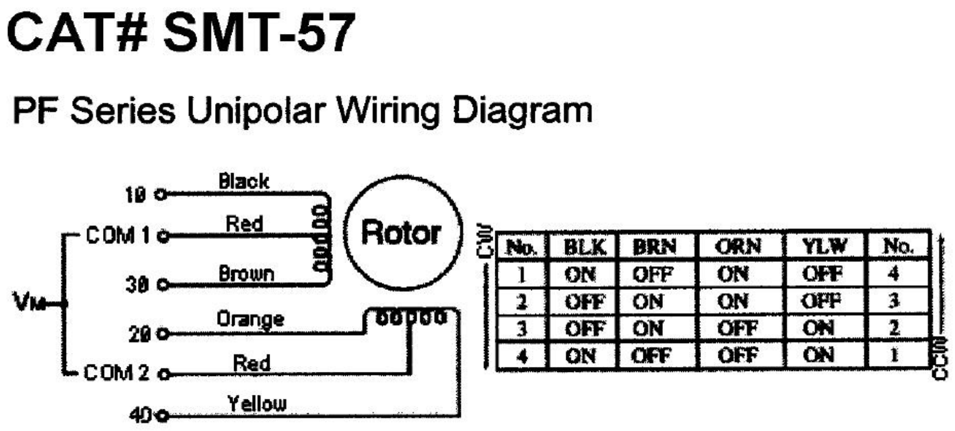


## Stepper Motors

The robot used PF35T-48L4 stepper motors to move forward, backward, or turn. The reason we chose stepper motors is because they offer precise control over the speed and direction of the robot. Because velocity of each wheel is an output of the neural network and is used to compute the fitness equation of each individual in our evolutionary model, the stepper motors' accuracy was a key factor in our hardware design.

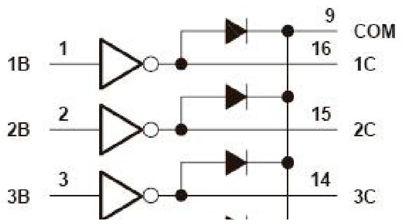
Stepper motors move at 7.5 degrees/step with one step at a time because each step in the motor draws 500mA at a time. To calculate the velocity of our robot, we know that 360/7.5 gives us 48 steps/rev. The diameter of our wheel is 2 inches, so (2 inch * 2.54 cm/ in * pi / 2) = 15.9593 cm/rev. Velocity is then 15.9593 cm / rev * 1/48 rev / (t2 / 1000 s) = .332485 cm / (t2/1000 s). Therefore, we could control the velocity of our robot by determining the value of t2.

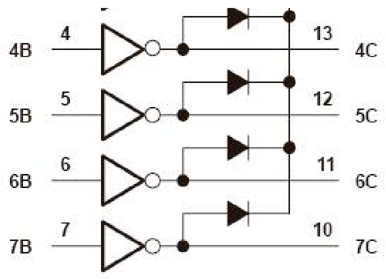
The red and green wires of our stepper motor are connected to our 12V battery supply. We connected left motor pins C.0-C.3 to orange, yellow, brown, black, respectively. Likewise, we connected right motor pins C.4-C.7 in the same color order.



| No. | BLK | BRN | ORN | YLW | No. |
|---|---|---|---|---|---|
| 1 | ON | OFF | ON | OFF | 4 |
| 2 | OFF | ON | ON | OFF | 3 |
| 3 | OFF | ON | OFF | ON | 2 |
| 4 | ON | OFF | OFF | ON | 1 |

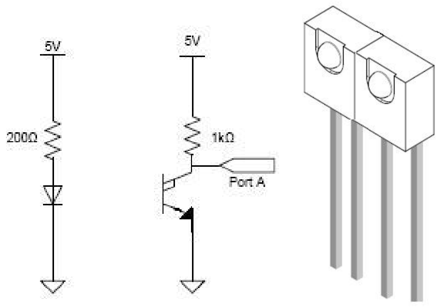
We needed to use Darlington arrays to connect the Port C pins as inputs from the Mega32 into the ULN2003AN and the outputs to the stepper motors. Darlington arrays output high voltage with common-cathode clamp diodes for switching inductive loads. Each pair operates at 500mA, which matches the current of each step in a stepper motor.





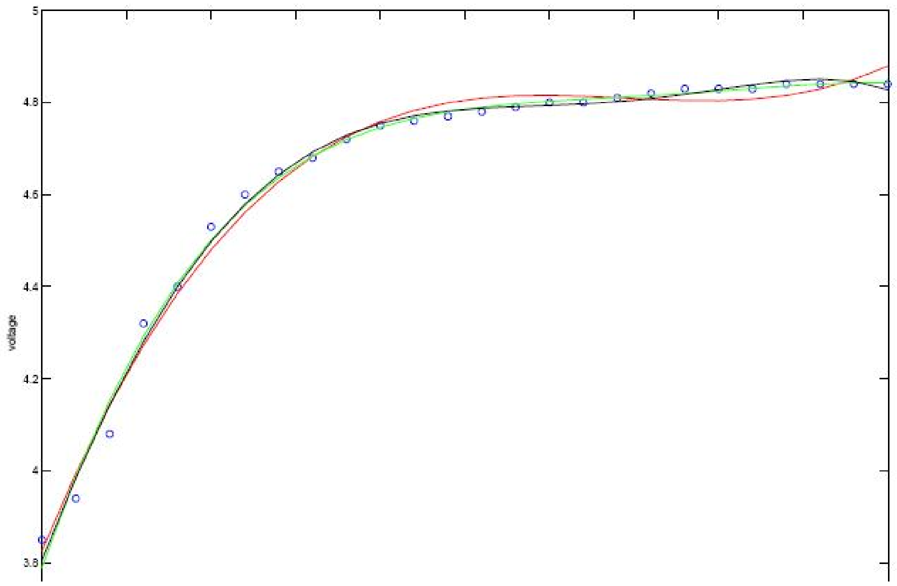
## Infrared Sensors

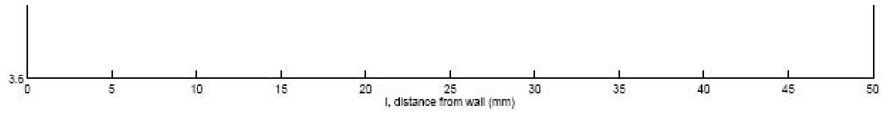
For our distance sensing, we paired Infrared Phototransistor QSE113-ND with Plastic Infrared Light Emitting Diode QEE122. The sidelooker infrared sensors operate at 880nm. We connected the node at the collector of the phototransistor to Port A for the analog to digital conversion.



After testing the infrared LED and transistor on white breadboards, we found that we got optimal distance sensing when the two components were placed as closely to each other as possible. In assessing the infrared sensors' impact to the input of the neural robot, we created four ranges based on the characteristic curve of the IR sensors. We saw that there was a constant .01V drop from >25cm and characterized that as the top range. From 15cm to 25cm, we see a more drastic voltage drop and call that our middle range. From 15cm to 5cm, we see a steeper drop and call that our bottom range. Anything less than 5cm is defined as collision.

The graph below is a Matlab plot of our data. We used the function polyval() to approximate a polynomial equation. Below, red was n=3 polynomials, black was n=4 polynomials, and green was n=5 polynomials. We also used the function polyfit() to find the difference between each curve and our data. This created the residuals for each function and helped us determine the most accurate curve.





While this method gave very precise characterizations of our sensors' behavior, we found that the infrared sensors were very sensitive to room temperature and any slight difference in setup. Therefore, in the software we needed to take values the day of and use the robot in accordance to the room temperature that day.

### Power Supply

For our power supply, we used 8 Duracell AA batteries connected in series. We needed to use lighter batteries to keep our robot weight down while allowing it to move freely. Another hardware concern for the batteries is drawing amps for the motor so that the voltage stays at approximately 12V and the does not drop. Because each motor step draws 500mA and there are two motors, we used AA batteries because we knew they supply voltage at 1.00A. The batteries are attached to the second layer of our robot, shooting 12V to the power ports on the protoboard and to the motors.

## Software

Our software was based on a time based scheduler scheme with time-critical functions in an Interrupt Service Routine. The design was based of a paper called Evolutionary Bits'n'Spikes by Dario Floreano, Nicolas Schoeni, Gilles Caprari, and Jesper Blynel. The paper contained details on how to implement a spiking neural network. We split our program into six tasks with one ISR. The ISR was a timer0 compare match ISR set to occur every ms. Its sole purpose was to update task timers to keep them at millisecond accuracy. The first task was the neuron implementation which updated the neural network every 2 ms. The next two tasks controlled the left and right stepper motors, each causing the motor to step once. The task timers for these were variable based on the neural network's spikes. The fourth task, which was executed every 20 ms, read the infrared sensors and updated the speed of the motors by changing the task timers. This task also calculated and accumulated fitness values. Our fifth task evolved the neural network based on the fitness accumulated in task 4 and was executed every 14s. Our final task was executed every 3 minutes and updated the EEPROM with the current population so that the robot could be turned off without losing what it learned.

The neural network was created to be fast, efficient, and cheap. Each run, a population would be created of 6 individuals. Each individual would have certain variables assigned to it. First is the variable signs, which gave the sign of the neurons for that individual. The variable is one byte long, with each bit corresponding to one neuron. The individual would also have a byte of nconnection for each neuron (for a total of eight bytes) which describes the connections from neurons to the one neuron. Likewise, the individual has a int of sconnection for each neuron to describe the connections from sensors to that neuron. This could not be fit into a byte due to the large amount of sensors we used. The individual also has its own threshold value. We decided that it would be best to make the threshold value a part of the evolution, instead of setting it to a certain value, since the way certain networks evolve may benefit more from one threshold then another. Finally, each individual had a fitness value. The neural network itself had variables to hold the current individual, a byte for the state of each neuron called neuron (spiking or not spiking), an int for the state of each sensor called sensor, and the membrane potential of all neurons called memb. To allow for easy program change, the majority of these variables are based on defines numindivid, numsensor, and numneuron. Any change to these will cause variables to adapt to the new number, though there are a few exceptions where additional lines of code may need to be added, especially if going above the amount of bits that a variable type can fit.

Our first task is the neural network, which operates on one individual at a time. This individual remains the same until 14 seconds when a new individual is randomly selected and mutated. The network implementation begins with checking to see if the sensors have been updated. If not, the sensor variable is set to zero. This allows the network to keep generating internal spikes without having to wait for new sensor input. The network then checks to see if the individual has just been evolved; if so it spikes the two neurons associated with forward motion of the left and right motors. Next the Refractory Period is taken into account. If a neuron spiked in the last run through of this task, it does not update its membrane potion or count the contribution of incoming spikes; it moves immediately to spike generation. For neurons not in the refractory period, the membrane potential is updated by calculating the contribution of spikes from other neurons/sensors. The fact that sensor/sconnection and neuron/nconnection/sign are all the same length allows these values to be ANDed together and bit masked/shifted to calculate the appropriate contribution to memb. Each spike adds/subtracts 1 to memb. After updating the membrane potential, the software then checks to see if the potential is greater than the threshold. We generate a random number between -2 and 2 to add to the threshold to prevent locked oscillations. If the neuron is spiking, we flip the corresponding bit in the variable neuron and reset the potential to zero, otherwise we clear the corresponding bit. We next check to see if any of the motor neurons are spiking, if this is the case we increment our variables to count the number of spikes per motor update cycle. Finally, we allow for leakage in the network by decrementing all the membrane potentials.

The second task controls the left motor. Our stepper motors move 7.5º per step, so there are 48 steps / revolution. The circumference of our wheels is 15.9593 cm so our velocity equals 15.9593 cm / rev * (1/48 rev)/(t2 ms / 1000 (ms/s)), where t2 is the task timer for task 2. This equation ends up being velocity = .332485 cm/(t2/1000 s) where t2 is in milliseconds. Knowing this, we can control our speed to a very fine precision by just changing t2, which occurs in task 4. The stepper motors are hooked up to PORTC and each step sets a different bit between C.0 to C.3 to step the motor. The motor can be stepped either forwards or backwards which gives us absolute control. The third task is almost identical to the second, except that it uses C.4 through C.7 and the order it goes through is different because the motor is facing a different direction. The variable DirL/DirR controls the direction and the way stepL/stepR is incremented or decremented.

The fourth task was one of our most complicated ones. Sensor updating was a long and complicated process, as was motor control and fitness calculation. Our sensor updating scheme begins with setting collision to zero to indicate that there are no collisions, which is an important factor in fitness calculation. We read all sensors through the ADC, and we used 10 bit precision instead of 8. Voltage is calculated using floating point variables by multiplying the ADC input by the reference voltage, 5V, and dividing by 2 number of bits. We change ADMUX after each read, and wait until the ADSC bit is set before reading the next value. Each IR sensor has a unique characteristic, and as a result it became important for us to constantly reading the next value. Each IR sensor has a unique characteristic, and as a result it became important for us to constantly test the sensors to see how the distance-voltage values change. We use defines at the top to measure certain range values. The values we chose is nothing (sensorxnot), 25mm (sensorxtop), 15mm (sensorxmid), 10mm (sensorxbot), and 0mm (sensorxcol). We have if statements for each sensor that will set between 0 and 3 sensor bits in the variable sensorread, which will become sensor once all sensors have been read. In addition, the variable sensoractivity is changed for each sensor so we can calculate the sensor with the highest activity for fitness purposes. The variable collision is also set if any sensors are or below the value for 0 mm. Finally, to allow some sensors to spike even with nothing in range to keep the network going, we add a noise factor. If the range is between sensorxnot and sensorxtop, which would normally spike no sensors, then we calculate a probability and random value based on that range and if the value is less then the probability we set a bit anyway. We repeat this process for each sensor.

At the end, we also calculate an additional ‘sensor’ which is an open ADC socket. This is used because of the rand() function. This function gave us a lot of trouble throughout the project. Because of its pseudorandom nature, it was generating the same networks and same values again and again, making our network very hard to mature. A neural network requires many very random values to function. Initially, we tried to add up all the ADC inputs and average them, and use that value as the seed. This turned out not to be random enough, so we found a better solution. Since the final sensor doesn’t have anything in its port, it is just noise. We predict where that noise will lie and then we use the same probability calculation as before by taking into account how far into the range of noise it lies. We then amplify this probability by 1000, making the noisiest part of the number more prominent, and use that as a new random seed. This turned out to be much more effective for all cases except initialization, where the average ADC values proved better.

After examining all sensors including the noise sensor, we calculate the sensor with the most activity for the fitness equation. Lastly, for an additional amount of randomness, we add even more sensor noise by using the random seed calculated by the noise sensor to potentially set a few more bits in sensorread. At the next execution of task1, sensor will take on sensorread instead of 0.

The next part of task 4 is to update the motor speed. We initially did this every 20 ms along with sensor updating, but we found that this did not allow enough time for the motors to even execute if their timers were constantly being updated. It usually ended up being updated with 0 velocity which would max the amount of time the motors would take to step. To fix this problem, we added another counter to the motor algorithm. This counter was set at the end of the motor update to the highest of the two task timers, as long as they were less than 350 ms which is approximately 1 cm/s, otherwise we settled on 60 ms which was slightly more than 5.5 cm/s. The motor speed calculation involved using the spikes in the last 20 ms to calculate speed. If forward spikes where higher than backward spikes, the motor moved forward and velocity was equal to the different over the total number of spikes multiplied by 8, the max value of velocity (8 cm/s). If forward spikes were smaller than backward spikes, then the motors moved backward and velocity was calculated the same way except with forward spikes being subtracted from backward spikes. After velocity was calculated, we used the velocity equation given above to calculate the task timer. We did this separately for both the left and right motors. If either motors direction variable was backwards, then fitness would be set to zero.

The final part of task 4 was the fitness calculation/accumulation. If either wheel was moving backwards or if there was a collision, there was nothing added to the fitness. Otherwise, the fitness = fitness + (v2 + v3)*(1.0-((v2-v3)/8.0))*(1.0 – sensormaxactivity)). V2 and v3 are the absolute value of velocities of the left and right wheels respectively. We divide by 8.0 in the second part because that is the max speed. Finally, sensormaxactivity is a value between 0 and 1 (0 nothing in sight, 1 a collision). So, fitness will be maximized when v2 and v3 are largest, but equal, and there are no walls in sight.

Task5 performs the evolution, selection, and mutation of individuals. It begins by calculating the worst individual, in terms of fitness, in the population. It then checks if the new fitness calculated in task4 is greater than or equal to the worst individuals fitness and greater than or equal to its former values fitness. If so, the new fitness is kept along with the signs, sconnection, nconnection, and threshold that belong to it. Otherwise, the old values are restored. Next, an individual is picked at random and its signs, sconnection, nconnection, and threshold are backed up in variables. The task then mutates the current values by XORing a 1 with a randomly selected bit in each of the four variables. However, since changing the threshold can have such a drastic effect, and the threshold has to be kept between 2 and 6, if the mutation doesn’t happen to occur within that range, the threshold value is not changed. At the end, the task sends a message to task1 telling it that an evolution has occurred.

The final task handles the EEPROM for this program. In case of power loss or if you want to take a break, it is best to allow the population to not be destroyed when the power goes off. As a result, every 3 minutes, the current population is stores in EEPROM. In addition, if a member of the current population is the highest fitness value so far, it is saved in EEPROM as well to keep track of the best individuals.

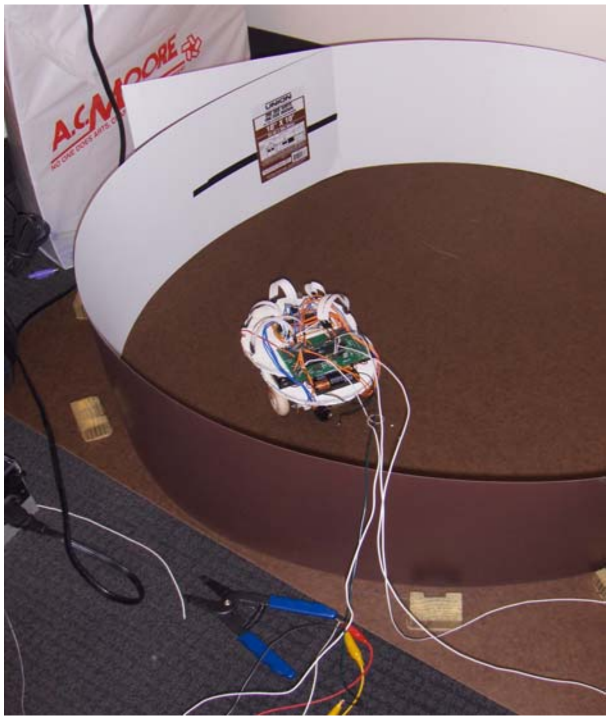
The rest of our program is fairly standard. Main simply calls tasks when their task timers are at 0. Initialize makes PORTC the output for motors. Timer0 is initialized to be a 1 ms timer, by setting OCR0 to 250, and TCCR0 to 0b00001011 which sets the prescaler to 64 and enables CTC. Hyperterm is set up for debugging. The ADC is set to use AVCC, be left aligned, start at ADC0, but not begin the first conversion. A random seed in then generated by averaging all the current sensor values. This is necessary since the next step is to randomly initialize the majority of the neural network variables. If the robot has already built some of its network, EEPROM will replace the randomly generated values. All variables used in the various tasks are then intializied. Finally, if the robots network is partially built, the EEPROM will output the best individual and all the best fitness values. This allows us to examine what a good network looks like and see the progression of fitness.

Programming this project proved to be quite a challenge. While the initial coding was simple, it became much more complex as testing began. There were a few very large hurdles to get over. The first was just understanding how the network worked. There were several problems with this. The first was it was not very explicit how to begin the spiking of the network if we just placed it in a random area. We decided upon spiking the forward neurons to start it off but it is likely that there is a more accurate method. We have yet to achieve an evolution where when there are no sensor values the robot moves as fast and forward as possible, though we are getting closer. In an attempt to find a solution, we e-mailed one one of the main writers on the paper, Dario Floreano, who told us he used the noise of his ADC to induce random spikes. Amazingly though, our ADC was almost completely noiseless even with 10 bit precision. We settled on sampling noise but there is probably still a better way. Another issue was relating the neurons to the motors. There were many different ways to do this and it took a lot of additional research into papers to find the forward/backward spike method. Sensor updating was another problem lot of additional research into papers to find the forward/backward spike method. Sensor updating was another problem, when and how to trigger the neurons was a huge issue, especially with the inconsistent readings of the IR sensors.

Another issue was the randomly generated number. It took us a very long time to figure out how to get a number that appeared good enough, and even at this point we can't be sure it isn't repeating as there are thousands of randomly generated numbers per second, and not nearly enough seeds. It is possible our network is going in mini loops because of the same numbers being randomly generated again and again. This is a huge problem as neural networks are so reliant on randomness.

One large issue was debugging. It is very hard to figure out if your network is working, especially before the robot is even built. We used printf statements throughout the entire program to see what values our variables were taking. Initially, we couldn't gain any fitness and it took a lot of debugging to begin improving. We would examine neuron and sensor to see what was spiking, we would count the number of spikes to try and see if some spikes just weren't coming. We even used the timing of the motors to try and find out how to make the robot move smoother and quicker without raising velocity limits or permanently setting the step time. At times we had to induce spikes to get things going to see how neurons would react and if internal spikes could be kept going. It is still a bit hard to understand how a few spikes can turn into many. When actually examining the robot, it is still hard to tell the difference between success and luck. Part of this is because the robot keeps switching individuals so it is hard to tell if you have a highly evolved individual or a weak one. Sometimes, the robot will stare at the wall moving closer and then moving back, slightly to the right or left, for a long period of time without hitting or gain any progress in a different direction. In other cases, the robot would suddenly 180 and start going the opposite direction. This is the hardest part about neural networks, they are unpredictable and each one could be completely different from the next.

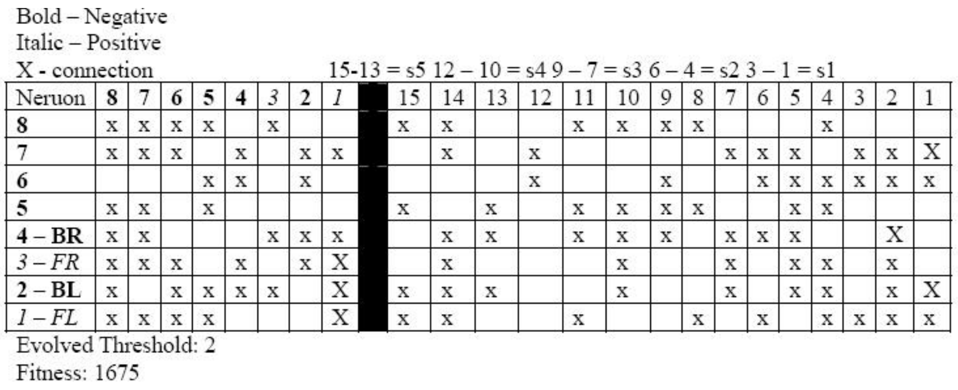
Overall, the coding has really come together at the end and from tweaking we have been able to achieve higher fitness values at a faster rate. Through good commenting and debugging, the code is very clear and it is easy for anyone to begin debugging simply by searching for debug. Tasks help organize code and a sample main and ISR make everything more modular and easier to handle one by one.



## Results

Our results ended up being pretty good. Though there was a little luck with what your starting population would be, the robot would usually begin moving quickly. Because of the stepper motors task times constantly being updated, the movement appears jerky though it is actually very exact. The speed of learning is a little longer than we expected, but it is clear there is learning going on. The design was very safe with the arena and foam board. Anyone can use it, just power it up and it begins learning. There is no interference as far as we know, though the temperature of the room may be effecting our IR sensors.

We were able to use eeprom to get results for one of our best individuals so far. Below is that individual's network. It is interesting to note that the Forward Right neuron and FL neuron's only exciter is the FL neuron, and that the majority of neurons are inhibitory. With so much inhibiting, it is hard to get going fast but really easy to avoid walls. This was reflected in the robots movement, it was very inhibitory but avoided walls well. Sensor networks are a little more unclear but the robot spent a long time with its right side towards a wall so it makes sense that there are more neurons connected in that area.

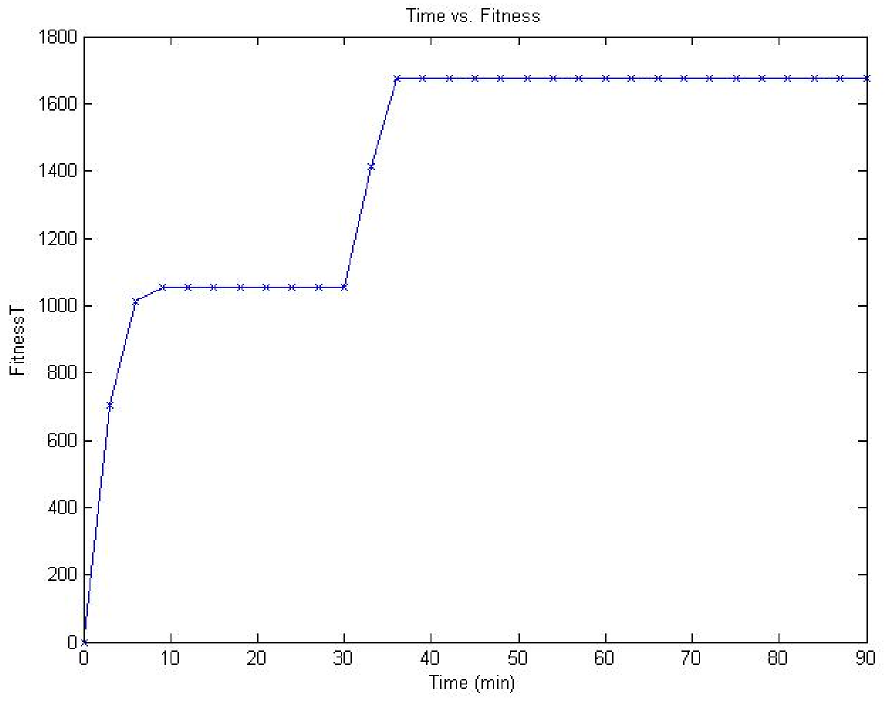


Bold – Negative
Italic – Positive
X - connection 15-13 = s5 12 – 10 = s4 9 – 7 = s3 6 – 4 = s2 3 – 1 = s1

| Neruon | **8** | 7 | **6** | **5** | **4** | *3* | **2** | *1* | | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **8** | x | x | x | x | | x | | | | x | x | | | x | x | x | x | | | | x | | | |
| 7 | x | x | x | | x | | x | x | | | x | | x | | | | | x | x | x | | x | x | X |
| **6** | | | | x | x | | x | | | | | | x | | | x | | | x | x | x | x | x | x |
| **5** | x | x | | x | | | | | | x | | x | | x | x | x | x | | | x | x | | | |
| **4 – BR** | x | x | | | | x | x | x | | | x | x | | x | x | x | | x | x | x | | | X | |
| *3 – FR* | x | x | x | | x | | x | X | | | x | | | | x | | | x | | x | x | | x | |
| **2 – BL** | x | | x | x | x | x | | X | | x | x | x | | | x | | | x | | x | x | | x | X |
| *1 – FL* | x | x | x | x | | | | X | | x | x | | | x | | | x | | x | | x | x | x | x |

Evolved Threshold: 2
Fitness: 1675

Here is the fitness curve for that population:



## Conclusion

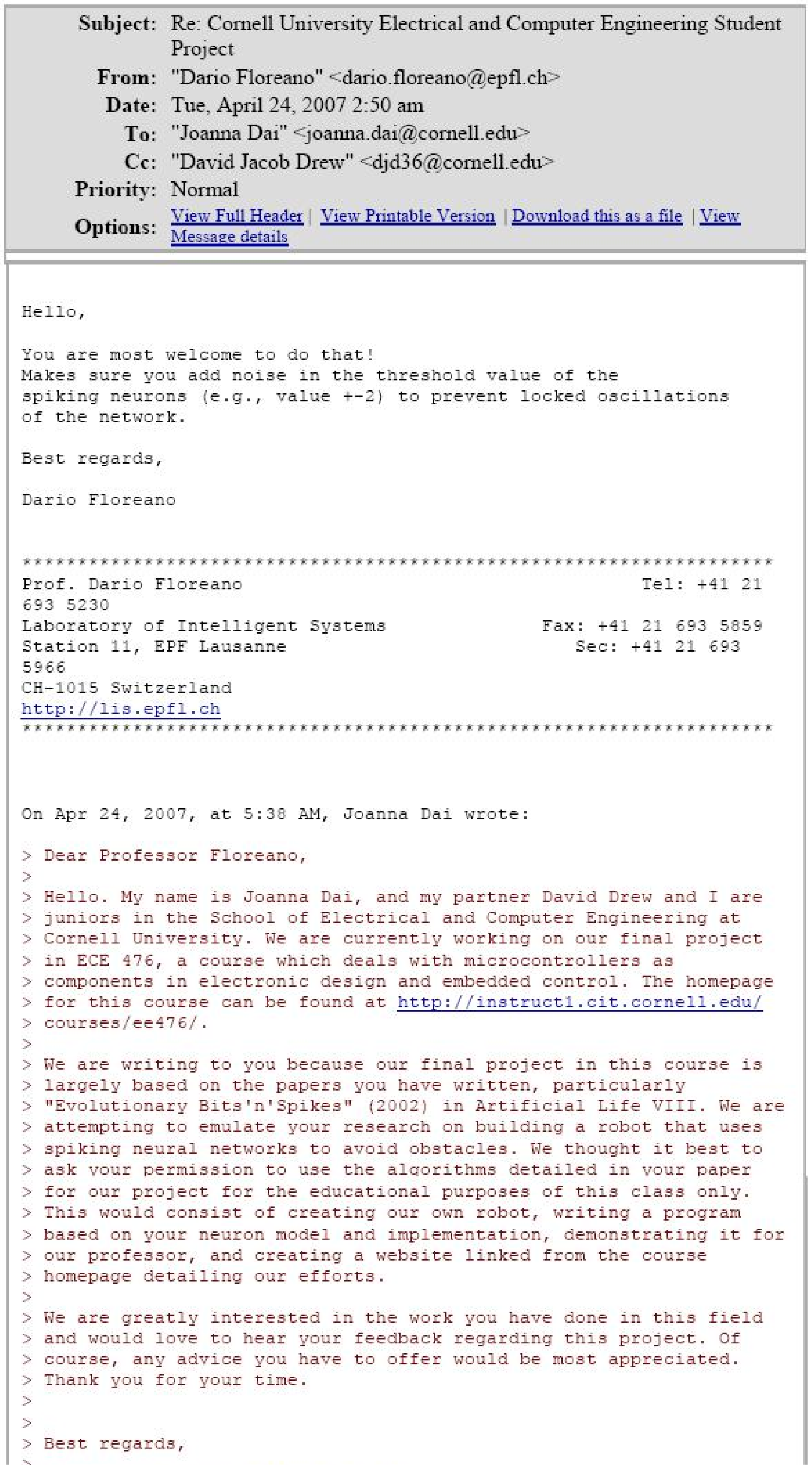
We are very happy with the way our robot came out. We found the topic very interesting but challenging. Understanding a neural network is not a simple task, but we both now feel we have a much better idea of how machines can learn and evolve. Overall we are very proud of how far our robot has come. We do feel there may be a better way to cause forward motor spikes to continue so that it can move a long distance over time, and with some tweaking we could probably achieve that. Nonetheless, we are happy to say our robot is completely unpredictable, but is clearly is abiding by its fitness. There are times where you can really see it hesitate near a wall and move away from hitting it. Learning has taken a little longer than we expected, probably due to the fact that the paper we modeled this off of used less bits and ran the network slightly faster. We wish we could have had more time just to let it go and see where the network went. We are sure there are network genetic strings we haven't even come close to that can achieve fantastic performance. We also would have liked to spend some more time tweaking, and could we do it again, we would have finished building the hardware much faster since testing on the STK500 was not at all comparable to testing with an actual robot. There are so many small tweaks to the network that can have a large impact on learning; it would be great to have more time to test it all. Another problem was the IR sensors; we wish we could have found a way to make them more accurate and consistent. We also think it would have been very cool to take things a step further and have two robots with the same algorithm, so they can avoid each other and the wall.

We believe our design conformed to standards. We used all the hardware within the appropriate voltage ranges, and used the RS-232 protocol properly. We also attempted to follow, within the best of our ability, the standard of a genetic algorithm and how it is meant to be run.

## Intellectual Property Considerations

In terms of intellectual property, we used the implementation of the neural network described in the paper 'Evolutionary Bits'n'Spikes'. While we did modify the implementation to suit our purposes, as well as add a few special touches such as evolving the threshold, it really helped guide us along the way. To make sure there were no issues (and ask for advice), we contacted the writer of the paper we were using, Dario Floreano.

He had this to say:



|  |  |
|---|---|
| **Subject:** | Re: Cornell University Electrical and Computer Engineering Student Project |
| **From:** | "Dario Floreano" <dario.floreano@epfl.ch> |
| **Date:** | Tue, April 24, 2007 2:50 am |
| **To:** | "Joanna Dai" <joanna.dai@cornell.edu> |
| **Cc:** | "David Jacob Drew" <djd36@cornell.edu> |
| **Priority:** | Normal |
| **Options:** | View Full Header \| View Printable Version \| Download this as a file \| View Message details |

```
Hello,

You are most welcome to do that!
Makes sure you add noise in the threshold value of the
spiking neurons (e.g., value +-2) to prevent locked oscillations
of the network.

Best regards,

Dario Floreano


*********************************************************************
Prof. Dario Floreano                                      Tel: +41 21
693 5230
Laboratory of Intelligent Systems                 Fax: +41 21 693 5859
Station 11, EPF Lausanne                             Sec: +41 21 693
5966
CH-1015 Switzerland
http://lis.epfl.ch
*********************************************************************



On Apr 24, 2007, at 5:38 AM, Joanna Dai wrote:

> Dear Professor Floreano,
>
> Hello. My name is Joanna Dai, and my partner David Drew and I are
> juniors in the School of Electrical and Computer Engineering at
> Cornell University. We are currently working on our final project
> in ECE 476, a course which deals with microcontrollers as
> components in electronic design and embedded control. The homepage
> for this course can be found at http://instruct1.cit.cornell.edu/
> courses/ee476/.
>
> We are writing to you because our final project in this course is
> largely based on the papers you have written, particularly
> "Evolutionary Bits'n'Spikes" (2002) in Artificial Life VIII. We are
> attempting to emulate your research on building a robot that uses
> spiking neural networks to avoid obstacles. We thought it best to
> ask your permission to use the algorithms detailed in your paper
> for our project for the educational purposes of this class only.
> This would consist of creating our own robot, writing a program
> based on your neuron model and implementation, demonstrating it for
> our professor, and creating a website linked from the course
> homepage detailing our efforts.
>
> We are greatly interested in the work you have done in this field
> and would love to hear your feedback regarding this project. Of
> course, any advice you have to offer would be most appreciated.
> Thank you for your time.
>
>
> Best regards,
>
```

```
> Joanna Dai joanna.dai@cornell.edu
> David Drew djd36@cornell.edu
```

We are happy to say we are locked oscillation free.

## Ethical Considerations

We believe we were consistent with the IEEE code of ethics to the best of our ability. We made sure to contact the creator of the algorithms we used to avoid any damage. We did not attempt to fabricate data or fitness values of any kind to make our project look more effective. We did a large amount of research before this project to ensure it we had the understanding to complete it safely. To create a safe environment around our robot, we built it with a soft, circular design that ensures that no collision related accidents will occur. The arena further protects this since the robot will not be able to pass the walls (though one would hope it would avoid them entirely). We did not attempt to anything beyond the level we could safely handle, and contacted TAs or Bruce if we had a situation we were unsure about. We strongly desired the advice and criticism that Professor Floreano, Bruce, and the TAs had to offer, and without them we would not have been able to complete our project as well as we did. And of course, whenever we had the moment, we took the time to help others around us who may have been struggling with their project. At no point did we accept any bribes, or attempt to damage any other projects for our own personal gain.

## Legal Considerations

There were no legal considerations in terms of the hardware for this project. For the software, we received the written permission of the designer of the algorithm we used.

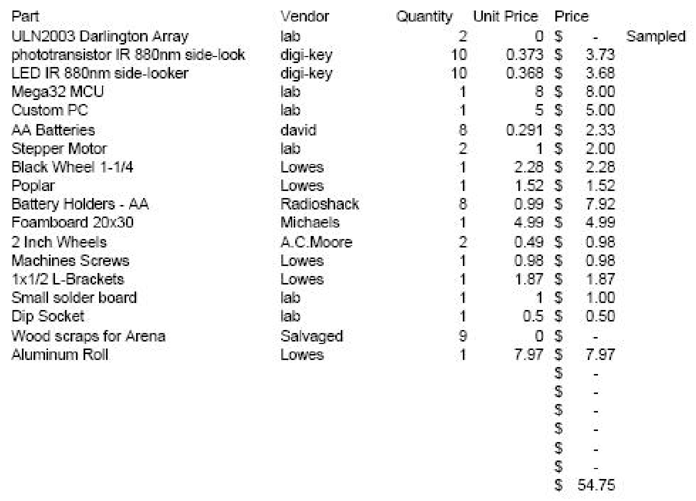
## Appendix

Link to fully commented code: robot.c

Link to stepper motor schematics

Link to infrared sensor schematics

## Project Costs



| Part | Vendor | Quantity | Unit Price | Price | |
|---|---|---|---|---|---|
| ULN2003 Darlington Array | lab | 2 | 0 | $ - | Sampled |
| phototransistor IR 880nm side-look | digi-key | 10 | 0.373 | $ 3.73 | |
| LED IR 880nm side-looker | digi-key | 10 | 0.368 | $ 3.68 | |
| Mega32 MCU | lab | 1 | 8 | $ 8.00 | |
| Custom PC | lab | 1 | 5 | $ 5.00 | |
| AA Batteries | david | 8 | 0.291 | $ 2.33 | |
| Stepper Motor | lab | 2 | 1 | $ 2.00 | |
| Black Wheel 1-1/4 | Lowes | 1 | 2.28 | $ 2.28 | |
| Poplar | Lowes | 1 | 1.52 | $ 1.52 | |
| Battery Holders - AA | Radioshack | 8 | 0.99 | $ 7.92 | |
| Foamboard 20x30 | Michaels | 1 | 4.99 | $ 4.99 | |
| 2 Inch Wheels | A.C.Moore | 2 | 0.49 | $ 0.98 | |
| Machines Screws | Lowes | 1 | 0.98 | $ 0.98 | |
| 1x1/2 L-Brackets | Lowes | 1 | 1.87 | $ 1.87 | |
| Small solder board | lab | 1 | 1 | $ 1.00 | |
| Dip Socket | lab | 1 | 0.5 | $ 0.50 | |
| Wood scraps for Arena | Salvaged | 9 | 0 | $ - | |
| Aluminum Roll | Lowes | 1 | 7.97 | $ 7.97 | |
| | | | | $ - | |
| | | | | $ - | |
| | | | | $ - | |
| | | | | $ - | |
| | | | | $ - | |
| | | | | $ - | |
| | | | | $ 54.75 | |

## Tasks

Researching - David and Joanna
Buying Parts - David and Joanna
Hardware - David and Joanna
Software - David
Website - David and Joanna
Testing - David and Joanna
Mechanical - Joanna
Documentation - David and Joanna

## References

Datasheet for Infrared Phototransistor
Datasheet for Infrared Light Emitting Diode
Datasheet for ULN2003AN Darlington Array

Vendor websites: Digikey

Link to borrowed design: Khepera

Links to published papers: Evolutionary Bits'n'Spikes, Evolution of Spiking Neural Circuits in Autonomous Mobile Robots

## Acknowledgements

Shout outs/Acknowledgements to the following people: Bruce Land for creating an awesome course. Dario Floreano for making such strides in evolutionary robotics and for helping us. All the TAs who advised all of our problems, big and small. Jack from Lowes for having the vision and a very nice saw.