Engineering Design: Design and Control of A Simple Robot

By Roby Velez

Advisor: Erik Cheever

Abstract: This paper details my senior design project. It goes through the process of the creation of a simple, crawling robot. It describes different control algorithms used to make the robot crawl. One of the control algorithms looked at was learning with artificial neural networks (ANN) and genetic algorithms (GA). The ANN and GA produced controllers which relied on coupling between sensors and servos and oscillators to produce robust and fast crawling.

1. Introduction

Most mobile robots are wheeled. They are easy to make and easy to control. The only issue is that these wheeled robots live in a world inhabited by walking organisms. Legged robots are more difficult to build because of limitations with actuators and power demands. They are also difficult to program because they require the coordination of multiple linkages. But legged robots offer immense possibilities such as searching through a collapsed building with rubble for victims, transporting equipment over harsh terrain for the military, or working in a home environment assisting people with disabilities.

One issue in robotics is trying to make robots smart enough to adapt and learn in changing environments as well as find solution to problems on their own. This has caused many roboticist to draw inspiration from biological systems which can fend for themselves in sometimes very harsh and changing environments. Two computation models/algorithms inspired by biology are artificial neural networks and genetic algorithms. Artificial neural networks are modeled after real neural networks. They consist of a collection of interconnected nodes where information propagates through the connections, is altered, and then shoots out the end. Genetic algorithms are a way of using evolution to search through a solution space for a solution to a problem. Artificial neural networks and genetic algorithms are both biologically inspired and incorporate elements of adaption and learning.

Researchers in Switzerland have been looking at robots which move not by rolling on wheels, but by flapping their fins(2), crawling on four legs(3), or slithering like a snake(1). The researchers have been able to accomplish this through the use of Central Pattern Generators. Central Pattern Generators are modeled off of a type of neural network. These neural networks produce oscillatory output given none oscillatory input. Central Pattern Generators and oscillatory motion has been found to be essential to locomotion. While this work is very interesting the researches used models of neural networks, not actual artificial neural networks to control the robots.

The purpose of this project is to try to emulate the researchers from Switzerland and use not a model of a neural network, but an actual artificial neural network to control the locomotion of a robot. Using actual artificial neural networks prevents constraining to the artificial neural networks to a particular method for solving the problem. The artificial neural networks could find oscillators to solve the locomotion problem or come up with some novel approach. For the most part there hasn't been much work on using actual artificial neural networks to control locomotion which is time, and memory dependent problem which doesn't have instant feedback.

Artificial neural networks are generally used to learn through a training algorithm which alters the weights to get the neural network to produce a certain output for a given input. However artificial neural networks can be used to solve a problem by evolving their weights and topology with a genetic algorithm. This can be done with an algorithm known as NeuroEvolution of Augmenting Topologies (NEAT) (4). This is what will be used in this project to search for artificial neural networks to solve this task.

The evolution of an artificial neural networks which can control a crawling robot is the pinnacle of this project, but to get there required many steps or stages which include design and construction of the micro controller and robot chassis, creation of the low level servo and command protocols, creation of the higher level control algorithms which includes the NEAT and artificial neural networks, and finally the analysis of the evolved neural networks. These will be laid out in the follow sections.

2. Hardware

The first stage of this project consisted of the creation of the actual physical robot and a protocol to program and do low level control. This consisted of the creation micro-controller, robot chassis, and command protocol.

2.1 PCB

A printed circuit board (PCB) was used to control the robot. A block diagram for the PCB is shown below.

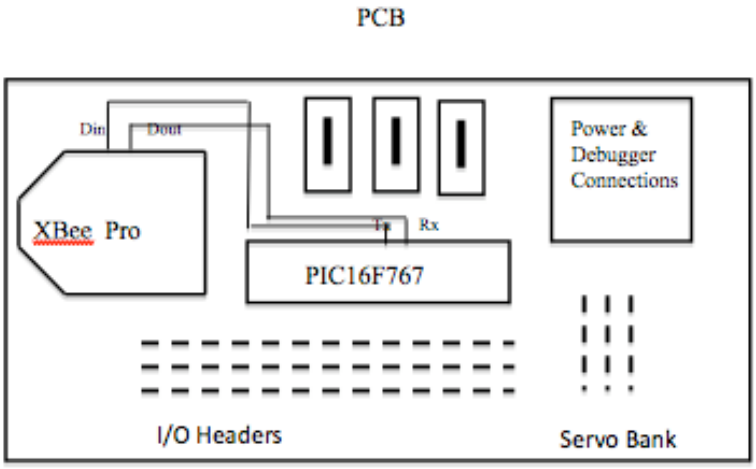

Figure 2.1: Block Diagram of Printed Circuit Board (PCB).

At the center of the PCB is a microcontroller. The microcontroller can be looked as a miniature-processing unit. It can be programmed to do calculations, read sensors and actuate servos.
The microcontroller used was the PIC16F767. The PIC16F767 is a simple, inexpensive microcontroller chosen because it had the most PWM compare pins, which would be used to control servos, and had a lot of digital and analog inputs.

Like a central processing unit in a real computer a PIC cannot work on its own. It needs peripherals such as power, debugger, and I/O connections in order to function. The PCB had to provide 5 volts for the PIC, 7 volts to power the servos, and 3 volts for the XBEE module. The XBEE module is an integrated chip that will provide wireless communication to the PCB. It will be discussed shortly. The circuit schematic in Figure 2.2 shows how 7 volts were brought into the PCB and drop down to 5 volts and 3 volts by voltage dividers.

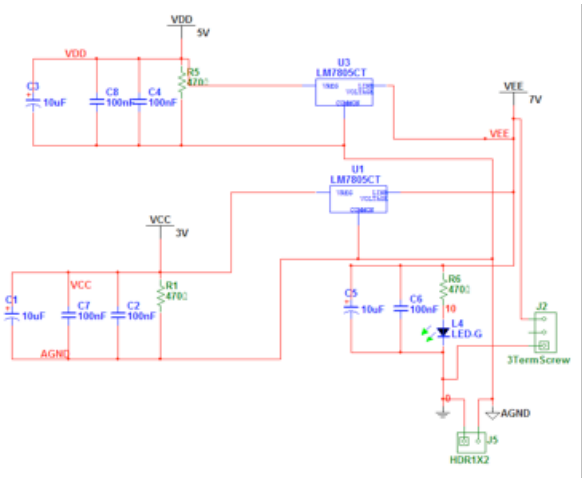

Figure 2.2: Power schematic.

Besides providing power the PCB also provided other peripherals such as a debugger (which is used to program the PIC) and I/O connections. Figure 2.2 shows the Multisim schematic for the rest of the PCB.

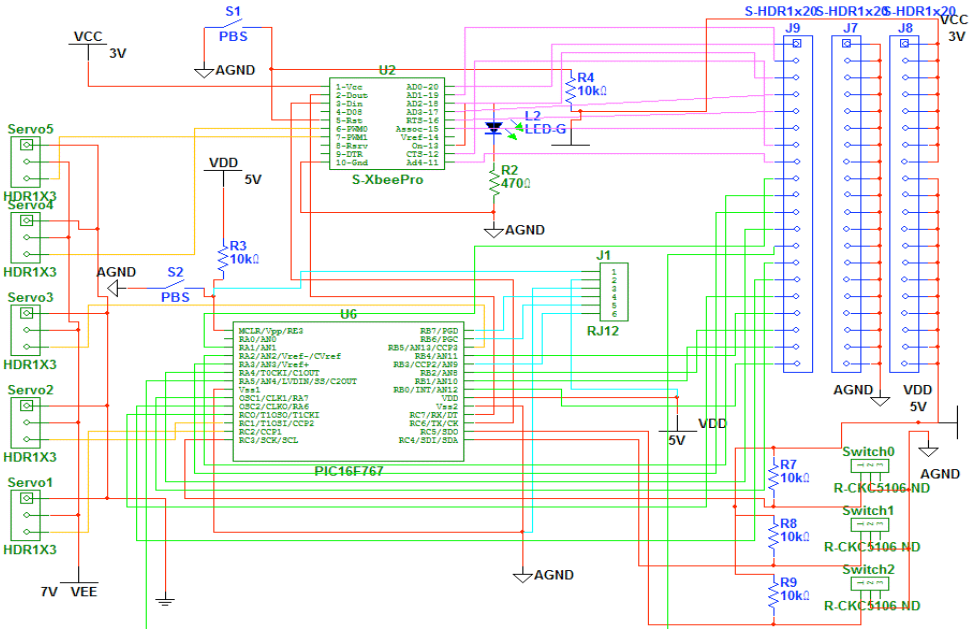

Figure 2.3: Circuit schematic.

Most of the I/O pins, not used for power, debugging, or serial communication between the PIC and XBEE, are routed to the bank of headers located in the upper right of Figure 2.3. The header banks provide access to the I/O pins for the PIC and XBEE as well as access to the ground, the 5 volt power supply, and 3 volt power supply. Designing the headers into the PCB like this enables a lot of flexibility for adding sensors.

In figure 2.3, to the left of the PIC, a bank of headers were made for the PWM pins of the PIC and XBEE. This is where the servos would be connected. To right of the PIC is the RJ12 used to connect and program the PIC. A debugger like this is standard for most microcontroller boards. Finally the PIC is connected to the XBEE by wiring the TX pin of the PIC to the dataIn pin of the XBEE and by wiring the RX pin of the PIC to the dataOut pin of the XBEE. These pins are used to transmit serial data between the XBEE and PIC. Figure 2.4 shows the completed PCB with all the components.

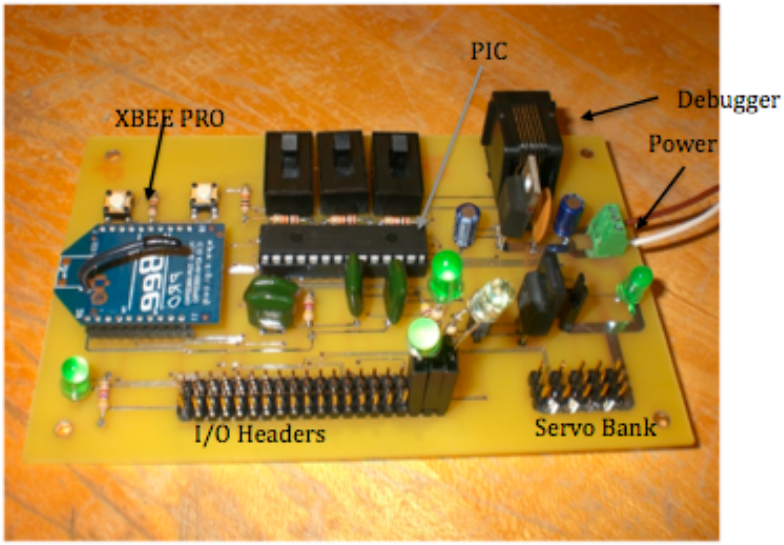

Figure 2.4: Completed PCB.

XBEE

The XBEE modules can form very simple, low power networks that transmit serial data. They act like wireless cables. Wireless communication with a mobile robot is very useful in debugging and creating control architecture fairly quickly. Live commands can be sent to the robot and executed instantaneously versus writing code to the PIC, compiling it, and then running it.

One XBEE module, called the Controller XBEE, was connected to the PIC and another called the Base XBEE, was connected to a remote computer running MATLAB. This is seen in figure 2.5. The Controller is a XBEE PRO while the Base is a regular XBEE. For the most part the two modules are the same except that the XBEE PRO is 5-volt tolerant. This was essential since it was receiving serial data from the PIC which was running off of 5 volts.

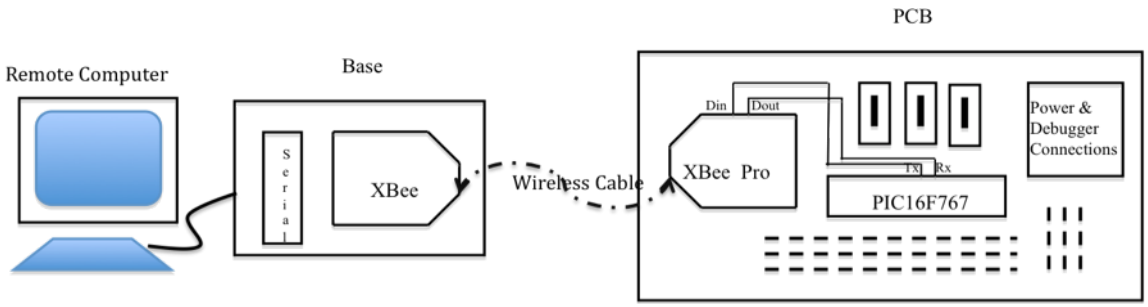

Figure 2.5: Diagram showing the wireless communication between the remote computer and PCB.

The serial port, on the remote computer, is opened by MATLAB and serial data, such as strings, can be sent through the port to the Base XBEE module. The Base XBEE module then transmits the serial data over a wireless cable to the Controller XBEE module on the PCB. The Controller XBEE module then transmits the serial data into the PIC where the PIC can read it. The PIC can also transmit serial data to the Controller XBEE where it is sent to the Base XBEE. The Base XBEE then sends the serial data to the computer where it is read with MATLAB

2.2 Chassis

Since the purpose of this project was to explore locomotion with a legged robot a very simpl, yet interesting design was created. The robot is composed of a 3 degree of freedom (DOF) arm mounted onto a wooden chassis. The chassis rests on two contact sensors in the front and has to two castor-like wheels in the back which are odometers. Figure 2.6 shows a concept drawing of the robot.

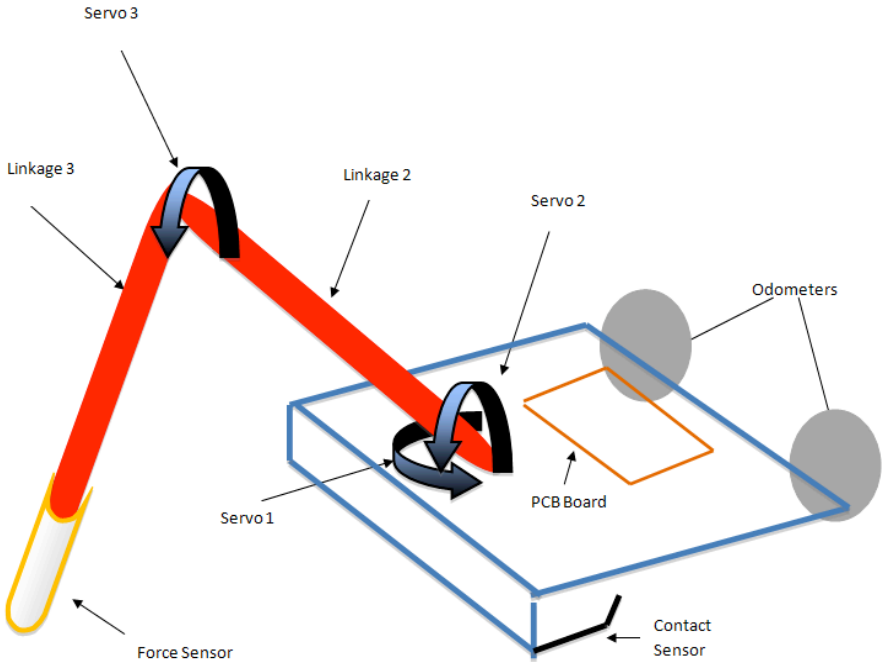

Figure 2.6: Concept drawing of robot.

A crawling robot was designed instead of a traditionally wheeled robot because a crawling robot would be more interesting to control with the learning algorithm instead of a wheeled robot. The design is simple in that there is only one limb and things like balance aren't issues as the robot moves. But the design still requires the coordination of multiple joints to produce locomotion so it retains the complexity of locomotion and manipulator control.

Linkage 3 measures 13 centimeters with the end affecter, and linkage 2 measure 11 centimeters. The end affecter is a half inch piece of pixel glass made into a disc and bolted onto the end of linkage 3.The plywood board, that the arm is mounted onto, is a piece of quarter inch plywood 16 by 18 centimeters. At the front, underneath the plywood is a foam block with the dimension 13 by 3 by 3. Two contact sensors are attached to either sides of the foam block. The robot rests on the contact sensors not the foam block. The foam block is simply there to allow the contact sensors to be attached to the chassis. Finally as the end of the chassis are two continuous potentiometers with wheels mounted onto their shafts. When the robot rears up and lifts the chassis off the ground it rolls on these two wheels. Figure 2.7 shows an actual image of the robot.

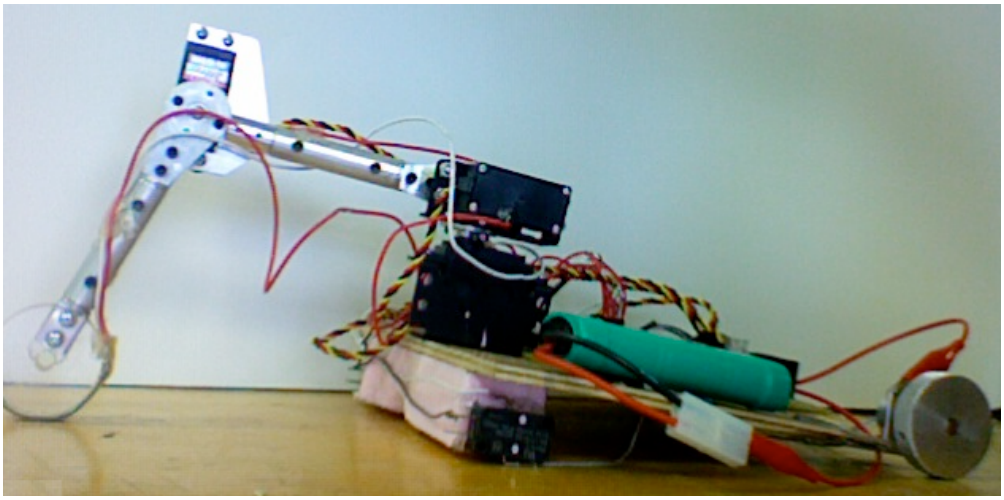

Figure 2.7: Image of robot chassis.

The robot has three servos. Servo one rotates the arm with respect to base. Figure 2.8 is a diagram from above the limits of servo 1's travel.

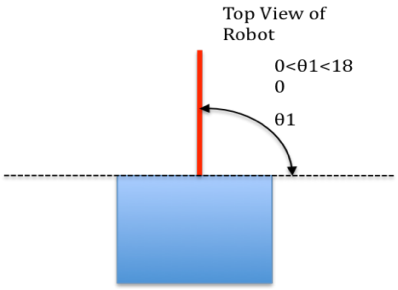

Figure 2.8: Top view of the robot meant to show the range of motion for servo 1.

Servo 2 connects linkage 2 to the servo 1. Servo 3 connects linkage 2 to linkage 3. Figure 2.9 shows the range of motion for servo 2 and for servo 3.

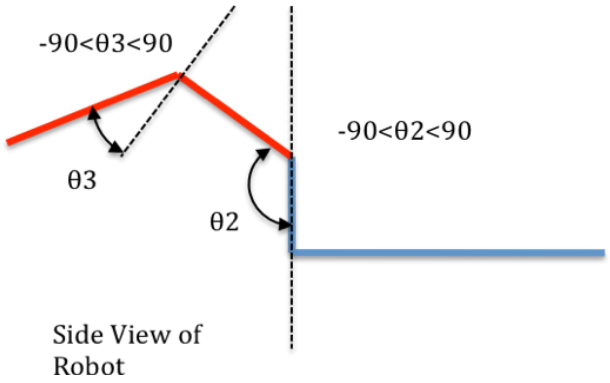

Figure 2.9: Side view of the robot meant to show the range of motion for Servo 2 and 3.

In order to have linkage 2 and 3 in almost the same plane Servo 3 is flipped from Servo 2. Figure 2.10 shows how the Servos are currently mounted(left) with the linkages in the same plane, and how the linkages look if Servo 3 was in the same orientation as Servo 2 (right).

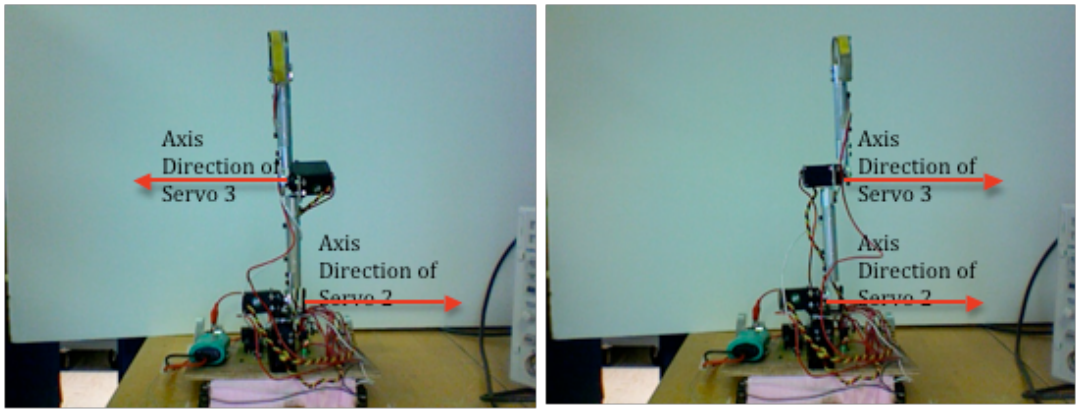

Figure 2.10: Comparison of the arm in a configuration where it is aligned and when it is not aligned.

Image on the left shows the actual configuration of Servo 3 and Servo 2 which brings the arm into alignment, but causes Servo 3 and Servo 2 to face different direction. The image on the right shows a possible configuration of Servo 3 and Servo 2 which causes them to point in the same direction, but the arm is not aligned. This means that if Servo 2 is given a command to rotate clockwise, that same command will make Servo 3 rotate counter clockwise. This is not a serious problem, but it comes into play during the NEAT evolution tests, discussed later.

The robot has six sensors. They are listed in the table below.

Sensor Name	Description
3 Position Sensors	Made from monitoring a pin off of the Servo control board. They indicate the angle of each Servo, but are quite noisy. They range from a value of 80 to 400.
1 Force Sensor	Mounted at the end of the arm. The Servos are not very strong so the sensor was calibrated to be very sensitive, which results it in not being able to distinguish a wide range of different force. The robot pretty much knows if it is touching the ground or not. It is either 1023 when the arm is off the ground and around 600 when the arm is one the ground.
1 Contact Sensor	Two contact sensors are mounted on either side of the foam on the bottom of the chassis. The robot rests of them. They are wired together and act as one contact sensor. They report a 1 if one of them is released and a zero otherwise.
1 Odometer	When the robot rolls forward it is rolling forward on two wheels mounted at the back of the chassis. The two wheels are made up of continuous potentiometers which read a changing voltage as they turn. Only one Odometer reading is read. This is used to measure how far the robot has moved. It reads a value from 0 to 1023. If the potentiometers wiper reaches 1023 but continues to roll it will roll over to 0 and increment normally. If the potentiometer reads 0 and continues to roll backwards it will roll over to 1023 and continues decrementing normally.

Table 2.1: Sensors of the robot.

3. Software

Once the micro controller and chassis were built code was written for the PIC and MATLAB which actuated the Servos and created a command/communication protocol between the remote computer and the PCB. Servos need a specific protocol to work properly and the communication/coding protocol will determine how easy it is to write control algorithms.

3.1 Servos and pulses

Servos work by sending them pulse widths. Pulse widths can be generated by a PIC holding a pin high for a set amount of time and then setting it low. The pulses can be launches at set periods. This is shown in Figure 3.1.

Figure 3.1: Illustration of pulses widths. Altered image taken from. http://www.servocity.com/html/how_do_servos_work_.html

For servos the pulses have to have a period of 20 milliseconds and widths ranging from 500 microseconds to 2500 microseconds. The width of the pulse determines the position of the servo. This is shown in Figure 3.2.

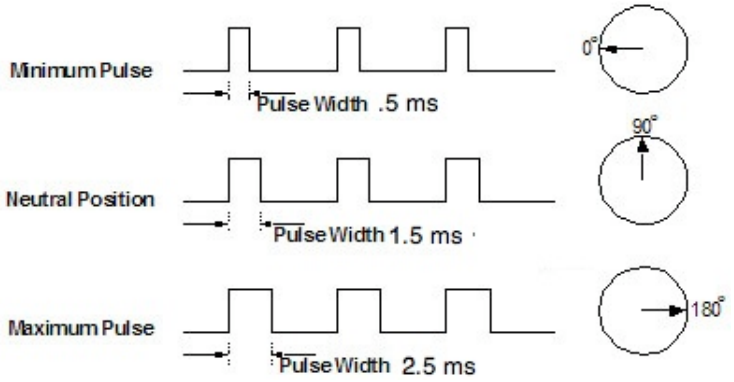

Figure 3.2: Image of how pulse widths correspond to position of the servos. Image altered and taken from http://www.servocity.com/html/how_do_servos_work_.html

3.2 Timer 1 and PWM Compare

In order to produce and vary the pulse widths Timer1 and the PWM compare functions of the PIC are used. An internal clock regulates the operation of the PIC. This internal clock (Fosc) is simply an internal (or for some PICs external) oscillator that oscillates and sends pulses at certain frequencies. Timer1 is a 16 bit counter which increments once every four clock cycles. Because Timer1 is a 16 bit number, once it counts up to 65535 it restarts at 0 and begins counting up again. This is called an overflow and is shown in Figure 3.3.

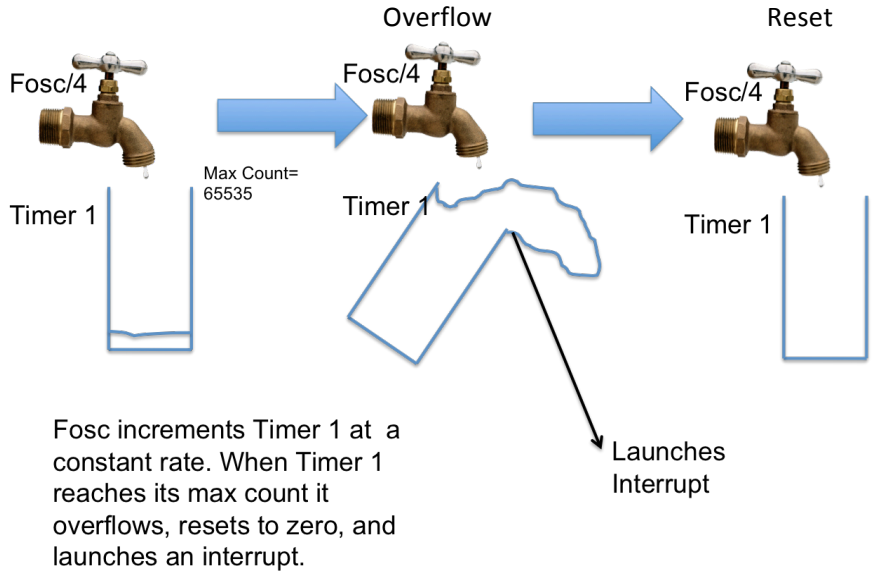

Figure 3.3: Diagram illustrating the filling and overflowing of Timer 1.

The PIC can be made to launch an interrupt whenever this happens. An interrupt is an operation that can be triggered by a number of things such as a register overflowing, a compare event, or a change of an external input. When an interrupt is triggered the PIC puts whatever it is doing on hold and does whatever code is assigned to the interrupt.

The count of Timer 1 can be set to some Initial Count (IC) after it overflows. Timer 1 will then count up from this Initial Count until it reaches 65535 and overflows again. Because Timer 1 increments at a fixed rate the time between overflows can be altered by through the Initial Count value. This is illustrated in Figure 3.4.

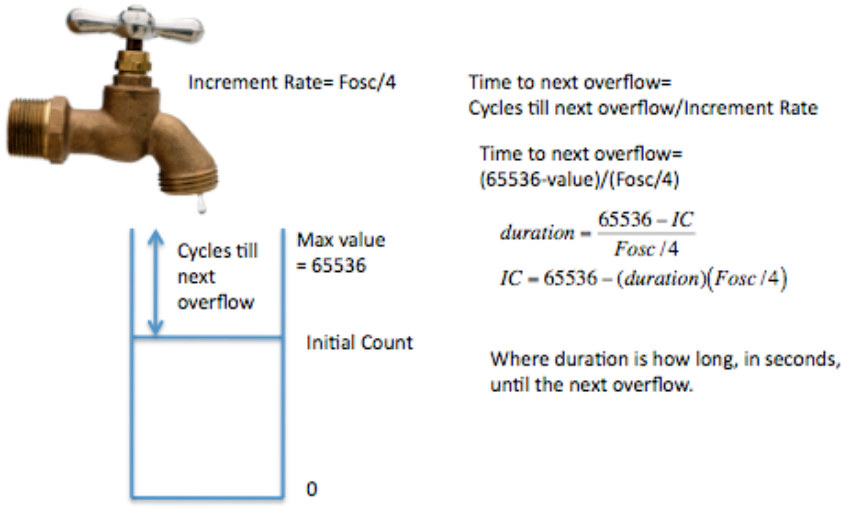

Figure 3.4: Diagram with describes how the Initial Count can be calculated to produce an overflow after a set about of time.

With Fosc equal to 8Mhz and an Initial Count of 60535 Timer 1 will over flow and launch an interrupt every 20ms. The interrupt will pull the pins connected to the Servos high. Servo 3 is connected to pin c1. Servo 2 is connected to pin c2. Servo 1 is connected to pin b3.

That's one part of the servo actuation. The second uses the PIC's PWM compare function. The PIC has three internal variables called CCP_1, CCP_2, and CCP_3. The PIC can be made to launch interrupts whenever one of these values equals the current value of Timer 1. The interrupts will pull the pins connected to the Servos low. This is shown in Figure 3.5.

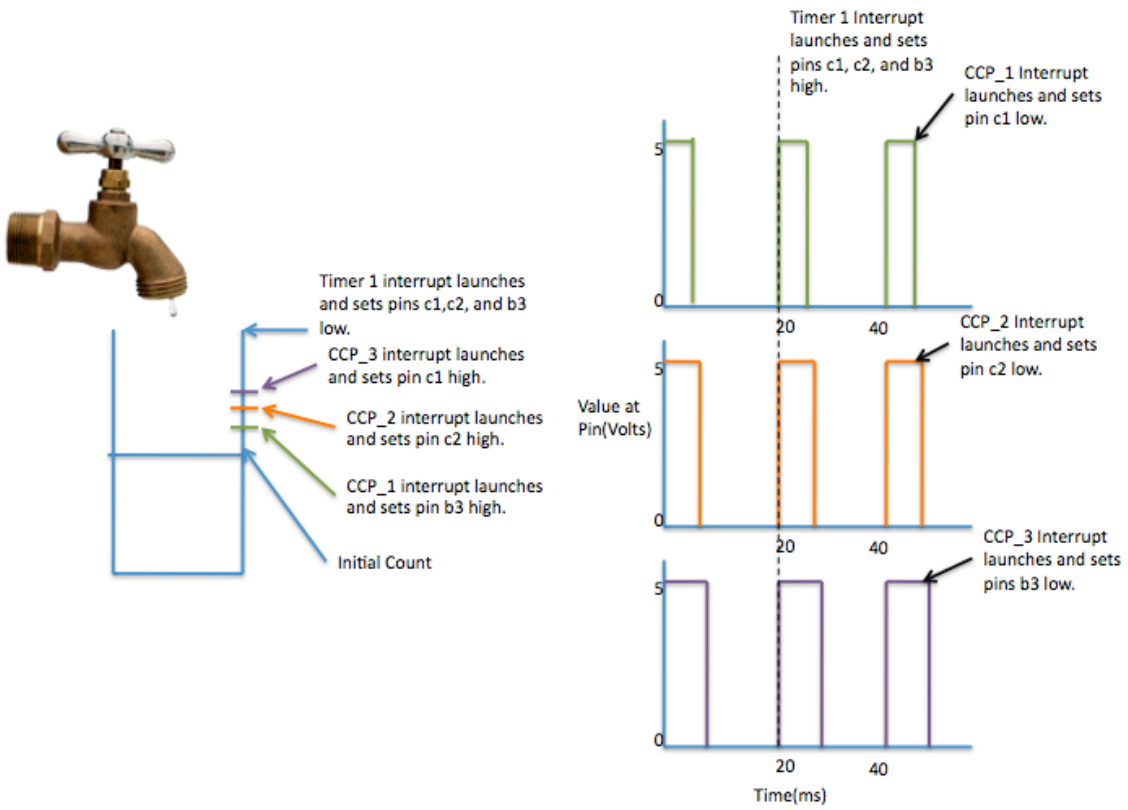

Figure 3.5: Illustration of how the filling of Timer 1 launches the interrupts of CCP_1, CCP_2, and CCP_3.

The width of the pulse can be calculated as:

$$\left(\frac{CCP - InitialCount}{(Fosc/4)}\right) = pulseWidth$$

The pulse width determines the position of the servo. The equation below solves for the CCP value to produce a certain pulse width.

$$CCP = SetPoint + pulseWidth * (Fosc/4)$$

This means the Servos can be controlled by changing the CCP values.

In summary setting Timer 1 to some Initial Count after it overflows determines the time until the next overflow when an interrupt will be launched. The interrupt will happen every 20ms and will pull pins c1, c2, and b3 high. The compare function of the PIC can be used to launch interrupts and pull the pins c1, c2, and b3 low. The time between Timer 1 setting a pin high and a CCP interrupt setting it low corresponds to the CCP value. This will produce a pulse a certain width every 20ms which can be used to control a servo.

3.3 Handshaking between PIC and MATLAB

The purpose of equipping the PCB with a wireless function was to be able to actuate the servos by typing command strings into MATLAB's command line. MATLAB would print the strings to the serial port which would then send the string to the PCB via the wireless serial connection. This is in contrast to writing, compiling, and debugging code right off the PIC. Figure 3.6 shows a flow chart comparing the two programming approaches.

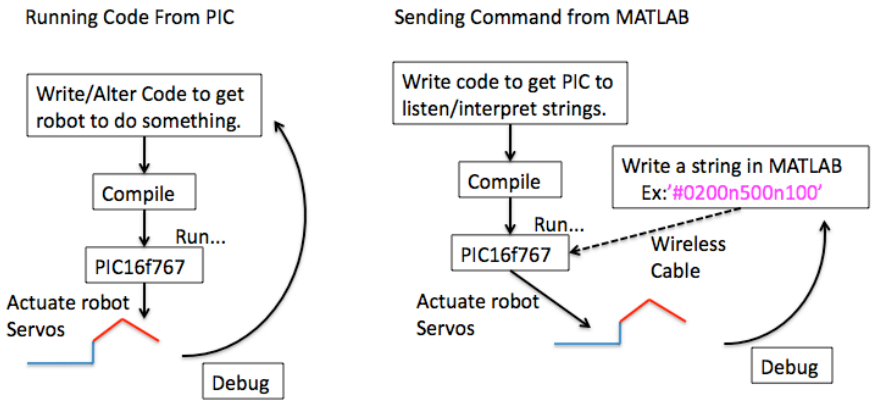

Figure3.6: Flow chart comparing writing code to the PIC versus sending commands over wireless cable.

Sending command from MATLAB clearly looks easier and faster. In order to do this a protocol was developed so that the PIC would understand the command strings sent by MATLAB and MATLAB would understand the command strings sent by the PIC.

MATLAB would send strings to the PIC. The PIC was programmed to look for the '#' symbol to signify the beginning of a command, parse the servo words in blocks of four, and execute the proper action based on the command and servo words. The command strings have the form of:

#	1	0	0	0	2	0	0	0	3	0	0	0	0	0	2	0
Signifies the beginning of a command	First four digits relate to servo1.				Second set of four digits relate to servo 2.				Third set of four digits relate to servo 3.				Teel the robot how to execute the command.			

Table 3.1: Command string protocol.

Currently 3 types of commands have been developed.

Command	Description
0010	Adds the values indicated for each servo to the CCP value assigned to the specific servo sensor. Effectively makes the servo go to the position corresponding to those values.
0020	Print the position reading from potentiometer. Ignore servo words.
0030	Increment the values for CCP_1, CCP_2, and CCP_3 by the values in each servo word. Indicate a negative increment by preceding a servo word with 'n'. For examples '#0200n500n1000030' means increment the first servo by 100 and decrement the second and third servo by 500 and 300 respectively. The largest negative command is n999.
xxxx	Anything not the three commands shown above will simply be ignored.

Table 3.2: Command type options.

The following code opens a serial port and increments Servo 1 by 500 and decrements Servo 2 by 200.

```
>>s=serial('COM1','BaudRate',13400); %Opens the serial port
>>fprintf(s,'# 0500n20000000030');
```

The following code requests the PIC to transmit information on its sensors.

```
>>fprintf(s,'# 0100n20000000020');
>>reponse=fscanf(s,'%s',29);
```

When the PIC receives a '0020' command it queries its sensors and sends them back to MATLAB is the format of:

#	0	2	7	0	0	0	8	0	0	4	0	0	0	9	0	0	0	0	0	1	0	5	4	0	0	0	1	0
Beginning of Command	Position of Servo 1				Position of Servo 2				Position of Servo 3				Force Sensor				Contact Sensor				Odometer Reading				Message from PIC			

Table 3.3: Format for the PIC sending sensor information back to MATLAB.

To streamline the sending and receiving of command and message strings a function called sendCommandString was written. It is defined as:

```
[serialHndl, sensorReadings, message,success]=
        sendCommandString(serialHndl,[servoWord1,servoWord2,servoWord3,commandWord]);
```

serialHndl	Handle for the serial object.
sensorReading	Array with the sensor values. This array is full of zeros unless a 20 command is sent.
Message	Message from the PIC
Success	Flag which indicates if the communication was successful.
servoWord	Instead of typing ‘0020’ or ‘n200’ the values 20 or -200 can be typed in.
commandWord	Instead of type ‘0020’ or ‘0010’ typing 20 or 10 would work.

Table 3.4: Parameters for the sendCommandString function.

The communication between MATLAB and the PIC is over a wireless communication and has some lag. On average the lag between sending say a ‘0020’ and receiving a reply from the PIC was 0.02 seconds. This will come into play in designing the simulator.

3.4 Simulator

A simple 2D simulator was created in MATLAB in anticipation of having to run multiple evolution runs with the robot. A screen shot is shown in figure 3.7.

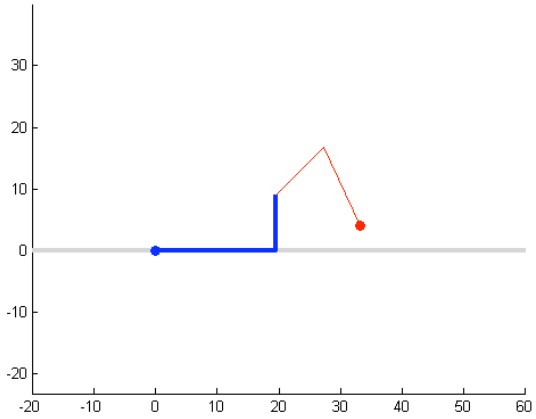

Figure 3.7: Screen shot of the simulator.

Ideally evolution would be done on the actual robot, but this is slow, can cause damage to the robot, and requires someone to continuously watch the robot. The simulator created is fairly simple, but for the amount of time it captures all of the relevant aspects of the robot needed to produce acceptable evolution runs

crawlingRobot()

The cornerstone of the simulator is a simulated robot object called crawlingRobot(). The simulated robot has the same attributes and physical characteristics as the real robot. It has parameters for the current values of all its sensors, current location of its physical body, and values for internal variables present in the real robot such as the CCP values.

Updating Position

MATLAB talks to the simulated robot the same way it talks to the real robot. It uses the sendCommandString function, but passes in a handle to the crawlingRobot instead of the serial port. Once the command strings are sent to the simulated robot it updates its servo locations, physical position, and sensor values. When the command string is deciphered the CCP values of the simulated robot are updated accordingly. The CCP values represent the desired location of the Servos. The simulated robot can not automatically set its servo locations to the angles specified by the CCP values. This would mean that the servos moved instantaneously to where they were told to go.

The servos have a max angular speed. The current location of the servos, which is encoded in a variable called sPos, approaches the desired location of the Servos at a speed equal to the max speed of the servos. The updated location of the servos after each update cycle is calculated from the max angular velocity and the elapse time from the last update. The elapsed time between updates is set to 0.02 seconds. This does not reflect how long the MATLAB code takes to update the simulated robot. 0.02 is the average lag in communication between MATLAB and the real robot. Because the shortest elapse time between MATLAB querying the real robot for sensor values or sending a servo command is 0.02 seconds then this can be used as the theoretical elapse time between update cycles.

Geometry

Once the position of the servos is known the orientation of all the points of the robot can be calculated. This is then used to determine what the sensors values should be, if a collision has occurred, or if the robot has moved any. The simulated robot has two different states which plot and update differently.

The simulated robot rests and crawls over a virtual ground. If the chassis of the simulated robot is on the ground and the end affecter isn't, the joints of the simulated robot are calculated by using the back of the chassis as the reference and figure 3.8. In this reference theta 1, which is the angle between the chassis and the ground, is 0.

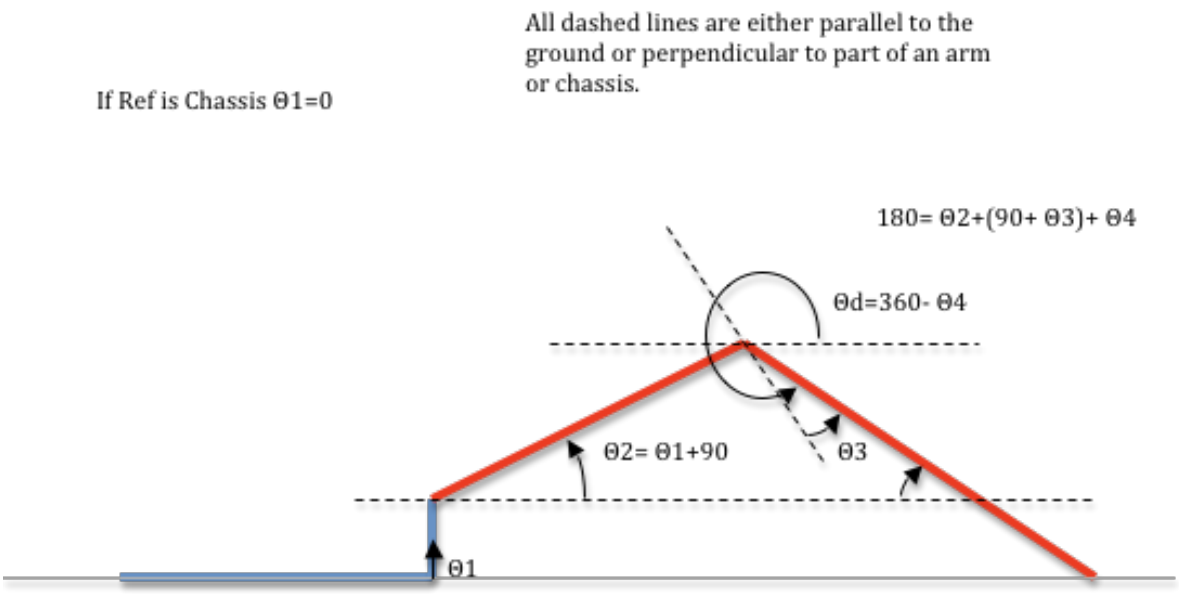

Figure 3.8: Diagram which illustrates the angles used to draw the robot. The reference location is the back of the chassis.

When the end affecter touches the ground it is pinned there and the robot is said to be off the ground. In this state the geometry plots different. In this state the reference point is the end affecter and all the points plot in reference to that. This means that the back point of the robot can move. The simulated robot is out of this state if the chassis touches the ground again. It will then go back to using the back of the robot, whatever location it is at, as the reference location.

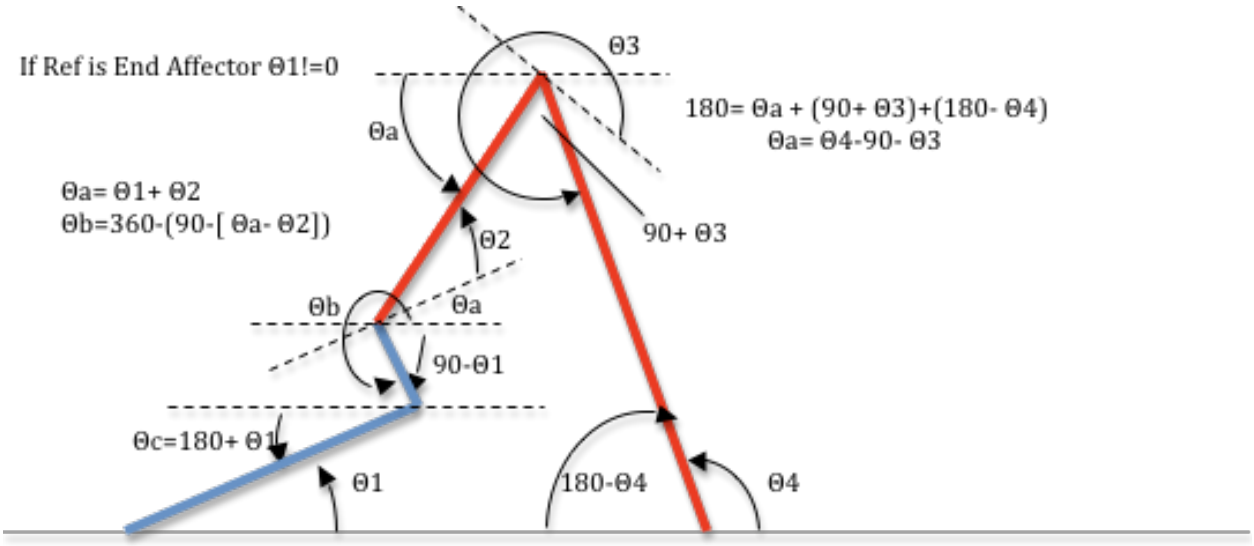

Figure 3.9: Diagram which illustrates the angles used to draw the robot. The reference location is the end affectr.

Simulated Sensors

Once the geometry of the simulated robot is known, after an update cycle is complete, the simulated sensors can be updated. The simulated robot has the same sensors as the real robot which are listed in Table 2.1.

The servo position sensors are calculated after the final position of the arm and chassis are known after the update cycle. Since the real position sensors are noisey, once the sensor value is calculated based on the orientation of the servo Gaussian noise with a standard deviation of 2 is added to the values.

When the simulated chassis is on the ground the contact sensor reads 0. When the simulated chassis is off the ground the contact sensor reads 1. The forces sensors at the end of the arm reads 400 plus a random number ~N(0,100) when it is in contact with the ground or chassis and reads 990 plus a random number ~N(0,30) otherwise. This again reflects the fact that there is some noise with the force sensor reading. When the back of the chassis moves the simulator takes that displacements and calculates how many turns the odometer has made. This is used to figure out what the new odometer reading. The simulated odometer like the real one rolls over from 1023 to 0.

4. Control Architectures

Once the robot was fully completed different control algorithms were explored. While the goal of this project was to explored different learning algorithms two classical control approaches were looked at. Looking at Direct Control and Direct Programming help to debugging some aspects of the robot design and gain an intuition about how hard the problem is.

4.1 Direct Control

Direct Control refers to robots or machines being operated by humans. The operation of construction equipment or underwater ROV's can be seen as the direct control of robots. Direct control is very simple to implement and was used as a test bed to see if the robot could actually crawl.

Three potentiometers were fed into the PIC. The potentiometers can send a continuous voltage into the PIC from 0 to 5 volts. The PIC read this value in as a 10 bit number and changes the pulse width, through the CCP values, for each servo according. 0 volts correspond to a pulse width of 500 microseconds and 5 volts correspond to a pulse width of 2500 microseconds. There was one potentiometer for every servo.

By turning the knobs a human operator can move the arm of the robot to get it to crawl. This was not an easy task and took some time to learn how to coordinate the servos and linkages. Time series images of the robot crawling forward and backward are shown in Figure 4.1 and Figure 4.2.
CCP values accordingly.

Figure 4.1: Time series pictures of the robot being controlled to crawl forward.

Figure 4.2: Time series pictures of the robot being controlled to crawl backwards.

4.2 Direct Programming

Using the command functions created in MATLAB, code was written which queried the sensors of the robot and actuated the servos. The goal was to control the robot and have it crawl forward. A flow chart for how the program worked is shown in Figure 4.3.

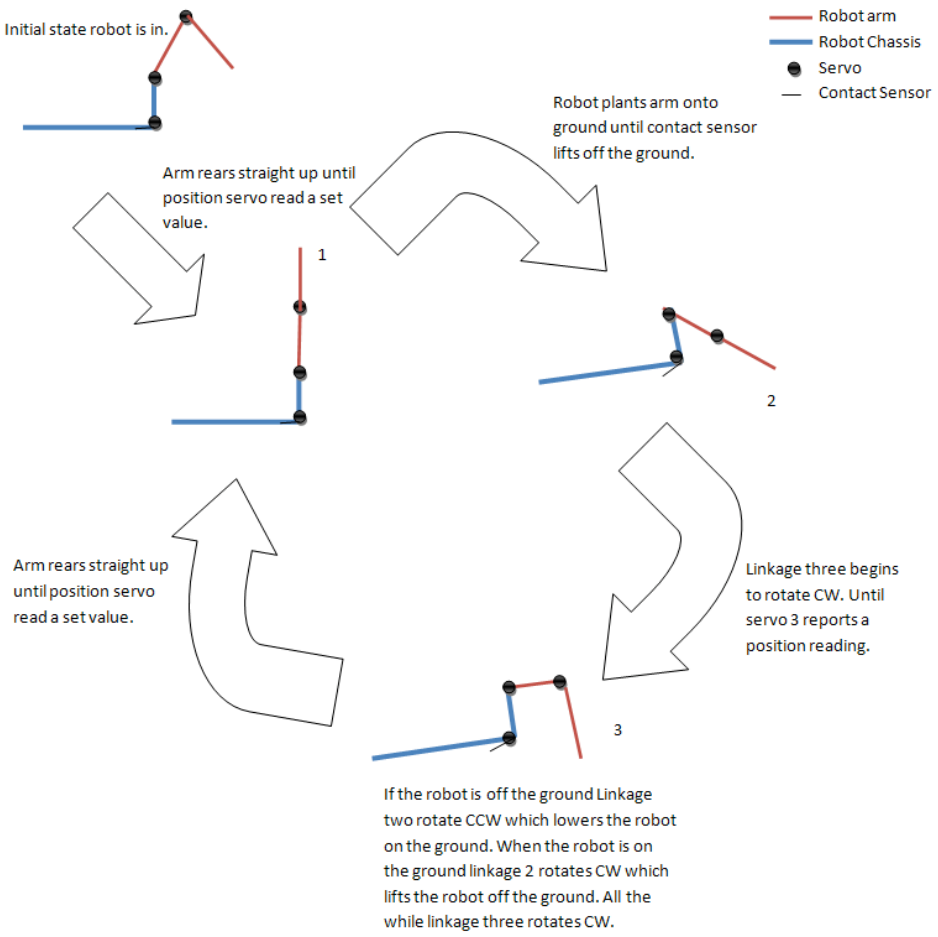

Figure 4.3: Flow chart showing the operation of the direct programming code.

The flow chart shown in Figure 4.3 shows a very determinist algorithm which simply goes from one state to another. The only feedback occurs between states 2 and 3 where linkage 2 will rotate CW or CCW depending if the robot is on the ground or not. Servo 3 and 2 are not well coordinated to one another. They are actually coordinated to the contact sensor. This approach produced a crawling robot, but it crawled fairly slowly.

4.3 Learning

Once Direct Control and Direct Programming were done successfully learning was looked at. In order to explore learning two tools from machine learning were used, artificial neural networks (ANN), and genetic algorithms (GA). The ANN are going to be used to control the robot. They are going to manage the interaction between the inputs and outputs. A GA will be used to tune the weights and topology of the ANN to do a task. For these set of experiments the task is to get the robot to move some distance, either forwards or backwards.

ANN

Figure 4.3 shows a ANN which solves AND. The capabilities of this ANN depend on the weights and topology. In order to produce a crawling robot the right weights and topology needs to be found.

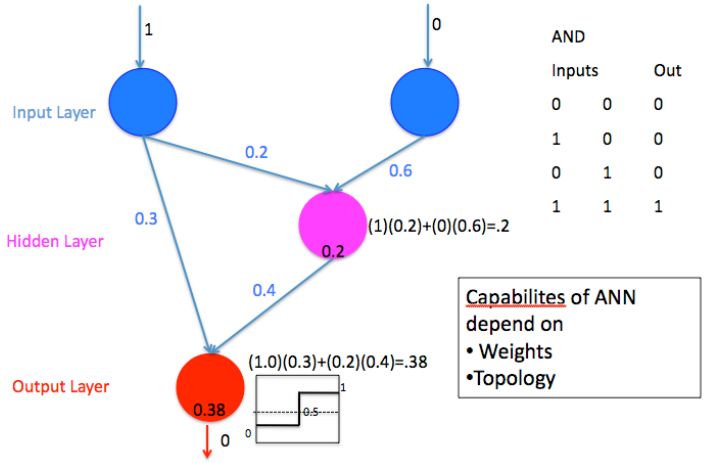

Figure 4.3: Example of a artificial neural network.

Genetic Algorithms

Genetic algorithms are type of search algorithm inspired by biology. They are based on the idea of evolution. A genetic algorithm will be used to find the right weights and topology for the ANN.For a given problem, such the one presented in Figure 4.4, the solutions are broken into genotypes.

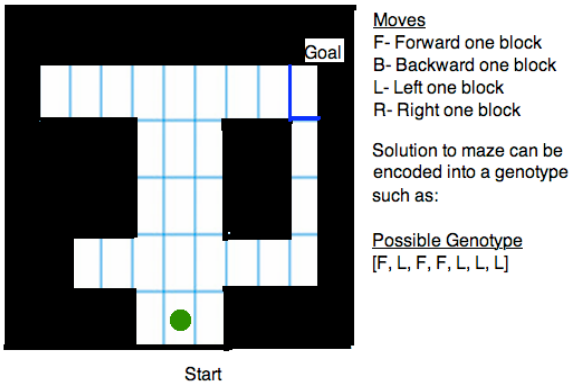

Figure 4.4: Application of genetic algorithm to a navigation problem.

Here the goal is get the green robot from the start location to the goal location. The sequence of steps it takes can be encoded into a genotype. The success or the fitness of this genotype can be evaluated by how close the green robot gets to the goal square.

In a genetic algorithm a population of random genotypes is created and their fitness is evaluated. The fitness function is different for each task. Here the fitness function is how close the green robot gets to the goal. After each genotype is evaluated the ones with the highest fitness are allowed to cross and produce offspring as shown in figure 4.5. Also there is a chance for mutation of the offspring.

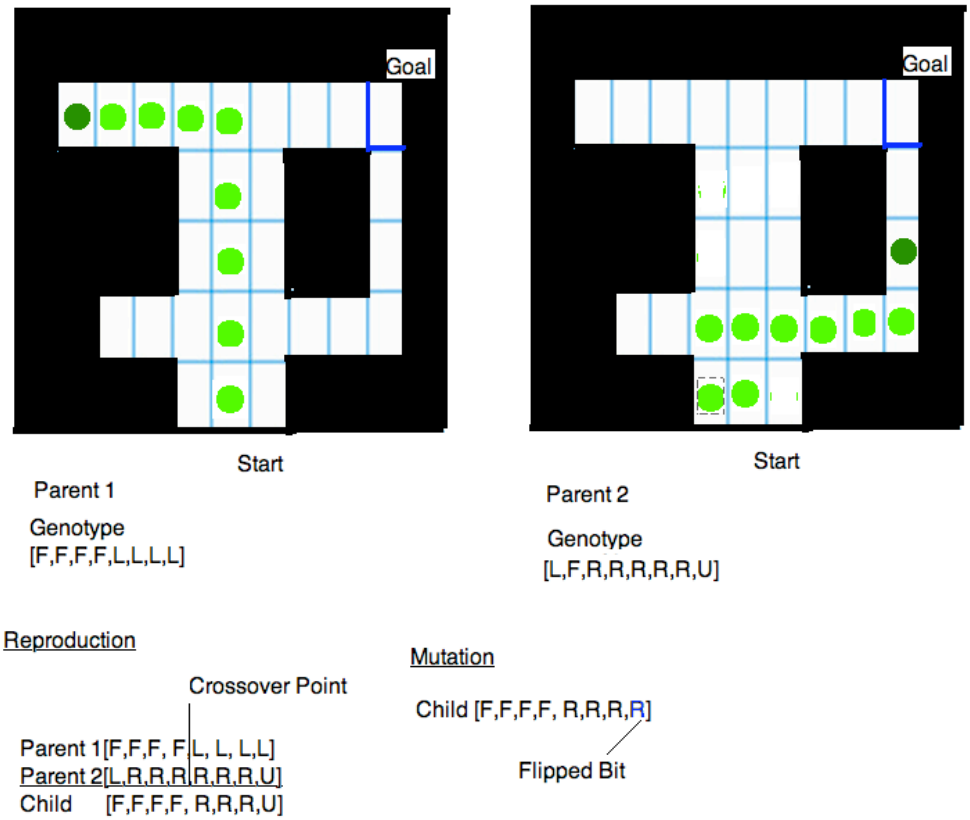

Figure 4.5: Illustration of reproduction with a genetic algorithm.

If this is done many times over many generations the fitness of the population will slowly rise until a solution is found as shown in Figure 4.5.

NEAT

NEAT which stands for the NeuroEvolution of Augmenting Topologies, is a genetic algorithm which evolves neural networks. The neural networks are encoded using their connections. Arrays are made which encode the strength of each connection, the from and to nodes, its innovation number, and whether or not the connection is disabled.

Mutations in NEAT alter the weights of the connections as well as add more connections and nodes. This is how the topologies of the neural networks are altered. Figure 4.7 illustrates how connections and nodes are added to the networks.

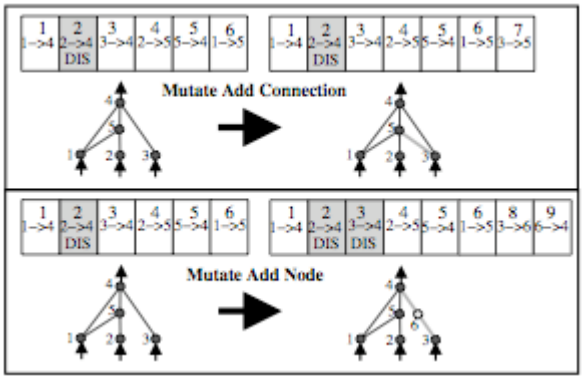

Figure 4.7: Diagram illustrating how mutation works in NEAT. Taken from the paper on NEAT(4).

Adding a new connection is fairly simple. There is not limit to where connections can be added. Nodes are added within connections. Node 6 in Figure 4.7 is added in the middle of the connection between nodes 3 and 4. The connection and its weight between 3 and 4 is reassigned to be between 3 and 6 and a new connection is added between nodes 6 and 4.

The algorithm keeps a record of all the connections created, and gives them a number based on their order. Note that the innovation numbers are not the order that connections are created within an individual neural network. They area the order connections are created in reference to the entire evolution run. NEAT uses these innovations numbers to line up the genotypes of the neural networks and perform crossover. This is shown in figure 4.8.

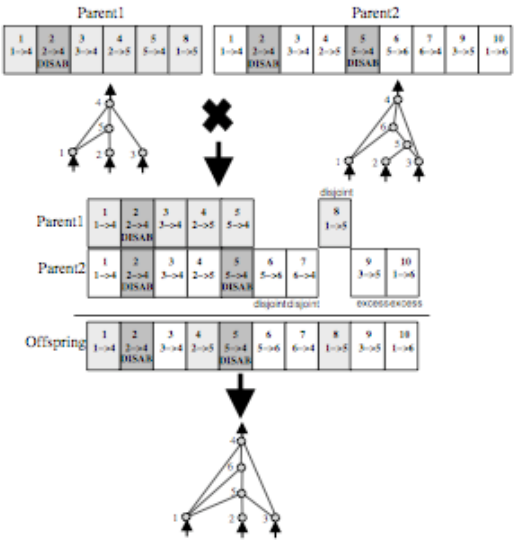

Figure 4.8: Illustrations of how crossover works in NEAT. Taken from the paper on NEAT(4).

The innovation numbers give a framework for taking genetic information from one part and mixing it with the other parent. The connections can be very complex and intricate. Just picking and swapping connections from the two parents to the make a child doesn't preserve the original structure which contributed to the high fitness of the parents. Using the innovation numbers to perform crossover keeps the general structure of each parent, but allows a mixing of the genetic information. From a biological standpoint this is similar to only crossing genes with a common history. All the genes associated with the vertebrate of humans share a similar history while all the genes associated with the opposable thumbs of humans also share a common history during evolution. You would want to cross the genes for thumbs together and the genes for spines together.

4.4 MATLAB implementation of NEAT

NEAT was originally developed by Kenneth Stanley. His implementation was originally written in C++. It has since been ported to various languages such as python, Java, and MATLAB. A MATLAB implementation of NEAT was used for this project. It can with a sample experiment which solved XOR.

Neural network setup

The implementation of XOR loaded inputs into the input layer of the neural network and waited until the values fully propagated through the network before returning the outputs. It waited until the output values were stable. It seems biologically unrealistic to expect activations to reach the output layer at the same time even though they take different lengths of paths through the neural network. This also stops activations from reaching the output with different delays.

The propagation code was changed to do the following. Each time step the algorithm goes through each connection and increments the activation of the node it is going to by the weight of the connection times the activation of the node the connection is coming from. Once all the connections are allowed to send their information the output is evaluated and the next round of inputs enters the net. The code does not sit there and wait until the values at the output node reach a steady state before loading in the next set of inputs.

The neural networks for these experiments have 6 inputs nodes, two output nodes, and one bias. An initial neural network is shown in Figure 4.9. The input nodes are fully connected to the output nodes. The connections weights range from -1 to 1.

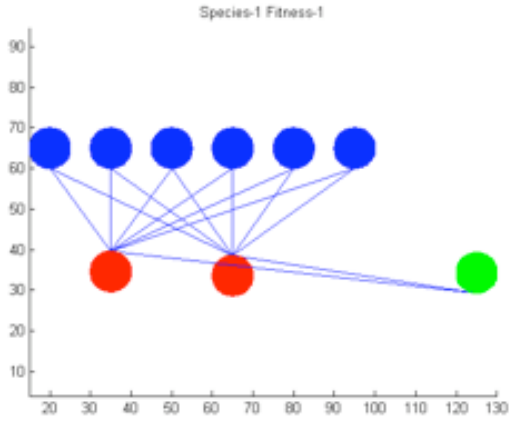

Figure 4.9: Initial network setup.

The six inputs to the neural network, from left to right, are the position of servo 1, 2, 3. The reading from the force sensor, the reading from the odometer, and the contact sensor. The two outputs go to servo 2 and servo 3. The values from the output nodes would be sent to the robot as '0030' commands. This means that the value output from the neural network will increment or decrement the CCP values for Servos 2 and 3 by some amount. The next section on the activation function goes into more detail on how the activation at the output controls the servos.

Activation functions

The activation for the input nodes in linear function of 1. That is the input values are simply passed the straight to the output. Before the sensors values are sent into the input nodes they are scaled linearly to between 0.1 and 0.9. For the position sensors this means they are scale from 80 to 400. For the force sensor and odometer they are scaled from 0 to 1023. And for the contact sensor it is scaled from 0 to 1.

The activation of the hidden and output nodes is sigmoid function centered at 0.5.

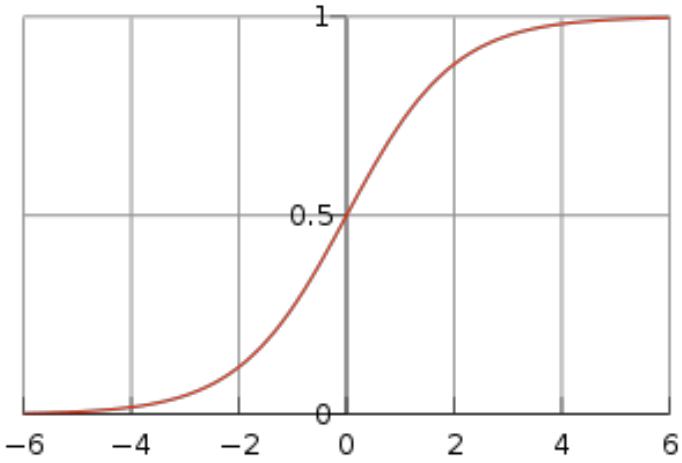

Figure 4.10: Sigmoid function used as the activation function of the hidden and output nodes. Image taken from 11http://en.wikipedia.org/wiki/Sigmoid_function

With the equation $A(t) = \frac{1}{1 + e^{-t}}$

The input to each node was scaled to be between -6 and 6 with the formula

$$t = \left(\frac{inputActivation}{weightCap} \right) 12 - 6$$

The weightCap for the connections is 1 because the weights can range from -1 to 1. The activation function of each hidden and output node is then

$$A(t) = \frac{1}{1 + e^{-\left(\frac{inputActivation}{weightCap} 12 - 6 \right)}}$$

If an node feeding into an output node or hidden node was firing at the highest activation possible, which is 1, and the weight between that node and the output or hidden node was at its maximum positive value then the activation of the output or hidden node would be 1. If an node feeding into an output node or hidden node was firing at the low activation possible, which is 0, and the weight between that node and the output or hidden node was at its maximum negative value then the activation of the output or hidden node would be close to 0. All other activations are scaled from this.

The activation of the output nodes ranges from 0 to 1. As was said earlier, the output nodes would send the servo words for sending a '0030' command to the robot. The values 0 and 1 are scaled linearly between -999 and 999 before being sent to the robot. -999 and 999 are the largest commands that can be sent to the robot with a '0030'. With a '0030' command the output nodes are incrementing or decrementing the servos by some value every time step. They are essentially setting the speed of servo 2 and Servo 3.

4.5 NEAT Experiment

The evolution experiments were not run on the real robot. This would have taken a lot of time and would have probably results in the robot destroying itself. Some initial runs were run on the actual physical robot. In these runs the robots learned to hit themselves thereby using the inertia of the impact and their arm to move backwards.

Instead of running experiments on the real robot the simulator described in section 3.4 was used. The task for this robot was to move. Its fitness was how much net displacement it gained in 450 time steps. The displacement is measured by the virtual odometer at the back of the simulated robot.

Three sets of four experiments were run with different mutation rates. Most of the parameters from the XOR experiments were kept. The ones altered at listed below. The only difference between the three sets of four experiments are the mutation rates.

Parameter Name	Value	Description
Population_size	30	
number_input_nodes	6	
number_output_nodes	2	
Initial.kill_percentage	0.5	Percentage of bottom performers eliminated every generation.
selection.pressure	1.95	Determines selective pressure towards most fit individual of species
mutation.probability_add_node	0.07(Run 23-27) 0.03(Run 28-31) 0.05(Run 31-35)	
mutation.probability_add_connection	0.09(Run 23-27) 0.05(Run 28-31) 0.07(Run 31-35)	
mutation.probability_recurrency	0.05	
stagnation.threshold	2500	Threshold to judge if a species is in stagnation.

Table 4.1: Parameters used for the evolution trials that are different from the XOR sample experiment.

Figure 4.11 shows the max fitness versus generation for the three sets of four experiments. They are clearly very similar which means that the mutation rates is not affecting how fast the neural networks are becoming fit. At least not at the rates tested. Therefore from here on the three sets of data will be lumped together and treated the same.

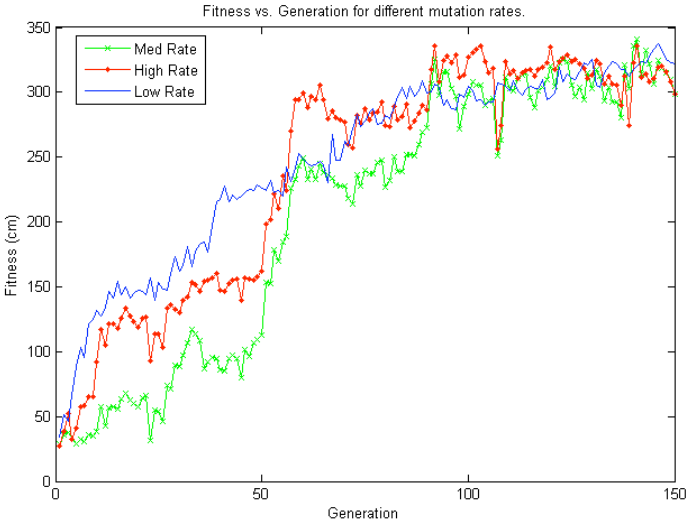

Figure 4.11: Fitness for the different mutation rates.

The average max and mean fitness are shown in figure 4.12.

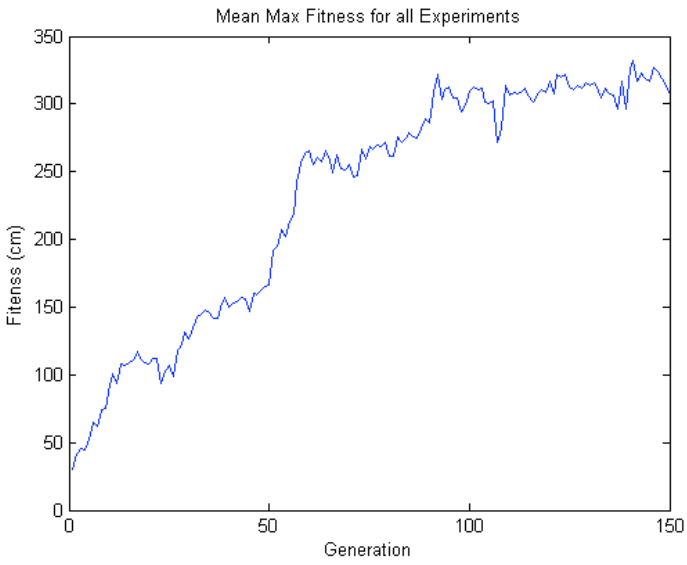

Figure 4.12: Combined fitness for all experiments.

The figures 4.11 and 4.12 show that the neural networks were able to solve the task and evolve crawling behavior. The evolution runs are repeatable and stable. Every run produced results.

Two different types of ANN evolved. Some ANN evolved which made the simulated robot move backwards (pushers) and some ANN evolved which made the simulated robot move forwards (pullers). For the most part most of the evolution runs produce backward or pushing ANN. The difference between the two types of behaviors and why one was more preferred than the other is discussed in the next section.

Porting to Robot

Some of the best neural networks were ported onto the real robot. Quantitatively it's hard to compare the performance of the ANN on the real robot with the ANN on the simulated robot so the distance traveled in the real world wasn't compared to the distance traveled on the real robot.

Qualitatively the ANN worked beautifully on the real robot. The pushing and pulling ANN make the real robot move incredible fast. These solutions go about four times as fast as the Direct Programming solution. Videos of the ANN ported to the real robot can be found on YouTube by searching for "crawling robot, Lola, NEAT"

5. Analysis of Artificial Neural Networks

NEAT was successful at evolving ANN to produce crawling motion, but how are the ANN producing the crawling motion? How are the ANN successfully coordination the motion of Servo 2 and Servo 3? The analysis of the ANN required writing code which plotted the ANN and showed how the information was propagating through the ANN. This analysis yielded some very interesting results which tie back into the earlier discussion on locomotion and oscillators.

5.1 Locomotion and Pattern of Activation

After the tasks were completed the artificial neural networks were analyzed in order to understand how they produced the crawling motion. The following images show screen shots of the simulated robot alongside its artificial neural network. The images go step by step and show how the pattern of activation of the nodes produces, in this case, pushing movement.

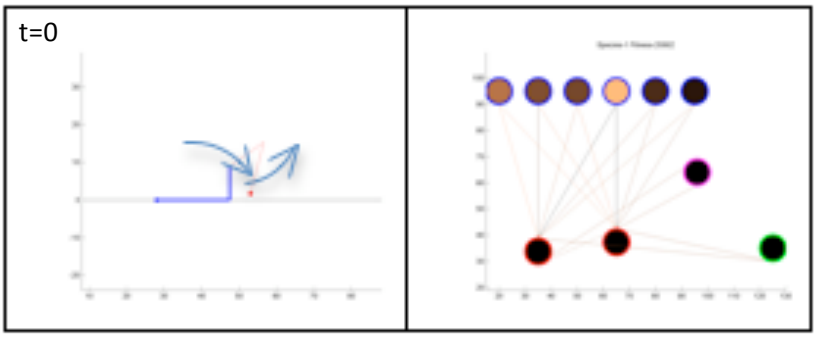

Figure 5.1: Simulated robot and ANN at t=0.

At t=0 both output nodes produce low activation. This in turn will cause linkage 3 to rotate counter clockwise and linkage 2 to rotate clockwise as shown in figure 5.1. The arm will do a stretching out motion until it makes contact with the ground.

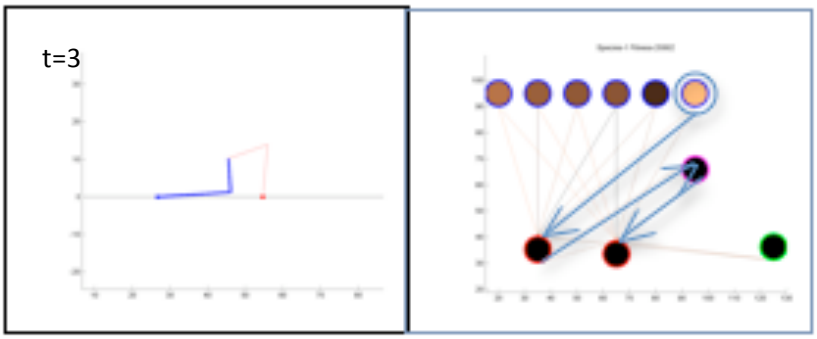

Figure 5.2: Simulated robot and ANN at t=3.

At t=3 the arm makes contact with the ground, lifting the chassis off the ground. The sixth input node, which corresponds to the contact sensor located at the bottom of the robot, goes from low activation (black) to high activation (orange). As the animation continues the activation will spread from input 6 to output 1, to the hidden node, to output 2 as indicated by the arrows in figure 5.2. NEAT has strengthened these weights over the course of evolution.

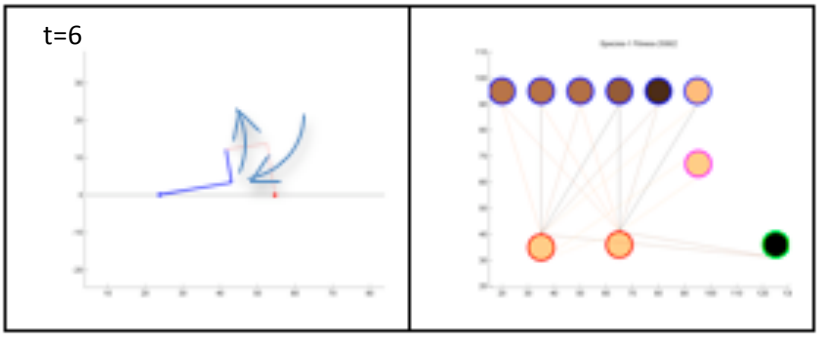

Figure 5.3: Simulated robot and ANN at t=6.

At t=6 the activation of the two outputs is now high. This causes linkage 3 to rotate clockwise and linkage 2 to rotate counter clockwise as shown if figure 5.3. Note it took two time steps for the activation from spread from output 1 to output 2. This means that linkage 2 changed directions(rotate counter clockwise) two time steps before output 2.

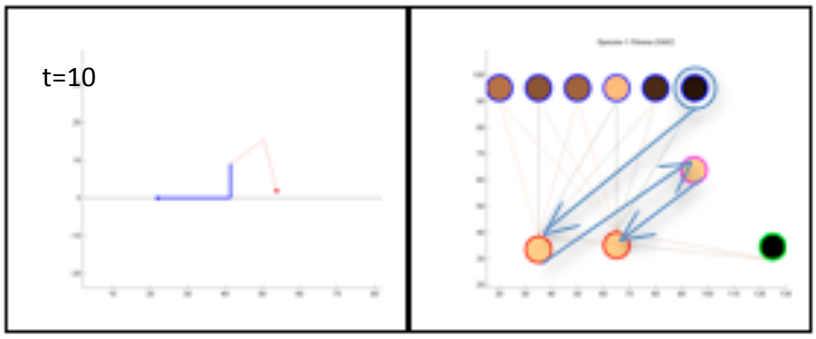

Figure 5.4: Simulated robot and ANN at t=10.

At t=10 the chassis makes contact with the ground and input 6 gets pulled low as shown in figure 5.4. The activation will now travel through the artificial neural network the same way it did between t=3 and t=6. This will bring the robot into a state seen at t=0, and the motion will repeat.

This is an example of how one of the artificial neural networks was controlling the simulated robot and making it push backwards. After more analysis it was discovered that all the pushers had a similar flow of activation.

5.2 Difference Between Pushers and Pullers

The ANN analyzed in section 5.1 shows a typical ANN which generated pushing motion. ANN which generated pulling motion also evolved during the experiments. Fundamentally there isn't much difference between pushing ANN and pulling ANN. Figure 5.5 shows examples of two ANN. One represents the simplest pushing ANN and one represents the simplest pulling ANN.

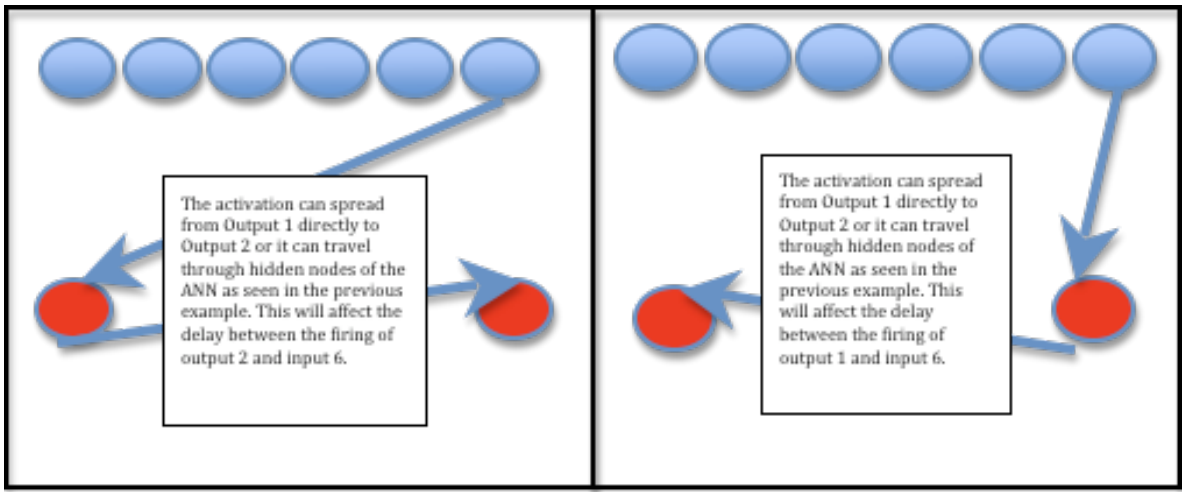

Figure 5.5: Simple illustration of how pulling ANN differed from pushing ANN. The ANN on the left is a pushing ANN. The ANN on the right is a pulling ANN.

As seen in figure 5.5 the only difference between pushing ANN and pulling ANN is the delay in firing output 1 and output 2 with respect to the contact sensor (input 6). For pushing ANN on the left, output 1 will fire one time delay after the contact sensor fires and output 2 with fire 1 plus some delay, equal to the number of hidden nodes between output 1 and output 2, after the contact sensor. For pulling ANN on the right, output 2 will fire one time delay after the contact sensor fires and output 1 with fire 1 plus some delay, equal to the number of hidden nodes between output 2 and output 1, after the contact.

5.3 ANN and Oscillators

To further elaborate on the difference between the pushing and pulling ANN and to link to the discussion of neural networks, oscillators, and central pattern generator mentioned in the introduction, the activation of the contact sensor, output 1, and output 2 was plotted versus time step. Two ANN and the plot of their activations versus time are shown in figure 5.6 and figure 5.7.

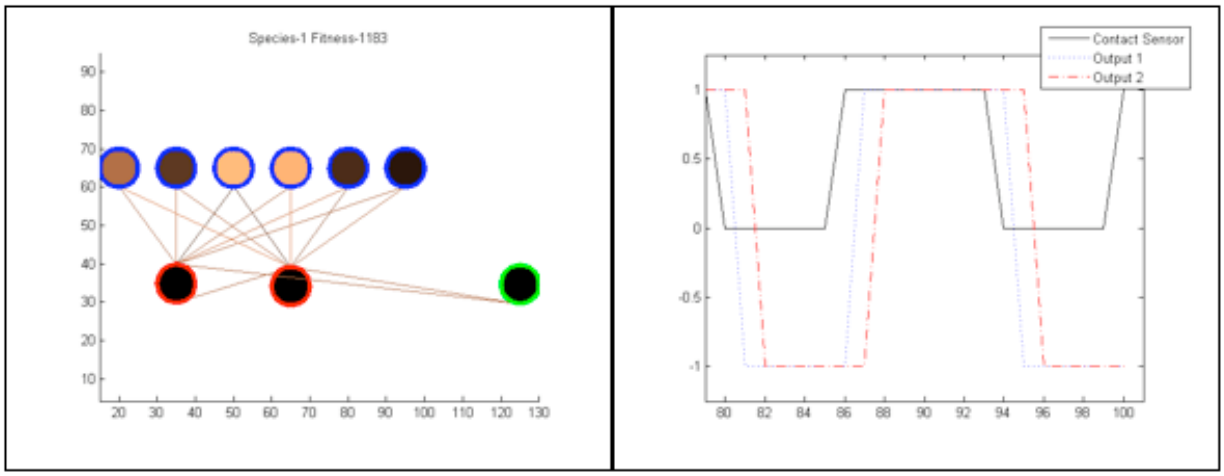

Figure5.6: A very simple pushing ANN and a plot of its activation versus time steps.

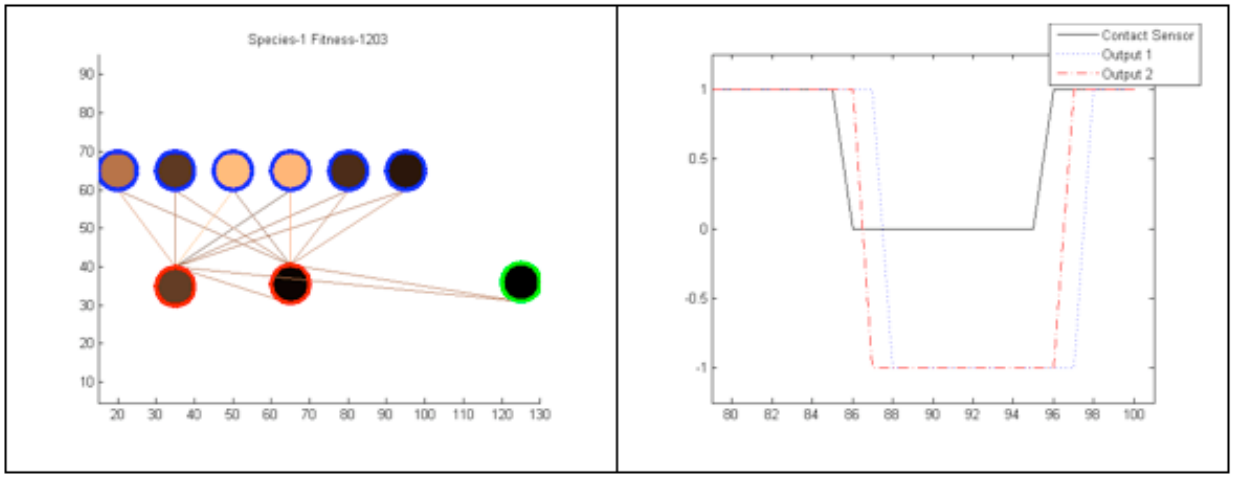

Figure5.7: A very simple pulling ANN and a plot of its activation versus time steps.

For the pushing ANN shown in figure 5.6 the dashed line, which corresponds to output 2, fires two time steps after the solid line, which corresponds to the contact sensor. The dotted line, which corresponds to output 1, fires one time steps after the solid line, which corresponds to the contact sensor. Output 2 is said to have a phase shift of two while output 1 is said to have a phase shift of one from the contact sensor.

For the pulling. For the pulling ANN shown in figure 5.7 the dashed line, which corresponds to output 2, fires one time steps after the solid line, which corresponds to the contact sensor. The dotted line, which corresponds to output 1, fires two time steps after the solid line, which corresponds to the contact sensor. Output 2 is said to have a phase shift of one while output 1 is said to have a phase shift of two from the contact sensor.

For both pushing and pulling ANN the output nodes are entrained to the contact sensor. The fundamental difference between pulling and pushing ANN are the phase shifts of output 1 and output 2 with respect to the contact sensor. Pushing ANN seem to have the phase shift of output 2 greater than the phase shift of output 1 while pulling ANN seem to have the phase shift of output 1 greater than the phase shift of output 2.

5.4 Exploring Phase Shifts

To explore this idea further a MATLAB script was written which plants the simulated robot's arm, to start the oscillation, and then varied the phase shifts for output 1 and 2. The simulated robot was allowed to move for 450 time steps and its fitness was recorded. The results for this test are shown in table 5.1.

	Phase Shift									
Output 2	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
Output 1										
1.0	15.7	109.8	193.8	218.7	238.3	258.2	235.7	254.1	268.8	293.1
2.0	-99.5	9.6	112.8	172.6	177.2	227.1	222.7	226.0	245.7	255.8
3.0	-224.7	-91.1	11.9	111.5	149.3	177.9	194.8	224.5	223.1	225.1
4.0	-297.0	-207.0	-58.7	46.2	92.1	152.2	156.8	195.8	201.9	217.4
5.0	-284.2	-226.8	-161.2	-39.5	41.8	105.5	121.3	171.1	184.7	198.1
6.0	-112.4	-305.7	-227.5	-139.5	-27.1	30.7	63.6	142.8	160.9	182.0
7.0	-19.2	-129.5	-286.3	-180.5	-93.6	-27.9	12.9	56.1	131.6	158.5
8.0	-20.1	-61.9	-159.2	-257.4	-146.4	-79.9	-30.8	15.5	85.9	125.2
9.0	-13.8	-17.4	-63.6	-282.1	-216.6	-158.1	-104.1	-22.5	37.6	90.9
10.0	-15.1	-16.3	-6.5	-125.0	-228.6	-220.0	-151.4	-87.8	1.2	32.1

Table 5.1: This table shows the fitness values for a given phase shift combination.

The horizontal and vertical axes on the edges indicate the phase shift or delay between either output 1 and the contact sensor or output 2 and the contact sensor. The values in the boxes show the fitness (in centimeters) that the simulated robot achieved after 450. The negative values grayed out indicated that the simulated robot moved backwards. Table 5.1 confirms the previous assertion that pushing ANN occur when the phase shift of output 2 is greater than the phase shift of output 1.

Table 5.1 shows that the changing of the phase shift can greatly change the fitness. This was seen in some of the evolution trials where there were huge spikes of fitness. Figure 5.8 shows the fitness versus generation of one of the evolution runs.

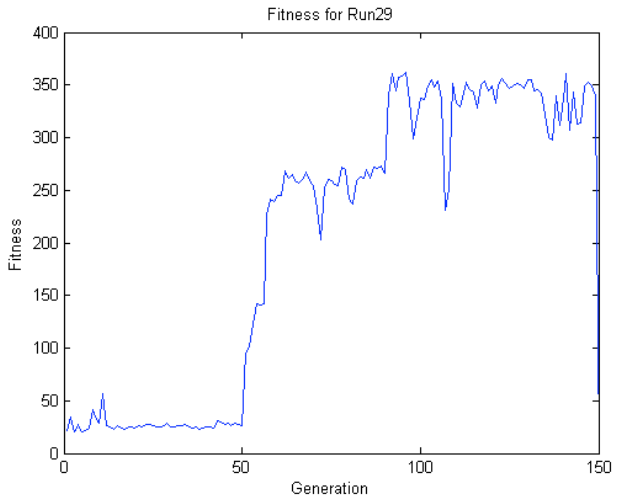

Figure 5.8: Fitness versus generation for experiment 29. Shows the sharp increases in fitness due to innovations.

The huge spike at generation 50 occurred because some innovation in the evolution (addition of connection, node, or changing of weight) produced an ANN with a good set of phase shifts. The same thing happens at around generation 80.

5.5 Pushers better than Pullers

The addition of connections, hidden nodes, and the changing weights are random events which occur during the evolution process. The rate of these events are controlled by the mutation rates set in the NEAT algorithm. Figure 5.9 shows two ANN with similar levels of complexity. They both have an extra node and two extra connections not found in the original ANN setup.

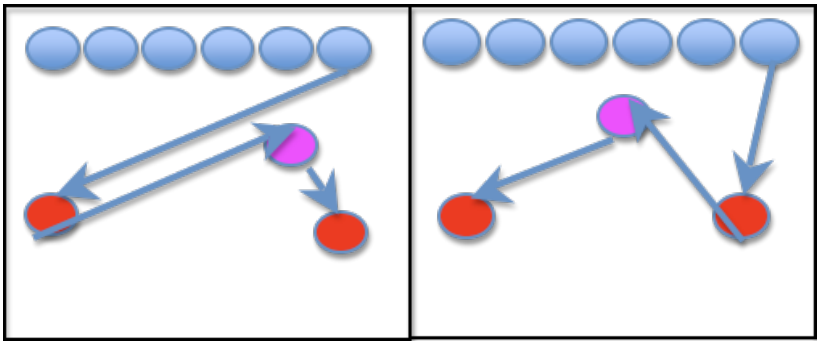

Figure 5.9: Two ANN with the same level of complexity. The one on the left is pushing ANN and the one on the right is a pulling ANN.

The ANN on the left has a phase shift of [1,3] and based on Table 5.1 would have a fitness of around -224.7. The ANN on the right has a phase shift of [3,1] and a fitness of around 193.8. Looking at Table 5.1 this trend is fairly consistent with a few exceptions. For a certain level of complexity the pushing ANN go faster than the pulling ANN. This becomes obvious when looking at the pattern of activations of a pushing and pulling ANN in figure 5.10. Figure 5.10 shows the same oscillators in figure 5.6 and 5.7 except zoomed out. It can be seen that the pushing ANN undergoes more oscillations or cycles, in 100 time steps, than the pulling ANN. This means the pushing ANN gets more strokes in than the pulling ANN which is why they are on average faster than the pulling ANN.

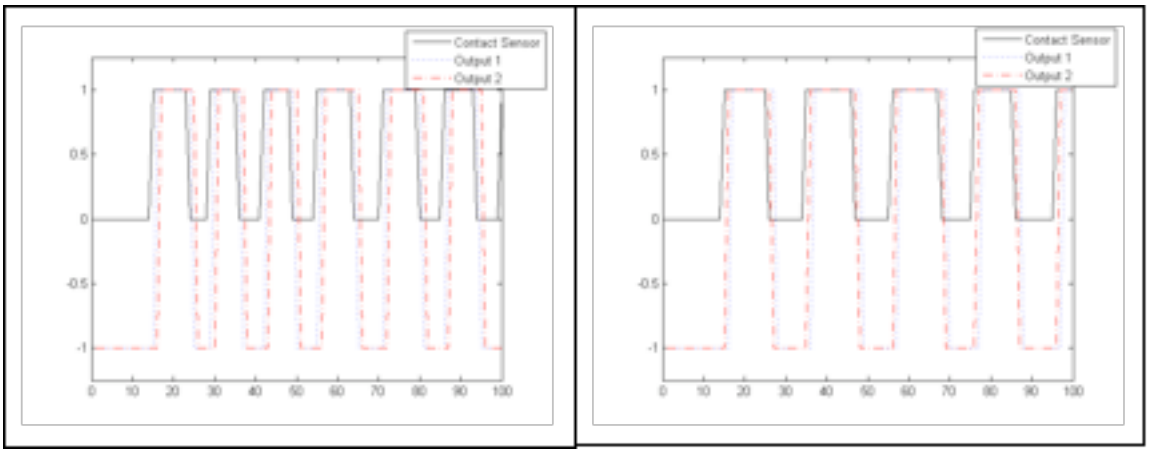

Figure5.10: Zoomed out images of the plots of the activations shown in figure 5.7 and 5.8.

This is one possible explanation for why pushing ANN are more common, during the evolution trials, than pulling ANN. For the same level of complexity, the pushing ANN are inherently faster and dominate the populations causing the extinction of the pulling ANN.

The increased number of cycles for a pushing ANN, when compared to a pulling ANN, is most likely caused by the geometry of the robot. Alterations to the robot's chassis were not explored in order to try to see if the pulling behavior could be made better than the pushing behavior.

6. Conclusion

In this project a fairly simple, yet orthodox robot was created quickly. It proved to be very robust and easy to work with. Classic engineering and computer science approaches to programming and control were applied to the robot and satisfactory results were obtained. Control was learning and adaptation using artificial neural networks and genetic algorithms was then explored with great success. This project showed the feasibility of a method like learning with artificial neural networks and genetic algorithms to control a mobile robot.

The ANN and GA produced very robust and interesting solutions to the problem. It's not hard to imagine taking the understanding gained from the analysis of the ANN and reverse engineering the solution. That is, take this idea of the coupling of inputs and outputs and oscillators to hand code a solution which would work as well as the evolved ANN. These learning approaches have shown a viable way to program a robot and learn new ways to conceptualize the control of a mobile robot.

References

1. Crespi, A.J.. " AmphiBot I : an amphibious snake-like robot." Robotics and Autonomous Systems 50(2008): 163-175.

2.Crespi, A.J.. "Controlling swimming and crawling in a fish robot using a central pattern generator." Autonomous Robots 25(2008): 3-13.

3.Crespi, A.J.. " From swimming to walking with a salamander robot driven by a spinal cord model" SCIENCE 315(2007): 1416-1420.

4.Stanley, Kenneth. "Competive coevolution through evolutionary complexification." Journal of Artifical Intelligence Research, vol 21. 2004