2000

Machine learning and small robot navigation

Christopher Ireland
Colby College

MACHINE LEARNING AND SMALL ROBOT NAVIGATION

by

CHRISTOPHER IRELAND

Submitted in Partial Fulfillment of the Requirements of the
Senior Scholars Program

COLBY COLLEGE
2000

Christopher Ireland
Senior Scholar Approvals

APPROVED:

Clare Bates Congdon

Clare Bates Congdon

[illegible]

Randolph M. Jones

Dale Skrien

Dale J. Skrien

[illegible]

Fernando Gouvêa

Abstract

Developing successful navigation and mapping strategies is an essential part of autonomous robot research. However, hardware limitations often make for inaccurate systems. This project serves to investigate efficient alternatives to mapping an environment, by first creating a mobile robot, and then applying machine learning to the robot and controlling systems to increase the robustness of the robot system. My mapping system consists of a semi-autonomous robot drone in communication with a stationary Linux computer system. There are learning systems running on both the robot and the more powerful Linux system.

The first stage of this project was devoted to designing and building an inexpensive robot. Utilizing my prior experience from independent studies in robotics, I designed a small mobile robot that was well suited for simple navigation and mapping research. When the major components of the robot base were designed, I began to implement my design. This involved physically constructing the base of the robot, as well as researching and acquiring components such as sensors. Implementing the more complex sensors became a time-consuming task, involving much research and assistance from a variety of sources.

A concurrent stage of the project involved researching and experimenting with different types of machine learning systems. I finally settled on using neural networks as the machine learning system to incorporate into my project. Neural nets can be thought of as a structure of interconnected nodes, through which information filters. The type of neural net that I chose to use is a type that requires a known set of data that serves to train the net to produce the desired output. Neural nets are particularly well suited for use with robotic systems as they can handle cases that lie at the extreme edges of the training set, such as may be produced by "noisy" sensor data. Through experimenting with available neural net code, I became familiar with the code and its function, and modified it to be more generic and reusable for multiple applications of neural nets.

The next stage of my project involved implementing my neural net system on my robot. My first task for the robot involved creating a system that would allow the robot to track a light source. The next application of neural nets was a system that interpreted the

data returned by a ranging sensor, putting this distance information in terms of units relative to the robot. These two networks proved to be very successful and useful.

The third and largest application of neural nets in my system was a system that would determine the best way for the robot drone to map an unknown environment. I implemented a system that would generate a number of possible paths for the robot to pursue to gather information about the environment, and then upload that information to the robot. The third neural net is the system that chooses which of those possible paths is would be the useful to pursue. This net examined a representation of each path, and output a measure of the projected usefulness and success of the path.

At this point I also created an environment in which to run my robot and test the mapping system. This is also when I discovered a problem with the compass on my robot. This sensor problem prevented the robot from consistently knowing which direction it was moving in for more than a minute or two, and essentially crippled the useful functions of the robot. This prevented testing the third neural net, as well as the overall mapping system.

Due to these hardware issues, it is impossible to draw any overriding conclusions. However, I completed most of the project with positive results. The robot I built turned out to be very successful, despite the issues with one sensor. I was able to apply neural nets to two aspects of controlling the robot, and the software system for controlling the robot is quite large and extensive. Overall the project has promising results, and was a tremendous experience. There are also many areas that remain for future research, including testing the remainder of my mapping system, and introducing more variables such as multiple robots, and implementing other machine learning systems.

Acknowledgements

There are many people who played important roles in this project. The first person to thank is my advisor, Clare Bates Congdon. Clare's role was quite extensive, as she was a resource of knowledge, materials, and a perspective very different from mine. This project would not have been possible without Clare's knowledge of robotics and machine learning, and would not have gotten very far without our brainstorming sessions. Also at the top of the list are my readers for this project, Randy Jones and Dale Skrien. They have offered help and suggestions at various points throughout the semester, and were kind enough to read through my paper multiple times. Allen Downey of the Computer Science department has also offered suggestions and help over the course of the year.

I would also like to thank the Dean of Faculty's Office and the Student Special Projects Fund for providing a budget with which to make the purchases necessary to build my robot. Along the same lines, the Computer Science department and NSF AIRE (National Science Foundation, Awards for the Integration of Research and Education) grant have also made additional funding available to cover expenses, including traveling expenses for a recent conference. I would also like to thank the Biology and Geology departments for making space available to me over the past year and a half when I have been pursuing this research.

Chuck Jones has been a great resource for help with interpreting electrical schematics, and implementing hardware by offering tips and assistance with soldering. Also deserving of thanks are the creators of the Handy Board at MIT, and the Handy Board web site and mailing list. Without these two resources, I would not have been able to implement much of my robot.

Additional sources of support throughout this project have been my friends and family. Finally I would like to thank my roommates for their help in my research, particularly their decision not to move my belongings out of my room when I spent those long nights in my office.

CHAPTER 1
INTRODUCTION

The primary goals of this project are to design and build a robot, and to apply machine learning to a robotic navigation and mapping system, with the hope of creating an efficient mapping system for inexpensive small robots. Many small robots are created with less than perfect hardware systems, as quality is often sacrificed for considerations of size and expense. Due to these hardware limitations, it is often challenging to create mapping systems for these robots. My robot is largely homemade, and therefore incorporates many of these hardware issues. Through the application of neural networks to several aspects of the robotic mapping system, I have created effective solutions to some of these problems.

My system is composed of a homemade semi-autonomous robust robot, controlled by a simple on-board computer. The robot includes a navigation system composed of an electronic compass and a shaft encoder, which serves as an odometer. The robot also uses a series of light sensors in a light tracking system, which allows the robot to return to a known location, signified by a light source, in the event of becoming lost. The robot also incorporates a series of touch sensors, as well as an infrared ranging sensor for obstacle detection and mapping purposes. The robot serves as a drone that reports back to a more powerful Linux computer. A fully autonomous robot is one which is entirely under its own control. I consider my robot to be semi-autonomous as it receives instructions from and reports data back to a second computer system.

The Linux machine is included due to its powerful computing and processing abilities. The Linux computer serves as a central controller and an information processor and data repository. The goal of the software running on the Linux computer is to collect enough data to successfully map the environment. Furthermore, the Linux software also evaluates and improves upon the general strategy of mapping an unknown environment.

In general, the Linux software generates a number of possible paths for the robot to pursue, and picks what it determines to be the most useful path based on the amount of data that will be collected. The robot then executes this path, gathering information about the environment as it goes. This information is then reported back to the Linux machine, which examines it, incorporates it into the map, and updates the strategies of choosing paths for the robot. This process is shown in more detail in Figure 1.1.

In an effort to maximize the robustness of this system, machine learning is applied in several areas. The type of learning system used is a neural network. First, a neural network is incorporated as part of the light tracking system. The network takes readings from three light

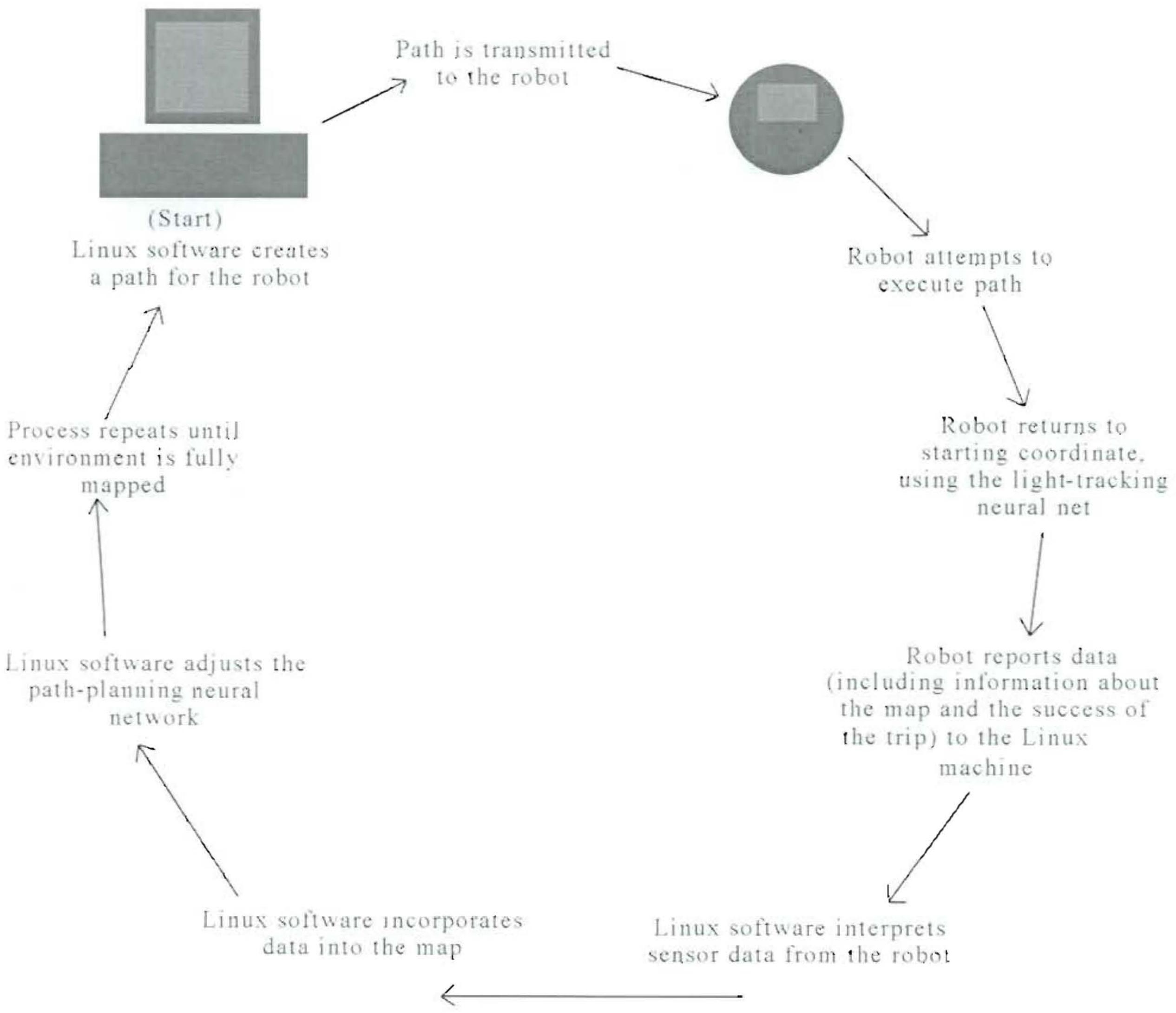

Path is transmitted to the robot
Robot attempts to execute path
Robot returns to starting coordinate, using the light-tracking neural net
Robot reports data (including information about the map and the success of the trip) to the Linux machine
Linux software interprets sensor data from the robot
Linux software incorporates data into the map
Linux software adjusts the path-planning neural network
Process repeats until environment is fully mapped
(Start) Linux software creates a path for the robot

Figure 1.1 – General process

sensors as its input, and outputs a direction for the robot to turn in order to move towards the light source. This network runs on the robot itself, as light tracking only occurs when the robot is operating in autonomous mode, and is not in communication with the Linux computer.

Next, the Linux software runs a network that interprets readings from the robot's infrared sensor. The robot uploads information from this sensor, including the location of the sensor reading as well as the reading itself. The network then interprets this information as a distance that is relevant to the map. This interpretation net converts sensor readings into distances relevant to the robot. A non-learning system then converts these distances into locations on the map by resolving the distance to an object, heading of the robot at that point, and the location of the robot at the time of the reading.

The third application of neural nets is to the general strategy of how to map an environment. This net has a series of inputs that includes a representation of the portion of the map that is relevant to the path in question. As more data-gathering excursions are made by the robot, this net is trained to accurately identify which paths of travel are the best to pursue, in order to map the environment in the least amount of time.

Many challenges and unexpected problems were encountered during my pursuit of this project, making it a tremendous learning experience. Implementation problems with one of the sensors I used held up the testing stage of the neural net for designing a mapping strategy. I successfully implemented and completed all other aspects of the project, including building my robot, designing and implementing the neural networks and extensive systems for controlling the robot and utilizing the networks, and testing all systems but the strategy forming network in an actual environment.

This document will describe the design and implementation of all aspects of this project. Chapter 2 provides a relevant background in robotics, machine learning, and navigation. Chapter 3 covers the design and implementation of the hardware that comprises my robot. Chapter 4 describes the software that I used and constructed for this system. Chapter 5 provides a discussion of the experiments that I performed with neural nets. Chapter 6 offers some of my reflection upon this project, and my thoughts for future work to continue this research.

CHAPTER 2
BACKGROUND

2.1 Introduction

Before I could begin to create a robot, I needed to experiment and research robotics and current topics in the field. My initial experience with robots consisted of experimenting with the Handy Board (a compact computer designed at MIT for small robots) and custom Lego sets. This introduction steered me towards interest in a navigation and mapping system. Research into these areas revealed how important and fundamental the topics are.

Another aspect of my project is machine learning. Prior to designing and creating the learning systems that my project utilizes, I needed to experiment with different types of learning systems. This involved research and using available code to get a feel for the strengths and weaknesses of the options available to me. My efforts in this process finally amounted to choosing neural networks as the system that I implemented in the project.

2.2 Robots

My introduction to robotics began with two independent studies during the second half of my junior year. These independent studies covered basic concepts and simple reactive robots such as Braitenburg vehicles[1]. Braitenburg vehicles are some of the simplest robots that can be created, and involve reactive systems with very simple control structures. These studies were performed using special Lego kits designed for small robot experiments. The Lego system applied to small robots works very well on a prototype level, however the robots do not really stand up to long-term or realistic rough handling. This fact introduced a number of hardware issues by itself, but there were many other issues encountered during these studies.

One of the more basic operations that is necessary for many of the possible systems is a navigation system. A navigation system allows a robot to maneuver through an environment safely. The inclusion of a mapping can allow a navigation system to operate in a more intelligent fashion, by affording knowledge of the environment. A mapping system gives a robot the ability to

[1] Braitenburg, Valentino, "Vehicles, Experiments in Synthetic Psychology"

actually map the environment on its own. The process of navigation and mapping became much more challenging than I expected, due to the various hardware problems that I encountered with the Lego system. Creating a system for navigation and mapping became topics of interest to me, and I chose to pursue these topics as the main focus of my Senior Scholar project.

Navigation is a topic that is useful and required by virtually every mobile robot system, from academic and research robots to commercial and industrial robots. Many of these units also rely on some sort of mapping system, whether it be the process of mapping an environment, or using a map to navigate in an environment. A source of inspiration for me in regards to robotics in general as well as the topics of navigation and mapping is a Somerville, MA based robot company, IS Robotics[2]. Much of the research done at this company is performed with the support of military and government funding. Many of these projects require a robot which is capable of maneuvering in unknown hostile environments, and many require the robot to serve as an autonomous reconnaissance robot, reporting back information about the environment the robot is infiltrating. Also, many of these robots must create some sort of map of the environment that they encounter, both as information gatherers, and to provide a way for the expensive robot to return back to home base and be saved for future use. These projects and others on IS Robotics' web site were great sources of inspiration for me in thinking of my own project.

A more public example of a navigation and mapping system in use is apparent in the much-publicized Mars Pathfinder mission[3]. While this robot is far more complicated than my system, the concepts and necessity for a robust navigation system are very important for this system. The Sojourner robot received a substantial amount of instructions from Earth-bound controllers. The design of the Sojourner robot is very similar to my own, as both systems are composed of a central controller and a robot drone. The Sojourner robot was also equipped with various sensors to detect obstacles, and ways of handling situations related to maneuvering. The importance of an intelligent navigation system is very clear given the limited life span and extreme expense of such a robot.

There are many other useful projects where a robust navigation system is required. One of the major areas of research focuses on constructing robot systems to function in an office environment. Indeed, Nils Nilsson has issued a challenge[4] to mobile robot researchers to create an autonomous system for use in an office environment. This challenge relies heavily on an effective

[2] http://www.isr.com
[3] http://mpfwww.jpl.nasa.gov/default.html
[4] Knotts, Ryan, et. al, "NaviGates: A Benchmark for Indoor Navigation

system of mapping and navigation, and "will be met only when a robot functions in an unmodified office building environment, on-the-job, for a full year."[5] Similar challenges and competitions incorporating issues of navigation are offered from organizations such as the American Association for Artificial Intelligence[6].

It is fairly obvious how navigation and mapping are important issues for autonomous mobile robots. The task of creating a robust and effective system for navigation and mapping is a deceptively difficult task. There are the obvious issues that apply to all autonomous mobile robots, such as computation and memory restrictions on an on-board computer. However, the primary issues for navigation seem to be caused more by hardware deficiencies than software and computational limitations. The largest issue is that of giving the robot the ability to keep track of its location. Even with this capability, the robot will probably need some sort of reference to verify its actual location. There are many variables, many of which are out of the range of control of the robot, which could interfere with the course of a robot. Any errors or deviations in location tracking are cumulative, and could cause the actual location of the robot and the location that it believes it is at to be very different. It is obvious how inefficient and inaccurate sensors could magnify this issue. One might expect that more expensive sensors could easily solve this problem, however "some experiments have shown that using higher resolution sensors introduces more variation, not less…"[7]. While there are many viable options, there is not an ideal solution as of yet, and the topic is still being researched.

There are various options for helping a robot to keep track of its position. One of these is to allow the robot to have a map of the environment. However, this is only useful if the robot can utilize the map to recognize locations, and thus constantly verify its position. This would require some sort of landmark recognition, or a "feature-extraction" system, as in the InductoBeast at Carnegie Mellon[8]. This type of solution will not take into account dynamic factors that might be introduced. In the model of an office environment, these might include doors being opened or closed, and the presence of people or other mobile robots. Another option is to introduce some sort of intelligent system to control the robot for navigation or mapping purposes. I chose to apply artificial intelligence to the mapping portion of this issue.

[5] Knotts, Ryan, et. al, "NaviGates: A Benchmark for Indoor Navigation"
[6] http://www.aaai.org
[7] Meeden, Lisa, and Kumar, Deepak "Trends in Evolutionary Robotics"
[8] Kunz, Clayton, "Automatic Mapping of Dynamic Office Environments"

2.3 Machine Learning

Similar to robotic navigation, the application of machine learning systems to robotics is a current research topic. As hardware and sensor systems may at times perform somewhat erratically and return noisy data, applying learning systems to deal with these issues is a logical step. Swarthmore College's Carbot robot, implemented by Lisa Meeden, uses a neural network system to control its movement[9]. Input to the net is in the form of readings from light sensors, and output from the net consists of instructions to control the robots motors. A further example of machine learning applied to robotics lies within the same project. Meeden also utilizes Genetic Algorithms in the controlling system. In this case Genetic Algorithms are used to alter the net by choosing the weights that are assigned to links between nodes.

Similarly, neural networks have been applied to many other robotics systems. Meeden and Deepak Kumar of Bryn Mawr have performed numerous experiments in this field[10]. Among these are networks applied on a commercially available Khepera robot to perform such tasks as learning to recharge a simulated battery system by moving to a specific location in an environment, and performing simple trash collecting tasks. Another interesting example is NAVLAB; an autonomous vehicle of larger size than other systems examined. This system learns to use camera images to stay on a set path or road.

All of these systems incorporate machine learning into the robot controller system. Most of these also tie in issues of navigation to the learning system. These are merely a few examples of some of the machine learning systems that could be applied to robotics.

When I was determining which systems to implement in my project, I examined several different types of machine learning concepts. Many of the problems I hoped to solve relied on hardware which was not consistent in its performance. Therefore I needed a system which was robust enough to handle this sensor noise, both within the training data, and within the normal operating conditions that the robot was intended to operate in. Another consideration when I was choosing learning systems was to choose one for which I had initial code available. Many systems are complex enough that designing and implementing my own system would be very time

[9] Meeden, Lisa, "An Incremental Approach to Developing Intelligen Neural Network Controllers for Robots"
[10] Meeden, Lisa, and Kumar, Deepak, "Trends in Evolutionary Robotics"

consuming, and not necessarily what I was most interested in. Most of the systems researched were available in code in one form or another.

2.3.1 Possible Machine Learning Systems

I entertained a handful of machine learning systems as options for use in my project. One of the methods researched was Genetic Algorithms (GA's) [11]. A Genetic Algorithm system is one that relies on selection to weed out the less successful solutions, and encourage better solutions. GA's cycle through a series of generations of solutions, selecting what it determines to be "good" options at the end of each generation, by choosing from a population of possible solutions by way of a function that identifies promising characteristics. These then become likely candidates to be allowed to serve as "parents" for the next generation, thus passing on some of their traits to offspring. The process further allows for mutations to be introduced into the population, and ensures that many options will be examined before the final population is reached.

Another system I examined briefly is called AutoClass II[12]. This system is a Bayesian classification system. This type of classification involves classifying objects based on the statistical layout of the entire data set, and determining the probability of each object being included in a particular class. This system offers the advantage that objects are not placed into a classification absolutely. The statistical analysis offers the ability to examine all attributes of objects simultaneously, and does not make arbitrary assignments to classes if more than one class is represented.

The third system examined is called COBWEB[13]. This is a conceptual clustering system. Objects are classified so as to offer the best ability for inferring other information about the object based on how it is classified. This is not a pre-trained or supervised learning system, but rather an observational system. The system classifies objects based on criteria that emerge as the best descriptors of the class. This system offers the clear advantage that it is unsupervised, that is, it determines for itself the important points in a data set, and does not require a user to offer information or opinion.

[11] Congdon, Clare, "A Comparison of Genetic Algorithms and Other Machine Learning Systems on a Complex Classification Task from Common Disease Research"
[12] Cheeseman, Peter, "AutoClass: A Bayesian Classification System"
[13] Fisher, Douglas, A., "Knowledge Acquistion Via Incremental Conceptual Clustering"

The final approach examined is Neural Networks. This system takes a series of inputs in numerical form, runs it through a structure of nodes and weighted links, and then outputs one or more numbers. The neural nets that I examined are all supervised learning systems, so they require a training set to be run in order to train the network. Neural nets are very good at generalizing, such that a net trained on a representative subset of the expected information can then successfully operate with the entire data set[14]. I ultimately decided to use Neural Networks as the machine learning system in my project. This was largely due to initial success I had when experimenting with a neural net package, the time put into learning this system, and the ability of neural nets to successfully learn based on a subset of the data, and the ability to generalize through the type of "noisy" information I expected to encounter with less than perfect sensors and hardware systems on the robot.

2.3.2 Neural Networks

As with many machine learning approaches, there are many different variations of neural networks. Generally, artificial neural networks are loosely based on biological neural networks. Biological neural networks are composed of many neurons interconnected by synapses. A generic neural net has a similar structure, consisting of a grouping of nodes interconnected by weighted links. Each node takes some number of inputs, which could be sensor output or connections from other nodes, and uses these input values to create its own output value, which could then be used as the input for other nodes or the output from the network. Each node is connected to other nodes by way of weighted links that affect the value of connected nodes.

A simple structure of a neural net can be thought of mathematically as a directed acyclic graph. This architecture consists of several layers: an input layer, some number of hidden layers, and an output layer. The signal flows into the input nodes, trickles through the network, and ends up in the output layer. A simple connection structure has each node in one layer connected to each node in adjacent layers. This type of system is said to be fully connected. A traditional neural network consists of the three layers, although different nets may utilize the node and link model as necessary, with many layers and different link structures. More radical systems abandon the formal layer structuring and have much more extensive connections between nodes. All of the neural

[14] Neural Network FAQ, ftp://ftp.sas.com/neural/FAQ.html

networks I use in this project follow the standard model, consisting of an input layer, a single hidden layer with many nodes, and an output layer. This type of structure can be seen in Figure 2.3.1.

One notable addition to my networks is the existence of an extra input node as advocated by Tom Mitchell[15]. This node serves to ensure that the values of the two hidden nodes are less likely to equal zero. The value of node i_0 is always one, and the weights from i_0 are set to random values along with the other weights. Values of nodes are set by using a very simple equation. Clearly the input nodes are simply set to whatever the input to the network is, and i_0 is set to one. The values of the hidden nodes are determined by the values of the input nodes and the weights between the two layers, and values of output nodes are determined by the values of the hidden nodes and the weights between hidden nodes and output nodes. First I must establish a general notation. The values of a node will be referred to simply as the node itself, such that the value of node i_0 is simply notated as i_0. The weights between any two nodes A_1 and B_1, where A_1 is at a higher level than B_1 in the net, is notated as W(A_1, B_1). In the example offered in Figure 2.2.1, the process is easy to follow through. The values of i_0, i_1, and i_2 are set by the input values. h_0 is set to i_0*W(i_0,h_0) + i_1*W(i_1,h_0) + i_2*W(i_2,h_0). h_1 is set to i_0*W(i_0,h_1) + i_1*W(i_1,h_1) + i_2*W(i_2,h_1). o_0 then becomes h_0*W(h_0,o_0) + h_1*W(h_1,o_0). The output from the net is then available for whatever purpose it was intended for. This is a very simple and elegant process to understand, and is also not computationally difficult.

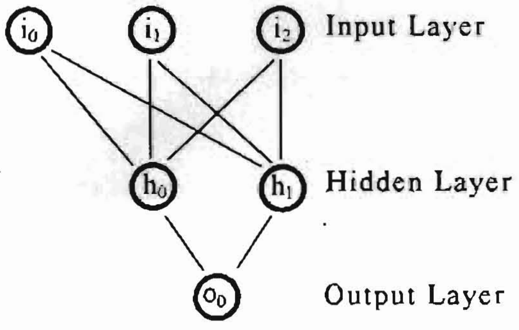


Figure 2.3.1

Beyond the architecture of a neural network, there is also the issue of how the net actually learns. The neural nets that I use learn by example. The type of process that I use is called a supervised learning system. My networks must be given some training data on which to base its internal structural adjustments. How these adjustments are made is the interesting part about neural nets. In general, a net learns by adjusting the weights between nodes, by either incrementing or decrementing their value. In this type of net, the correct outcome is known for some subset of the

[15] Mitchell, Tom. "Machine Learning"

data that is expected to be run through the net. This is known as the training data. On a high level, the net takes each case of the training set and runs it through the nodal structure. The output is then examined and compared to the desired output. The structure of the net is then changed by a correction process.

The correction process that I use is called backpropagation. This is known as a feedback system, as the system examines the output, then backtracks up through the net, correcting the weights of links appropriately. When the training data is run through the network, the output values are compared to the desired output, based on the training set. The backpropagation algorithm then works back up through each node and link, comparing the value of a node to the valuc that it should have been to determine the error. This is done for the hidden and output nodes, and then the algorithm adjusts the weights of the links connecting the nodes. After training the network, there may be the opportunity to test the network on data that was not included in the original training set, depending on the nature of the data being used. For instance, in a net that is trained to recognize a function such as XOR, it is not possible to test the network with data that is not included in the training set. However, in a network which has learned to recognize a pattern or a more general function, the network can be tested with data that was not part of the training set to test the generalizing capabilities and success of the net.

A strength of neural networks lies in their ability to generalize to the desired function. That is, a network can learn a function that is present in the training data and successfully apply it to data that has never been encountered before. The type of data and function being represented by the network will have some effect on the networks' ability to generalize well. In general, the one important restriction is that the training data must actually represent what the network is supposed to learn. If the training data includes some sets that are on the extreme edges of the average input data, the network will not perform as well. The flip side of this is that a well-trained network will perform very well on abnormal data after the training stage is completed, and can in fact include some abnormal examples in the training set. This is of particular importance to robotics applications, as sensors frequently return noisy readings. Under these conditions, a robot could receive a strange sensor reading, and still perform the proper response to the situation.

To learn more about neural nets, I used some examples of code. The first was a network that learned to recognize the exclusive OR function[16]. The exclusive OR function, or XOR, is a

[16] code from Patrick Ko Shu-pui.

bitwise operation that takes two binary input values. The function is satisfied if one but not both of the input values is one. So, the four possible scenarios are as follows: 0, 0 → 0; 0, 1 → 1; 1, 0 → 1; 1, 1 → 0. The next coded example of a neural net that I examined was designed to perform face recognition in simple images[17]. This neural network package contained a great deal of code that was specific to the problem of face recognition. Most of this was not essential to the neural net itself, and could be removed. I used this code base to create another example of a net to perform the XOR function, based on the operation of the first net used. The XOR function is an interesting example to use. The entire data set must be used as the training set, as there is no way to generalize this function. This is due to the fact that the XOR function is non-linear in nature. While there is not way to test the generalization capabilities of the net using this function, it is a very good illustration of the capabilities of a neural net, as it is difficult to learn a non-linear function. Once this task was completed, I had a working neural net structure that I could apply to other problems.

Once I had an XOR function working on the Linux computer, the next step was to move this code over to the Handy Board and run it there. There were a number of changes that needed to be made to the code in order to compile and run it through Interactive C, due to some limitations of Interactive C. Once these changes had been made, I began to run neural networks on the Handy Board. I quickly determined that running any sort of complex or large net on the Handy Board would be extremely time consuming, due to the memory and CPU limitations of the Handy Board.

[17] code from Tom Mitchell

CHAPTER 3
HARDWARE

3.1 Introduction

I put much time and thought into the design and construction of a sturdy robot. A reliable base that is not prone to breakdown or erratic behavior is desirable for both practical and research applications. Likewise the sensors and attachments to the robot base must also be consistent in performance. While there are commercially available bases that are well designed, these units are often too costly for a research project such as this. I chose instead to design and build a homemade base. This yielded complete control over mounting custom sensors, in addition to a relatively low cost. This also allowed me to experience building a robot, which was a challenging and educational experience. Along a similar line, I used commercially available parts to build sensors rather than purchasing more expensive prefabricated sensors. There were many issues and problems that I encountered during this process.

3.2.1 The Robot Base

My preliminary robot research entailed using Lego pieces designed for robot experimentation. Using Legos offers several advantages over other materials. Primarily, Legos are reusable, whereas materials such as wood or metal are often more permanent. This makes Legos an excellent option for prototyping robots, and even better for an initial introduction to robotics in general. While there are some restrictions based on limitations of the Lego pieces, such as the rectangular nature of most pieces, and the inflexibility of Lego pieces, in general they are a good tool, as well as fun to play with.

After experimenting with the Legos however, I determined that a Lego base would not be appropriate for long-term use. The ability of Lego pieces to be disconnected and reattached also means that they are more likely to break apart, and therefore are not able to provide a sturdy base. Options for the material of the base included plywood, Plexiglas, and a combination consisting of Legos glued to either plywood or Plexiglas. After my experiences with gluing Lego components to other materials I quickly dismissed this option, as the glue would typically be knocked loose during normal operation of the robot. Plywood was eventually chosen due to availability.

The robot base is a 9 3/8" diameter circle of 1/4" plywood. Using 1/4" plywood instead of a thicker grade offers the benefit of easy attachment of hardware, as there is a relatively small thickness of material to drill through. In addition to the 1/4" plywood being relatively lightweight, being able to use shorter screws to attach hardware also helps to keep the overall weight of the unit down. Centered on the centerline of the circle and located symmetrically across from each other are two oval holes cut out of the wood on the outside edge of the circle. These holes are designed so that the drive wheels do not stick out beyond the edge of the circular base, and so that the wheels will stick up through them, thus decreasing the overall height of the robot (Figure 3.2.1). Keeping the height of the robot down is important as it helps to keep the weight of the robot closer to the ground. This unit does not have a wide wheel base, and so cannot support a top-heavy design.

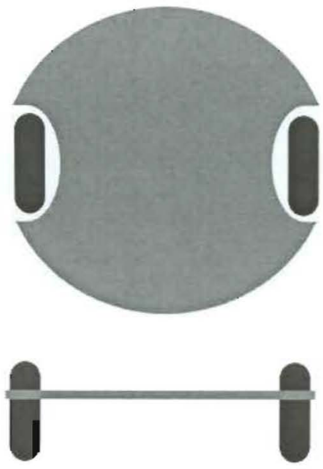


Figure 3.2.1 – Robot base showing placement of wheels

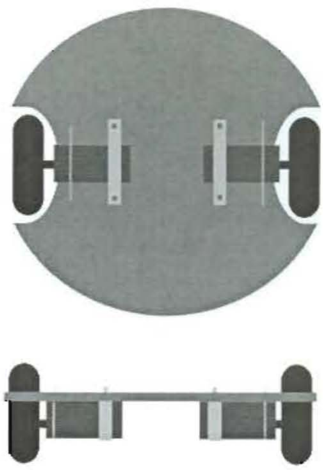


Figure 3.2.2 – Robot base showing placement of motors

The motors are mounted on the underside of the robot base. The motors are aligned symmetrically along the centerline of the robot base. As the drive shaft leaving the gearbox is not centered, the larger part of the motor unit is facing upward toward the robot base. This gives the robot a higher ground clearance, and creates more space underneath the robot for sensors to be mounted (Figure 3.2.2). The two motors are attached with a strip of brass. The strip is placed across the motor, and then screwed upwards into the base. The holes in the brass strip needed to be pre-drilled, as it was not possible to start a hole with just a screw due to the thickness of the brass strip. At the inside end of the motor unit, a pair of thin brass wires is placed over the motor. These wires are doubled against each other, and then go up through the robot base, where they are twisted together on the top of the base (Figure 3.2.2). This was done to prevent the motors from loosening in the brass strips. I found that when the robot would begin to move, the effort expended to overcome the initial friction of the wheels was enough to move the motors. This was serving to loosen the brass strips that bind the motors to the base in such a way that the motors were no longer mounted symmetrically and straight. As the motors cannot touch anything metal, there is insulation between the motors and the brass wires and strips. I used tubes of plumber's foam insulation. Meant to be slipped over a hot water pipe, these tubes were easily cut to size to be placed against the motors. This is not an ideal situation, as the foam will eventually compress enough to loosen the motors' attachments to the base. However, occasionally adding more foam and then retightening the brass pieces serves to fill in the gaps caused by compression.

As the two motors are positioned in the middle of the robot, and as the ground clearance of the robot is so high, it is very important to have some method of stabilizing the base so that it does not tip forward or back. My original design called for a castor wheel on each side of the base, located at the edges of the base, ninety degrees off from the drive wheels. After some experimentation, I discovered that trying to push a castor wheel in front of the robot was too difficult, due to the amount of force required to swivel the castor wheel. The castor would often not swivel, which would cause the robot to either not turn when it was supposed to, or sometimes to not even move at all. This was solved by removing the front castor wheel, leaving only the trailing castor wheel. This solved the movement problems, and did not create any noticeable balance problems. To be sure that the robot would not tip over, I mounted the Handy-Board over the rear castor wheel, as added insurance. As the castor is not as tall as the ground clearance of the robot, I needed to add a spacer between them when I mounted the castor wheel. To do this I took a piece of 3/4" plywood and a 1/4" piece of scrapwood. The castor wheel has four holes for mounting the unit. I used two of these, opposite each other diagonally. This is still a very strong mounting system, and serves to save some weight by dropping out two screws. The castor screws into the plywood, and then into the base of the robot, with the scrapwood as a spacer. The spacer was then removed as the screws held the

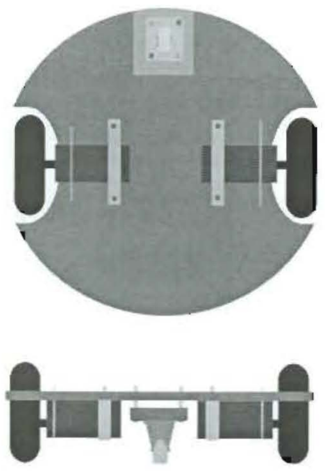


Figure 3.2.3 – Robot base with castor wheel mounted

castor attached very securely even without the spacer being there (Figure 3.2.3).

3.2.2 Shaft Encoder

In order for the robot to keep track of its location in the environment, it must have some mechanism for recording how far it has moved. This requires some form of odometer. In my previous independent studies in robotics, I attempted to create an odometer from Lego components. This proved to be troublesome on many different levels.

My initial Lego design was a physical shaft encoder, where a touch sensor would be tripped with every rotation of one of the drive wheels. This was difficult to maintain, prone to breaking frequently, and required constant supervision to ensure that the encoder did not exert too much pressure on the drive wheel and prevent motion of the robot. My next design involved a spinning disk attached to the drive system of the robot. There was a light on one side of the disk, and a light sensor on the other side. The disk contained three holes, such that the light sensor would only detect the light when the disk was positioned so that light could travel through it (Figure 3.2.4). This was more successful, but was not very accurate, as the robot traveled a significant distance before the odometer would increment. This system was also susceptible to missing light and not incrementing the distance traveled when it should have.

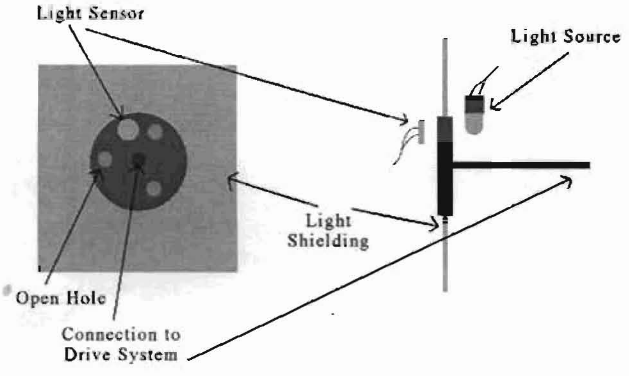


Figure 3.2.4 - Light Based Lego Shaft Encoder

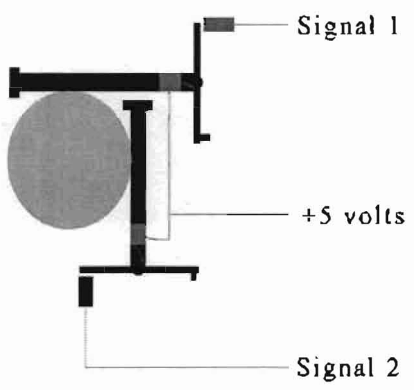


Figure 3.2.5 - Schematic of Mouse Quadrature Shaft Encoder Design

My next design involved a modified computer mouse. This design used the existing small-scale quadrature encoding system of a mouse, and incorporated it into my robot. Most mice use a series of break-beam infrared sensors to encode movement. I had initial difficulties in interacting with the existing infrared sensors, so I opted for a physical system. This involved attaching conductive strips of metal to the quadrature system, and writing code that would increment a variable whenever the mouse ball made a complete revolution. This offered the capability to keep track of four directions of travel, but also was limited by the physical nature of the system, which was prone to breaking, and difficult to repair due to the small scale and fragility of the components.

Given these options and my experience with them, I opted to purchase a commercially available shaft encoder. The unit I chose is the S1 model, made by US Digital. Information about this unit is available in Appendices A and B.

The space in between the two drive motors was intentionally made large enough to house the shaft encoder. The encoder is mounted in the center of the robot base, so that the only movement recorded is that of the robot moving forward or backward, and not when the robot performs a rotating turn to the left or right (Figure 3.2.6). The shaft encoder has a sturdy brass wire wrapped once around the shaft, close to the housing of the encoder, secured with a nut screwed down tightly against the encoder housing. This forms a strong yet flexible system for holding the shaft encoder in place. The wire comes up on either side of the shaft, roughly forming a ninety-degree angle. Each end of the wire then goes up through the robot base, where each is bent over towards the closest outside edge of the base. The wire ends are then secured with additional wires, so that they are held down tightly, but can still be easily removed or adjusted. The wires were adjusted until the shaft encoder was positioned firmly against the floor. The securing wires are designed to have enough spring in them so that the shaft encoder wheel can deal with a slightly uneven floor surface without missing revolutions of the drive wheels. Not only can the wires move up and down, they also allow the shaft encoder assembly to move side to side. I anticipated that this would not be utilized during normal operation, but it decreases the chances of the assembly becoming broken if the robot is handled roughly or bumped.

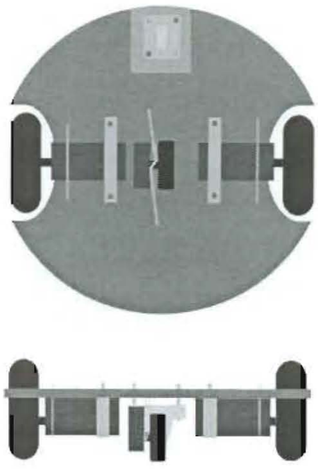


Figure 3.2.6 – Robot base with mounted Shaft Encoder

3.2.3 Light Sensors

The robot must always return to a known home coordinate. This is necessary as the robot needs to report its findings about the world to the Linux computer which is located at this home coordinate. It is also necessary because the robot may need to reorient itself in the environment should it become lost or get off track. Should the robot become lost, it will no longer be able to rely on its interpretation of the coordinate system to find its way back to the home coordinate. Therefore it becomes necessary to have some sort of backup system for locating and reaching the home coordinate. My solution to this was to incorporate a light source at the home coordinate and a series of light sensors on the robot.

The three light sensors are mounted on top of the robot, facing forward. The sensors are mounted on a small piece of wood, which serves to raise them above other sensors mounted on the front of the robot. The sensors are attached by gluing them to the piece of wood on which they are mounted. The sensors are located in a line, with the middle sensor facing directly forward, and the two sensors to its left and right facing forty five degrees away from the center sensor. My original robot design called for the light sensors to be mounted on the rear of the robot, facing behind the robot. This would allow the robot to back into position at the home coordinate, and be facing the correct direction when it came time to execute the next path. However, the previously mentioned problems with the castor wheel prevent this from being an option, as the robot would have to drive the remaining castor wheel ahead of it if it were to move in reverse. I decided to mount the light sensors on the front of the robot, and have the robot perform a simple one hundred eighty-degree turn once it was in position at the home coordinate. This allowed for accurate light tracking and maneuverability. This system was sometimes not sufficient to realign the robot in the environment, and occasionally required some interaction on my part to ensure that the robot was reoriented properly. Due to hardware problems I was unable to fine-tune this system to operate successfully without user interaction.

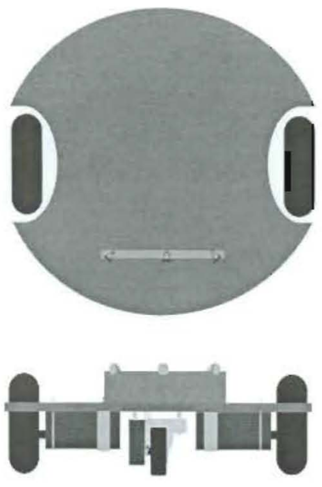


Figure 3.2.7 – Robot base with light sensors

3.2.4 Touch Sensors

In order for the robot to know when it bumps into an obstacle, there must be some sort of sensor that interacts with the environment. Due to the size of the robot, this sensor system must be large enough to encompass the entire leading edge of the robot, yet must be composed of enough individual sensors so that the location at which the robot hits an object can be determined with some degree of accuracy.

The natural solution was to utilize simple touch sensors. These sensors are homemade and customized for my robot. There are four sensors mounted on the front half of the robot. The touch sensors consist of two brass strips nailed into the edge of the plywood base. The two strips are on the left and right sides of the leading edge of the robot. Suspended over these are four brass wires. These springy wires are attached in the top of the robot base by way of drilled holes and glue. Each wire is bent so that it hangs in front of the brass strip on the front of the robot base (Figure 3.2.8). In this way the brass wire can touch the brass strip when it encounters an object. This completes the circuit, and the robot then knows there is a solid object in its way.

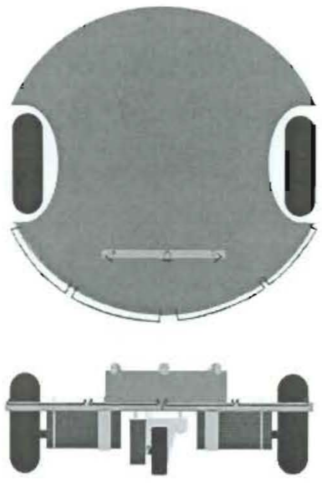


Figure 3.2.8 – Robot base with touch sensors

The decision to place the light sensors on the front of the robot reduces the number of touch sensors that are necessary on the robot. With the light sensors located on the front of the robot, the robot does not need to travel in reverse. This eliminates the necessity of having touch sensors on the rear edge of the robot.

3.2.5 Electronic Compass

While the shaft encoder allows the robot to keep track of how far it has moved, this information is useless without the knowledge of which direction the motion of travel was in. The robot is not reliable enough on its own to be able to move in a straight line at all times. Therefore some other sort of check of the direction of the robot is necessary. My solution for this problem was to purchase and integrate an electronic vector compass.

The wiring of the compass is complex, and the unit must be positioned carefully so as to avoid bending wires and jeopardizing the integrity of soldered connections. There were also several other issues that influenced the positioning of the compass on my robot. I wanted the compass to be centered in the middle of the robot, so that errors due to the relative positioning of the robot as compared to the compass would be minimized. The compass itself is mounted on a perf-board with mounting holes at the four corners, both for ease and safety of wiring, easier repairs to the wiring, and for the ability to mount the compass onto the robot without having to actually attach the compass unit itself.

Another major issue that impacted the location of the compass was that of magnetic fields. Magnetic fields have a large impact on the successful operation of the compass, so the unit needed to be mounted away from sources of magnetic fields, such as the motors, the Handy Board, and to some extent perhaps the unshielded wires connecting the Handy Board to hardware components. While the compass has a built in calibration system to compensate for the effects of magnetic interference, this system is run when the compass is first initiated, which occurs only once during operation. Therefore any magnetic sources with fluctuating strengths, such as the motors, could conceivably have an effect on the operation of the compass. To position the compass as far away as possible from these sources, I mounted the compass onto a 5" long piece of wood, and positioned it above the robot base. As it was also necessary to have the compass located in the center of the robot, the compass is fastened to the wood on only one side, such that it projects forward over the approximate center of the robot. Also, in order to ensure proper operation of the compass, the unit needs to be held completely level at all times. To accomplish this, I made sure that the wood on which the compass is mounted is held upright securely. This wood is not permanently fastened to the robot base as I need to access the compass on a regular basis, and must be able to detach and reattach the unit easily.

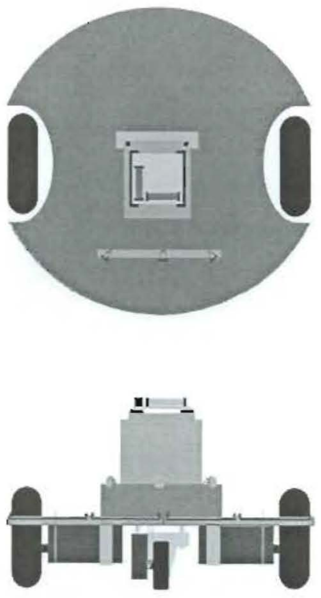


Figure 3.2.9 – Robot base with electronic compass

3.2.6 Infrared Ranging Sensor

While the touch sensors are sufficient for navigating and successfully moving about the unknown environment, using only these touch sensors for mapping the entire environment would not be the most efficient method. Using only touch sensors would require the robot to attempt to physically travel to every coordinate on the map. This would require a large amount of time, as the test environment that I created was ten feet by ten feet in size. In previous studies I had used inexpensive infrared sensors for tasks such as obstacle detection. So for this project I purchased and integrated a slightly more expensive infrared ranging sensor. This allows the robot to detect objects from a distance, and perform sensor sweeps that create a picture of the objects surrounding the robot within the radius defined by the maximum range of the sensor.

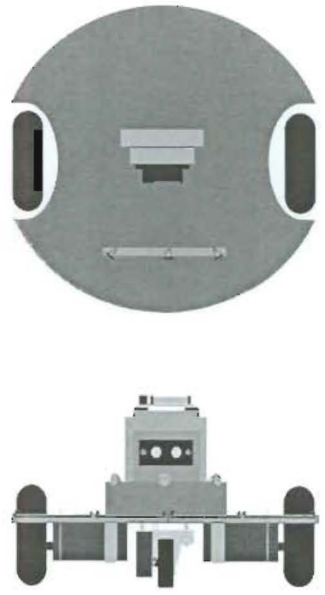


Figure 3.2.10 – Robot base with Infrared Ranging Sensor

The infrared sensor, like the compass, is mounted onto a piece of wood, approximately three inches high. The sensor is mounted facing forward, and is mounted above the top surface of the robot base so as to eliminate interference from obstructions such as the other sensors mounted on the front of the robot. This sensor has both an effective maximum range of 80 cm (31 1/2"), as well as an effective minimum range of 10 cm (3 9/10"). Therefore, while this sensor does not need to be located at the center of the robot, it does need to be set back from the leading edge of the robot base to ensure that the sensor does not report back erroneous readings that lie outside of its effective range. So the sensor is located approximately four inches back from the front of robot. This eliminates the possibility of an obstacle being between the sensor and its minimum range. The sensor is screwed into the mounted wood by way of two mounting holes in the infrared sensor's housing (see Figure 3.2.10).

3.2.7 – The Handy Board

The biggest component to be mounted onto the robot base was the on-board computer. The computer used was a Handy Board. The Handy Board is a small three MHz computer, equipped with thirty-two kilobytes of RAM. More complete information and specifications of the Handy Board can be found in Appendix B.

The Handy-Board needs to be attached to the top of the robot, positioned so that all sensors can be plugged in without straining the connections. The board must also be fastened securely enough so that it would not be in danger of falling off or being damaged. The way I chose to do this was to have the board held in place by a series of nails hammered into the surface of the base (Figure 3.2.11). The board lies within these nails, and is prevented from sliding or moving around while allowing it to be accessed and removed easily. The board is mounted at the rear of the robot, just forward of the castor wheel. In this way the relatively heavy board serves as a counter balance to ensure that the robot does not tip forward, and also keeps the board out of the way of the forward mounted sensors. Enough of the weight is focused on the center of the robot that there is not too much weight placed on the castor wheel, which would prevent the wheel from turning freely.

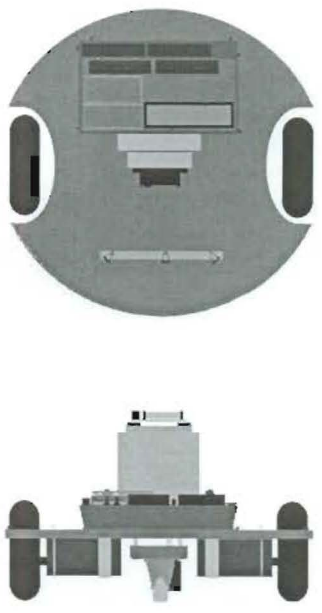


Figure 3.2.11 – Robot base with Handy Board

3.3.1 Constructing Sensors

All of the sensors used, including those that were purchased commercially, required some degree of construction and wiring to be performed. Information was sometimes not immediately available for the wiring schematics for these sensors. Every component of the robot that required wiring to be done had some common issues.

For all wiring and connections I used flexible stranded ribbon cable. In previous robotics projects I had used solid copper wire. This proved to be too stiff due to its thickness and strength. In addition, soldered connections made with the thick copper wire were more likely to break under normal usage, especially if this included any sort of rough handling of the robot. The ribbon cable is very flexible, and comes connected in strips, so that the desired number of wires can be torn off as necessary. Also, the wire inside the plastic insulation consists of strands of small diameter wires. Soldered connections made with these wires are more likely to retain conductivity if mishandled, as there are many more connections present than with a single copper wire. One drawback to using these stranded wires is that if a connection does become loose, it is very likely that one or more strands may touch other connections, creating contact points where no contact is desired. This has the potential to damage the Handy Board or sensor equipment. The Handy Board is equipped with detection routines that shut the board off in the event of overloaded circuit, however it is still possible to damage expensive sensors. I used two solutions to overcome this. First, I used shrink tubing whenever possible. This entailed putting unshrunk tubing on the wires before soldering a connection, and then shrinking this tubing after soldering. The tubing is shrunk by a heat gun, which operates at a relatively high temperature (approximately seven hundred degrees Fahrenheit). As this temperature is often above the safety threshold of many electronics components, it was often not possible to use the shrink tubing due to the proximity of sensitive electronics. The second solution was to pre-treat the stranded wire with solder. This involved twisting the strands of wire together, heating them with a soldering iron, and allowing a small amount of solder to be drawn into the strands. This helps to hold the wire together both during the process of connecting the wire to an electronic component as well as after the connection is soldered. This also makes the soldering process easier by already having solder present when the wire and electronics component are heated to make the final connection.

Most sensor connections to the Handy-Board are made with male strip header, which is composed of a series of metal posts embedded in a plastic holder. Wires are soldered to the top part of the post, and the bottom part of the post plugs into the corresponding female strip socket on the Handy Board. Both male posts and female sockets are spaced at a regular and standard distance from each other. The Handy Board has nine digital input ports, and seven analog sensor input ports. The Handy Board also contains other options for sensor input, and the Expansion Board increases these options even more. More information about the Handy Board and Expansion Board can be found in Appendix B.

3.3.2 – Constructing Sensors: The Expansion Board

The Expansion Board is a recent addition to the world of small robots. The Expansion Board is designed to plug into the Handy Board, and offers several more options for sensor input and output. While the Handy Board was pre-assembled, the Expansion Board came in the form of a kit, with no instructions or directions for constructing it. While the Expansion Board (Figure

3.1.1) does not contain any very complicated wiring, it does consist of some sensitive electronic components. I constructed the Expansion Board by examining pictures of a completed Expansion

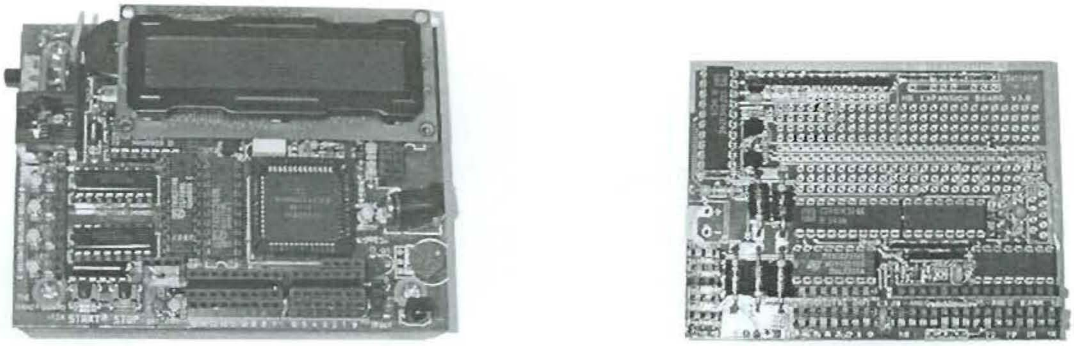


Figure 3.3.1 – Handy Board and Expansion Board

Board found on-line[18]. While this was ultimately successful, it is not the ideal method, as I had no real way to test every component of the Expansion Board to determine if everything was wired correctly, so I simply tested the simple input ports that I could experiment with.

3.3.2 Constructing Sensors: Shaft Encoder

Similar to the Expansion Board, the shaft encoder (Figure 3.3.2) did not arrive with any directions or instructions for construction. However, the pins for the sensor were very clearly labeled, and it was a simple task to determine the proper wiring for the unit. As the unit had posts that fit into the strip socket that I used, I did not solder connecting wires directly to the sensor, but used a piece of female strip socket as an interface to the sensor. Whenever possible I attempted to avoid soldering wires directly to sensors, as I did not have temperature control of the soldering iron, and heat can have a devastating effect on electronic components. The connection to the Handy Board is a simple one. The shaft encoder takes only two analog input ports to connect to the Handy Board, and only one of these is actually necessary. This sensor was very simple to wire and implement.

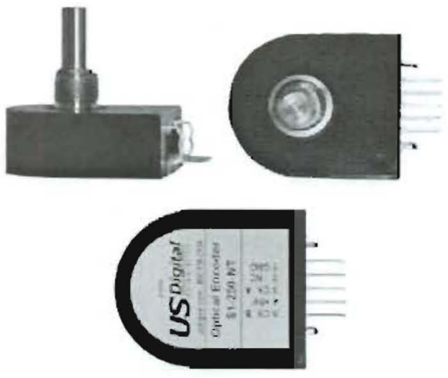


Figure 3.3.2 – Shaft Encoder

The only other issue remaining with this sensor was that of interacting the sensor with the environment. As I was not going to connect the shaft encoder to the drive system in any way, I

[18] http://el.www.media.mit.edu/groups/el/Projects/handy-board/hbexp30/

needed to have some way of turning the shaft on the encoder. My solution was to attach a wheel to the shaft. I used a Lego wheel of relatively small diameter. I needed to make the hole in the wheel a bit bigger to accept the 1/4" shaft from the encoder housing, and used a drill to do this. As I had neither the 1/4" drill bit nor a drill press to drill directly down into the wheel, I used a small drill bit. I held the drill perpendicular to the flat side of the wheel, and circled it around a number of times, slowly stripping plastic out until I had an even, larger hole to accept the shaft of the encoder.

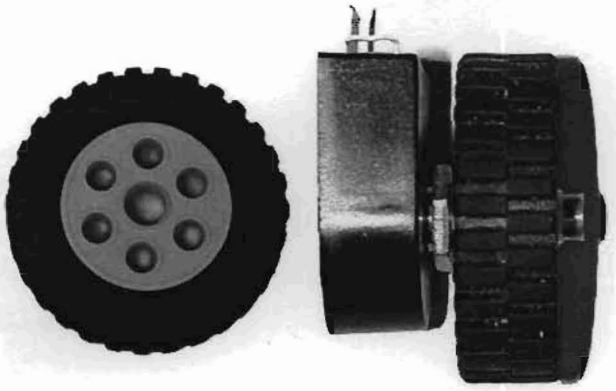


Figure 3.3.3 - Shaft Encoder with Lego Wheel

3.3.3 Constructing Sensors: Light Sensors

The light sensors are some of the easiest sensors to wire and implement. The light sensors are composed of a simple photoresistive cell. There are only two connections to be made for these sensors to work. One of the wires of the sensor goes to the signal port, and the other goes to the ground port of an analog input on the Handy Board. Electricity comes into the resistive cell, and the amount of light present determines how much of the electricity is allowed to continue through back to the Handy Board. These sensors are fairly standardized, and there is little variation in performance between them. Ordinarily I would shield the soldered connections with shrink tubing, however, for the light sensors, it was easier to wrap the connections in electrical tape.

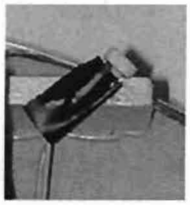


Figure 3.3.4 - Light Sensor mounted on Robot

3.3.4 Constructing Sensors: Touch Sensors

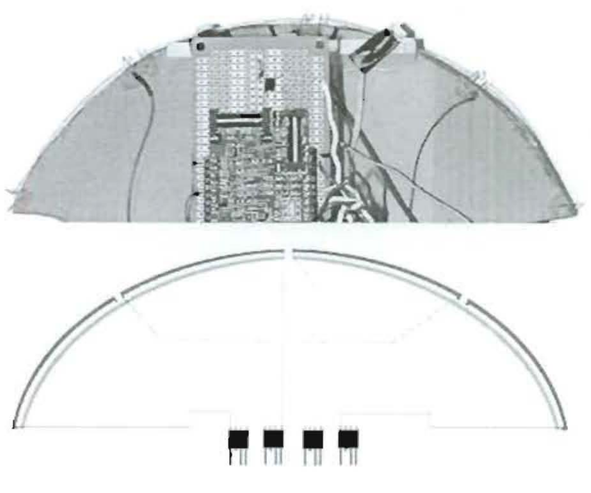


Figure 3.3.5 - Robot Touch Sensors

Similarly to the light sensors, the touch sensors were also very easy to wire. The touch sensors are a very simple design, which relies on a circuit being either opened or closed. When the circuit is completed, the touch sensor returns a signal that it has been triggered. I implemented my touch sensors such that there are two sources of power for four touch sensors, as is shown in Figure 3.3.5.

3.3.5 Constructing Sensors: Electronic Compass

The most difficult sensor to wire and implement was the vector compass. While this sensor did come with documentation and schematics, the information given was at an advanced level that was not of much help to me or my peers. Eventually, through several sources, enough information was gathered so that I could create a wiring schematic that would allow the compass to function.

There were several issues that went along with wiring the electronic compass. The first was my reluctance to solder wires directly to the pins on the compass itself. The heat involved would certainly have damaged the sensitive electronic components on the compass. This called for mounting the compass on a small piece of perf-board. I initially used the same strip socket that I used for all other sensors for the compass to plug into. However, as the pins on the compass are flat and the strip socket is designed to accept square pins, this did not guarantee a constant connection. Until I was able to locate and integrate sockets designed for flat pins, I decreased the size of the square holes by inserting a piece of brass wire into each hole to secure the connection between the pin and the metal connectors inside the socket. This was no longer a problem once I rewired the system with sockets designed for flat pins.

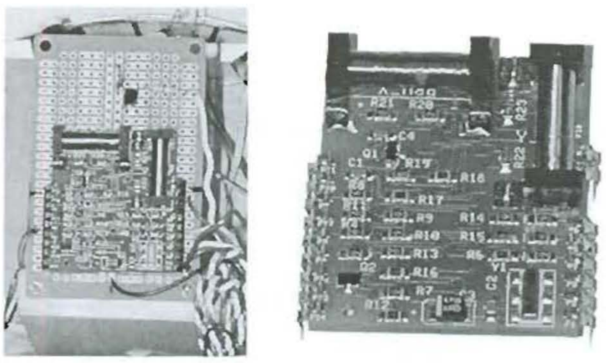


Figure 3.3.6 - Electronic compass

The next major issue in the wiring was that of the connection to the Handy Board. The compass connects to a synchronous serial port for communications, which on a Motorola system such as the Handy Board is the SPI port. When the Expansion Board connects to the Handy Board, it plugs into the available SPI port. While the connections continue through to the Expansion Board, there are no sockets on the Expansion Board to utilize the SPI port, and it is a very difficult task to connect female strip socket to the top of the Expansion Board, as this would involve attaching the posts of the strip socket to the tops of the posts sticking up through the Expansion Board. Due to this, connections were made directly onto the tops of the posts that are used to connect the Expansion Board to the Handy Board. This is a difficult task, and as the space involved is extremely limited, it is easy to have unwanted connections. I would frequently have to check the connections using a voltmeter to check for conductivity. Stray solder, loose wires, and loose pieces of metal would often be culprits in bad or unwanted connections.

Upon examining the schematic of the Expansion Board, I found it to be possible to access all pins of the SPI port through other locations on the Expansion Board. Some of these locations were no more accessible than the original ones, but using a combination of these pins made it easier to wire the connections, and easier to track down and repair problems.

Another issue with wiring for the compass was that a wire needed to be connected to a pin on a chip on the Handy Board. Not wanting to solder directly to a chip, I wrapped the wire around the pin and secured the connection that way. While this seemed to work, there were too many ways for the connection to fail. Upon examining the schematic of the Expansion Board, another connection was found to be possible, and the wire was soldered directly to the Handy Board this way. More information is available in Appendix B.

3.3.6 Constructing Sensors: Infrared Ranging Sensor

The infrared ranging sensor was very easy to wire. This unit came both with instructions and additional parts, such as wires, a pre-made socket to plug into the sensor, and a transistor necessary for use with the Handy Board (Figure 3.3.7). The infrared sensor plugs into a digital input port on the Handy Board, but also requires one of the digital output ports on the Expansion Board. This particular sensor is an active sensor, in that it emits a signal in the infrared

Figure 3.3.7 – Infrared Ranging Sensor

wavelength, waits for a return signal to bounce off of some object, and then determines the distance based on the time it took to receive the reflected signal.

CHAPTER 4
SOFTWARE

4.1 Introduction

I put a tremendous amount of time and planning into the software that would make this entire system work. There are a large number of components that work together to form the bigger mapping and navigation system. Each component is in turn comprised of still smaller parts. I view the software in two groups: the software on the robot, and the software running on my Linux system. This is simply a way of breaking up the code to make it easier to examine. The two groups of software cannot perform independently of each other, as they rely on components of each other in order to produce any useful results. For the purposes of this discussion the software will be broken up into subsystems, so as to simplify the task of examining the systems and the design decisions behind them.

4.2 Robot

4.2.1 Robot Software – Communications

Periodically throughout the operation of this system, the Linux system and the Handy Board need to communicate. Communication with these two systems requires a serial link between them. Both computer systems check the serial line for communications. Communication over this serial link requires a non-trivial amount of time, and also requires periodic checks to ensure that data is actually being received on the other end of the communications link. Another issue with this system is that communication from the Handy Board to the Linux computer takes a significantly smaller amount of time than communications going in the other direction, probably due to the significantly faster processor speed of the PC.

There were a couple of ways to establish this serial communications link. The first possibility was to use the infrared transmitter and receiver located on the Handy Board in conjunction with a similar hardware system built for a PC. The second option was to use the already established telephone wire interface between the Handy Board and an RS232 serial port on a PC. As I would have had to create the Linux component of the infrared system, I opted for the

direct physical connection. This meant that whenever the robot and Linux machine wanted to communicate, I would have to connect and disconnect them at the proper times.
Using publicly available code for the Handy Board and a modification of code[19] for a Linux system I created a reliable system of communication that caters to the needs of this project. This system can send integers over the serial line. My solution includes a way to acknowledge messages sent between the two computers. This is necessary as the communication line is not infallible, and as the line is broken and reattached periodically, which could easily lead to miscommunication.

The communications requirements of this system consist of transmitting paths of travel and a representation of the map from the Linux machine to the Handy Board, and returning the results of the data collection trip from the robot to the Linux machine. Due to the large amount of data that must pass between the robot and the Linux machine, I opted not to acknowledge every transmission of data between the two computers. This meant that more data was sent in between acknowledgments, so more data would have to be resent if a transmission failed. When sending communications from the Linux system to the Handy Board, I discovered that I needed to introduce a delay between each transmission, as the Handy Board was unable to receive information at the rate that the Linux system was sending it.

The code required for the Handy Board to transmit and receive data over the serial line is available from the code repository on the Handy Board web site[20]. All of the necessary methods are provided. In general, the serial link to a controlling compiler such as Interactive C must be overridden, by disabling the pcode, the low-level interface on the Handy Board. Transmissions can then be handled by transmitting a character at a time over the serial link. I used a function that would loop through the digits in integers larger than one digit in order to speed the process and decrease the code that needed to be written. Similarly, receiving information on the Handy Board is handled by taking a character at a time off of the serial line. It is important to note that all items sent through the serial line are characters, represented by ascii numbers, and not actual integers. This fact can easily go unnoticed. All characters that are meant to be integers must be converted from characters to integers. It is also important to note that a controlling program and compiler, such as Interactive C, must be shut down or disconnected from the serial line before attempting communication between the Handy Board and Linux machine. The first system to take control of

[19] Thomas Heidel – theidel@advis.de
[20] http://el.www.media.mit.edu/groups/el/projects/handy-board/software/contrib/drushel/serialio.c

the serial line has control until it releases it. Should either the Handy Board or Interactive C attempt to send signals to the wrong system, either system could easily misinterpret characters sent over the serial line, and exhibit unexpected behavior.

4.2.2 Robot Software – Interacting With Sensors

All of the software necessary to interact with and utilize the sensors used on the robot is available from various locations on the Handy Board web site. This includes both the assembly code necessary to interface the hardware systems together as well as the code to activate and get data from the sensors.

The first sensor that I implemented was the shaft encoder. The assembly code[21] for the shaft encoder is available with a couple of options, namely the speed at which the encoder operates, and which input port the user desires the sensor to be connected to. The speeds available are fast and slow. I experimented with both and determined that the fast speed was the most accurate and appropriate for my robot. The versions of the assembly code for different input ports are included as the assembly code must explicitly specify which port to access in order to increment the counter variable. I arbitrarily chose the encoder to be connected to input port six. The user has the capability to set the thresholds at which the total count from the encoder will increment. The user can also access and reset a variable representing the number of times the encoder has incremented, and access a variable representing the current velocity of the encoder. These variables are integers, and thus are limited in size.

The next sensor I implemented is the infrared ranging sensor. This code[22] provides the necessary subroutines and interfaces to control the IR sensor. The user must first call a function to enable the sensor before using it. Similarly, when use is completed, or if the user wishes to free up processor cycles being used by the process controlling the IR sensor, there is also a disable function available. Getting the current sensor reading is done by accessing a variable that contains the most recent reading from the sensor.

The final sensor that required special software is the vector compass. The code[23] for this sensor again provides all necessary subroutines and interfaces to control the compass. The compass software must also be enabled, and can likewise be disabled. The current heading is

[21] http://el.www.media.mit.edu/groups/el/projects/handy-board/software/encoders.html
[22] http://reality.sgi.com/barry_detroit/GP2D02_1.html (linked from Handy Board site)

stored in an integer variable, and can be accessed at any time. During proper operation of this particular implementation of the compass, the reading should always be between zero and three hundred fifty nine, signifying the current compass heading.

The code for all three of these sensors is somewhat taxing on the processor. Each software system is constantly updating and interacting with the sensor, which chews up time and processing capabilities that affect the other sensors as well as other computations being performed. The ultimate effects of this are discussed in following sections.

4.2.3 Robot Software – Measuring the World

Having the robot interact with the environment created some issues and problems that needed attention. In my representation of the coordinate system, I split up the world into a grid of one inch squares. The most obvious problem was that the Linux software and Handy Board to this point have dealt with paths of travel and locations as if the robot were one grid square in size, and haven't compensated for the fact that the robot is significantly larger than this. So the first problem was to interface the robot to the world by putting grid squares in some sort of unit that was useful to the robot. As the robot measures distance with the shaft encoder, it made sense to determine the size of a grid square in terms of clicks on the odometer, and I established the number of odometer clicks per grid square by performing experiments. These experiments included measuring certain distances, running the robot over these distances, and then dividing the number of clicks of the odometer by the number of inches that the robot had moved. I did this for various lengths, and at varying speeds of travel. This seemed to work well and consistently, and I found that a grid square was about equal to two hundred clicks of the shaft encoder. However, once I began running the fully implemented software package for the robot that I had written, this was no longer true. It seems that once I enabled the infrared sensor and the compass, and had my own code running constantly, enough cycles of the processor were taken away to significantly decrease the number of encoder clicks that covered an inch in distance. I repeated the experiments with all of the software running, and found that an inch was then covered in one hundred clicks of the encoder. It is difficult to know if this number will now be consistent or not, given more or less computationally intense periods on the Handy Board, and varying power levels as the robot is run more and more. This is a significant problem, and one which is difficult to solve due to uncontrollable variables.

[23] http://el.www.media.mit.edu/groups/el/projects/handy-board/software/contrib/tomb

Future systems would need to compensate for this, perhaps by running each sensor in its own thread, and ensuring the consistency of sensors such as the shaft encoder.

Due to the size of the robot, it covers just over nine grid squares in width. The software on the robot is designed to incorporate this fact as it records its movements and keeps track of its location.

4.2.4 Robot Software – Travel

Moving the robot through the environment is a major issue. The vector compass is the essential component of this portion of the system. The robot cannot even move in a consistent straight line by itself, due to hardware limitations of the motors and unknown qualities of the environment, such as dirt on the floor. The addition of the compass allows the robot to know which direction it is heading in, and correct for any errors that may occur during travel.

To help this system and to reduce the probability of error, as well as simplify the task of coding, the robot was restricted to four directions of travel. These directions are determined when the robot is first activated, and is guaranteed to be oriented in the correct direction. When the robot is still sitting in its starting position, it first checks for normal operation of the compass, and then sets the primary direction, which is considered to be north. The other three directions are set by incrementing the heading by ninety degrees. These numbers are then checked to ensure that they do not exceed the upper boundary of three hundred fifty nine degrees. In this event, the number is decremented by three hundred sixty degrees to bring it back into the proper range. Whenever the robot needs to change direction, it is done in terms of moving in the direction of north, east, south, or west.

When the robot does need to turn, there is a function that turns to this new heading. The robot turns in the direction that brings it from the current heading to the target heading in the least amount of time. The algorithm behind this turn is quite simple, and is as follows:

```
X = Current Heading
Y = Target Heading

if |X – Y| >= 180, and X >= Y → Turn Right
else if |X – Y| < 180, and X < Y → Turn Right
else if |X – Y| >= 180, and X < Y → Turn Left
else if |X – Y| < 180, and X >= Y → Turn Left
```

This function will turn the robot to within five degrees accuracy. The accuracy of the compass does not allow for an exact system that would turn the robot to within one degree of accuracy. Five degrees seemed to be the best amount of accuracy that I could achieve.

The same algorithm is applied in function to keep the robot travelling in a straight line. The function is constantly called when the robot is in motion, and makes small adjustments to the power of each motor in order to keep the robot moving in a straight line. If the current heading of the robot is more than ten degrees off from the desired heading, the robot stops all forward motion and calls the function to turn to within five degrees of the desired heading. The combination of these two functions keeps the robot on course with a very good degree of accuracy.

4.2.5 Robot Software – Obstacle Detection

Obstacle detection plays a large role in the navigation and mapping system. When the robot is travelling around the environment, the forward-mounted touch sensors must constantly be checked for contact. I created a function that checks each sensor, and returns the number of the sensor that had contact. The four touch sensors must be distinguishable as the system needs to know where the robot encountered an obstacle, for the purposes of mapping. Having only four touch sensors makes this an approximation, but this is sufficient. The function to detect obstacles is called during normal travel, when the function to correct for the proper heading is called.

Once the robot successfully reaches the proper location, it calls a function to perform the sensor sweep. This function slowly rotates the robot around three hundred sixty degrees. At every ten degrees it takes an IR sensor reading and stores it in an array. As the likelihood of the robot being able to stop at every tenth degree is relatively low, I implemented this system so that it doesn't bother to attempt to achieve the precise heading, but rather rotates slowly and takes a reading once the tenth degree is achieved or passed. This way the robot only has to download the heading that began the sensor sweep, and the thirty-six sensor readings taken.

4.2.6 Robot Software – Light Tracking Network

The light tracking network is the only neural net actually implemented on the Handy Board. I had to make some changes to the code in order to bring my neural net code from the Linux system to the Handy Board. The first is that Interactive C neither requires nor accepts prototyping the functions used, as is possible on the Linux system. The next change is that the "main" function needs to be declared as "void." Next, Interactive C does not accept "#include" statements. Some of the functions that used calls to "math.h" also needed to be changed at this point, to make them compatible with math functions built-in to Interactive C. The next change to be made was that all variables and functions declared as "float" needed to be changed to "double." These were the primary changes that needed to be made in order to have a neural net work on the Handy Board.

The final structure of the net contains two input nodes, two hidden nodes, and one output node. The structure was such that it took the leftmost light sensor reading as the first input, the rightmost light sensor reading as the second input, and the output was the direction that the robot should turn to. As the robot is only allowed to turn to the left or right, the third and middle light sensor on the robot is not necessary, and was left out so as to simplify the learning process. This middle light sensor is left on the robot in the hopes of creating a more comprehensive light tracking system that will incorporate all three sensors. The light tracking net will be discussed in more detail in Chapter 5.

4.2.7 Robot Software – Infrared Interpretation Network

The neural network for interpreting infrared sensor readings is actually located on the Linux system. This was due to the size of the network. The net was trained on the Linux machine, and it was just as easy to upload sensor readings from the robot as it would be for distances, so I decided to have the Linux machine hold the network and perform the calculations.

However, the robot was the source of the data for the training set, and this simple data-gathering task is worth mentioning. I would position the robot a set distance away from a large object, usually a wall. The robot would then move slowly towards the object, and record the distance traveled by the shaft encoder every time the infrared sensor reported a change in reading.

When the robot ran into the obstacle, it would stop. The robot would then download the sensor reading and corresponding distance to the object. The distance was determined by subtracting the distance traveled by the robot at the change in infrared sensor reading. This process took a significant amount of time due to frequent acknowledgment of transmissions. This information was then stored on the Linux machine for the process of training the network.

4.3 Linux Software System

4.3.1 Linux Software – Communications

The communications process and requirements were discussed in the previous section about the robot software. The code on the Linux machine is fairly simple. There are standardized routines for accessing a serial line on a PC, and this code merely utilizes these routines. Reading and writing to or from a device such as a serial line is basically the same as reading or writing to or from a file. The major difference is that the program needs to be run as root in order to access the device.

4.3.2 Linux Software – Map Representation

The map is represented in a grid coordinate system. While the system works under the theoretical premise that the unknown environment is very large, this premise is not practical for several reasons. The first is that the physical space available to me was very limited, and increasing the size of the environment would have increased mapping time considerably. Additionally, I was limited by the memory and computational capabilities of the computer being used. A large map would take a very long time to process and would inefficiently use up memory resources. While a very large environment would be possible with a more powerful computer system and a larger environment, in light of the restrictions placed on me, I had to limit the possible size of the map to a two hundred by two hundred grid. Each grid square is a component in a two hundred by two hundred array in the Linux software.

Each grid square represents a number of components. As each grid square can either be occupied or free of obstacles, there must be some way of keeping track of the status of a grid square. As the robot is expected to return some "noisy" data and data that conflicts with previously recorded information, I deemed it necessary to assign levels of confidence to the current status of a grid square. This is determined by the number of times the square was visited, and the status of the square as it was found during that visit. The status is determined by the greater number: the number of visits the square was found to be empty versus the number of times it was found to be occupied. In the event of both numbers being the same, the system assumes that the square is filled. In my system the map is assumed to be static. That is, there are no moving obstacles, and the environment never changes. The confidence level is determined by dividing either the number of visits that showed the square to be empty or the number of visits that showed the square to contain an obstacle (whichever is larger) by the total number of times the square was visited. For instance, consider a square that has been visited ten times. Say that the square was found to be empty two times, and found to be occupied eight times. This means that the square is considered to contain an obstacle and has a confidence level of 0.8, as yielded by dividing eight by ten. A grid square is not considered to be mapped until the confidence is greater than point five, and the total number of visits to the square is at least nine. This is done to ensure that erroneous data is discovered by comparing multiple trips to the same location. While primarily serving to guarantee that the correct map is discovered, this also affords the machine learning system enough opportunity to gather a sufficient training set.

Another component of a grid square is whether or not the square is on the horizon of the known map. For this mapping system, the horizon is defined as the outer edge of a mapped region, and is used in the large learning system as an indication of the unknown area traversed by the robot in a given path. The horizon is an expanding region of areas that are considered to be mapped. The area is contained by consecutively mapped squares or an outer wall. Outside the horizon is considered to be completely uncharted territory, regardless of how close to being mapped the area is. An area that is contained within the horizon is considered to be known and safe for the robot to traverse without difficulty. If this is not true, the area has been incorrectly mapped. Assuming normal operation of hardware system, these errors will eventually be discovered, and if the area is traversed enough times, will be corrected on the map. The map may contain islands of mapped areas and therefore many different areas with horizons, due to the mapping strategy and the random nature of creating goal locations for the robot to achieve.

Each grid square is represented as a structure in an array of structures. This structure contains variables to convey all necessary information:

1. The number of times the square was visited, either through the robot physically moving to the space, or by a sensor sweep, and found to be empty

2. The number of times the square was visited and found to contain an obstacle
3. The status of the square (zero being empty and one being filled)
4. The level of confidence in the current status of the square
5. Whether or not the square is on the horizon

These variables are set at various times, and are all set to a default state in the initialization of the map. All of the values are set to zero, meaning that the square has not been visited, is not on the horizon, and is assumed to be clear of obstacles with zero confidence.

4.3.3 Linux Software – Unreachable Areas

As the environments created will contain obstacles of notable size, and as it is likely that the theoretical outer edges of the environment will not be reached, it is necessary for the location-generating system to recognize the existence of solid objects, so as not to enter into infinite attempts to reach an unreachable area of the map. This is a deceptively difficult task. The outer edges of these objects are the only parts of the obstacles that will be discovered. However, depending on the accuracy of the sensory equipment and the generation of random locations to visit, it is possible that the outer edges will not be exactly determined until much time and many paths have occurred. However it would be much more efficient to realize and recognize these solid objects early in the mapping process, so that time is not wasted attempting to reach unobtainable areas.

The method for recognizing the existence of solid objects is rather time and computationally intensive. The process examines every known grid square in the map. If a square is filled, the system attempts to follow the path of filled squares parallel to the x-axis, if there is such a path. When the end of this path is found, the process then moves along the y-axis, again following the filled grid squares. This continues, alternating between x and y-axes. If the starting coordinate is reached again, then the area inside the boundaries of this outer rectangle is marked as being occupied by an obstacle. This is a simple iterative process, which assigns values to the number of times visited and the status of the square in order to designate these squares as filled. This system is redundant, but the repetitive nature of the system helps to ensure that objects will be recognized by the system.

Obstacles with large boundaries are recognized as being the outer edges of the actual environment, and are treated as such, marking the area outside of these edges as filled. This system is not ideal in that it may not be likely that the robot will accurately determine the exact outer edges of an obstacle. The concern here is that the system may be able to trace the outside of an object, but may not end up exactly at the same coordinate that the process started at. Also, if there are obstacles up against the walls of the environment, this system may have a difficult time identifying these objects. Unfortunately this system was not tested with any real data, due to the problems with the vector compass. The extent of the limitations of this process is not known. The full system to recognize the outer walls of the environment is not yet fully implemented, due to the problems with the compass.

4.3.4 Linux Software – Random Number Generation

At several points I needed to make a random decision or choose values randomly in order to create possible locations to travel to. For decision making I only needed two possible values, but for creating locations for the robot to travel to I needed to be able to create numbers that covered the entire range of the size of the map. So I combined these requirements into a system that generates a random number between 0 and 199, inclusive.

It is difficult to create numbers that are actually random, but programming languages offer a number of options that can serve as solutions to this problem. While systems exist to generate random numbers, I did not have any viable options when I needed one, so I chose to implement my own system. My solution was to get the current microsecond and store it in a variable as the number of microseconds so far in the current second. I then take the sixth digit from the right (the one hundred thousands place) and store this value. I then take the third digit from the right (the one hundreds place) and store this number. I then take the second digit from the right (the tens place) and store this value. I then place these three digits into a new variable which gets returned to the calling function. The digits are put into place by multiplying each by one, ten, or one hundred. The resulting integers are then added together. The magnitude by which the three digits are placed varies on a rotating basis, such that one will take the hundreds place, one the tens place, and one the ones place, but they will not take the same position until five more random numbers have been created. This prevents two random numbers called in rapid succession from being related or close to each other, in most cases. Any of the three digits taken from the current microseconds value can be zero, so the system actually does cover the range of zero to nine hundred ninety nine. One drawback to this process is that it takes more computation time than is desired. This system has proven to be sufficient, and is certainly good enough for the requirements of this research.

4.3.5 Linux Software – Path Generation

The system that generates possible paths for the robot to take creates a number of options for the neural net to choose from. In the interests of speeding the learning process, and thus the mapping process in general, the system is guaranteed to produce some desirable options. The system will create seven completely random locations to travel to, regardless of whether or not they're already mapped or even if they are filled. The remaining three paths are guaranteed to go to an unmapped location on the map, as long as there are unmapped locations to go to. This is done by checking to make sure that the target location chosen is not yet considered to be mapped. This does not mean that one or more of the random locations will not be a better choice than one of these three "good" choices, but it means that there will always be somewhere desirable to go to, so that when the machine learning system has been trained sufficiently, there will be a good option for it to recognize and choose.

A variable between possible paths for the robot to take, besides simply the coordinate traveled to, is the number of waypoints within the path. For this project I have defined a waypoint to be a point where the robot changes its direction of travel. This allows the robot to move to locations by avoiding known obstacles and to gather more data per trip by covering more ground. I decided that it would be pointless to have a waypoint where the robot does not change direction, so there are special cases, such as when the starting coordinate and goal coordinate are aligned along an axis of travel, which need to be handled separately.

Note that if a path is not possible due to obstacles, the path generator will move the variable coordinate component successively closer to the starting coordinate in the hopes of achieving a clear path. Should this fail, the attempt to generate a path with that particular number of waypoints will also fail. If a path for a particular number of waypoints cannot be generated, there will be one less in the total number of paths that the neural network has to choose from.

The function to generate a path with zero waypoints is the only function that actually changes the stored path that the robot will follow. The functions to create paths of one to five waypoints all call the function to create a path with zero waypoints. In addition, each of these generators also calls the function for the next smallest number of waypoints. That is, the function to generate a path with n waypoints makes a call to the function to create a path with zero waypoints, and then calls the function to create a path with n – 1 waypoints. The function for n – 1 waypoints then calls the function for zero waypoints, and then the function for n – 2 waypoints, and so on, until the function call reduces to n + 1 calls to the function for zero waypoints, thus creating a path with n waypoints and n + 1 transitions between them.

Each function for creating a path with more than 0 waypoints makes a random decision about the initial direction of travel. As it is pointless to have a waypoint in line with the start and goal coordinates, this possibility is excluded by overriding the random choice of direction in the subcalls. The end result of this is that only the direction of travel from the starting coordinate to the first waypoint is random, and the remaining movements alternate between x and y, depending on the initial movement. A more complete version of this system will allow for initial movement in one of four directions. Given the current system of path generation, this is neither required nor possible.

The functions for creating paths with certain numbers of waypoints takes a number of parameters. These consist of the starting x coordinate, the starting y coordinate, the goal x coordinate, the goal y coordinate, the randomization override value, the index into the list of commands, the number of waypoints currently being attempted for that index, and the number of the point that is currently being attempted. The starting and goal coordinates are self-explanatory. The randomization override value will only be zero, one thousand, or negative one thousand. This value is added to the result of a call to the function that creates a random value between 0 and 199 such that if the override is zero, the value remains random; if the override value is one thousand, the random value is skewed to force the function to move in the x direction; and if the override value is negative one thousand, the random value is skewed to force the function to move in the y direction. This prevents the case where a path could contain waypoints that lie in line. The index into the list of commands and the number of waypoints being attempted keep track of which values in the array of possible moves are currently being altered. The number of the current point keeps track of the order of the waypoints.

For example, the call to create a path with three waypoints would contain the start and goal x and y coordinates, a value of 0 as the override value, the current index into the list of commands, a value of three for the number of waypoints, and a value of zero to indicate that the first waypoint is being created. After the initial direction of travel is chosen, a call for a path with zero waypoints is made for the first waypoint, and a call to create a path with two waypoints is made, after incrementing the current point being created.

Each of these functions has some built-in capability for dealing with waypoints that cannot be reached. If an attempt to create a path between a set point and some attempted waypoint fails, the function can alter that waypoint within a specific range in an attempt to find a more viable coordinate. For example, if a path with two waypoints fails on the first waypoint, the path generator will move the first waypoint closer to the original point. If the waypoint gets too close to the starting point, the system will cease its attempts to create that path and report a failed attempt. This system is not ideal in that it is not exhaustive; that is, it does not seek out every possible path with three waypoints before reporting that it is not possible. However, this system is only a tool through which to focus on the learning system, so I deemed this path generator sufficient for creating paths.

The first path calls for zero waypoints. This path is simply a straight line from the home coordinate to the goal coordinate, and thus is only possible if either the x or y components of both locations are in line. A path with zero waypoints is not always possible, regardless of the density of the map and the placement of obstacles. The function first checks to ensure that either the x or y coordinates are in line, and then checks to make sure that the path is clear between the two locations. The system checks to see if the path is clear between two points by projecting the path of the robot between the points. It does this by centering a line on the two points, and then examining the area on either side of that line, in a width equal to the radius of the robot. This ensures that any space the robot will occupy is clear of obstacles. Figure 4.3.1 shows the two possible paths from point A to point B.

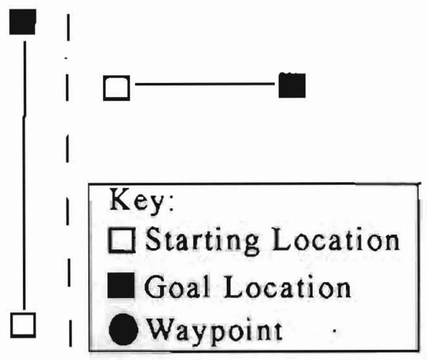


Figure 4.3.1 – Possible paths with zero waypoints

Like straight-line paths, paths with one waypoint are also not difficult to create. These paths are formed by two straight lines. This path can move first along the x-axis, that is, remain on

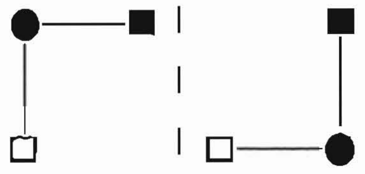


Figure 4.3.2 – Possible paths with one waypoint

the home y coordinate while travelling out to the x component of the goal, or move first along the y coordinate. Thus if the starting coordinate represents the bottom left corner of a rectangle and the goal coordinate represents the upper right coordinate, the waypoint will be either of the remaining corners of the rectangle, depending on if the path moves first along the x or y-axis. This rectangle concept is the premise behind all paths that are created. Figure 4.3.2 shows the possible paths from point A to point B with one waypoint.

Paths with two waypoints introduce some more difficult issues to be dealt with. In the case where the starting coordinate and goal coordinate are in line, whether it be along the x or y-axis of travel, the function should still be able to create a path with two waypoints. To accomplish this, the function will make the first waypoint out from the starting coordinate some random distance away, along the opposite axis of travel from the direction which is in line between start and goal coordinates. If the start and goal coordinates are not in line, the function makes the first waypoint in line with either the x or y component of the starting coordinate (where the initial direction is determined randomly), and out a value of half the distance between the respective coordinates of the start and goal. The second waypoint is created by moving along the other axis of travel so that the second waypoint is in line with the goal. Figure 4.3.3 shows the possible paths from point A to point B with two waypoints.

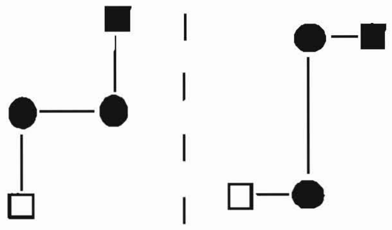


Figure 4.3.3 – Possible paths with two waypoints

Creating paths with three waypoints entails difficulties similar to those encountered in creating paths with two waypoints. If the start and goal coordinates are in line, the first waypoint is again chosen somewhat arbitrarily by a random value. Otherwise, the three waypoints are determined by a distance of half the distance between the respective coordinates of the start and goal. This means that, in the ideal situation, the second waypoint will lie at the center of the rectangle bounded by the starting and goal coordinates, as shown in figure 4.3.4.

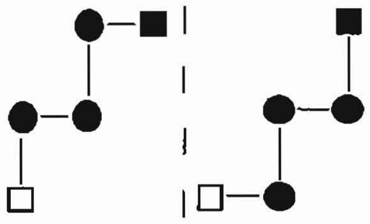


Figure 4.3.4 – Possible paths with three waypoints

Paths with four waypoints are slightly easier create. In the event of the start and goal

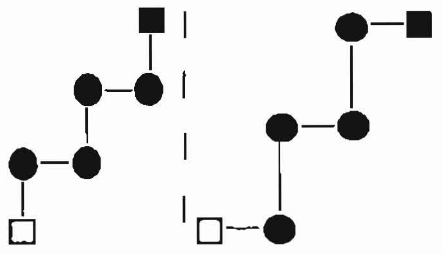


Figure 4.3.5 – Possible paths with four waypoints

coordinates being in line, the function forces the first waypoint to still be in line with both the start and goal coordinates. The function then calls the process to create a path with three waypoints, which will handle the situation of the coordinates being in line as previously described. Otherwise, the first waypoint is determined by randomly moving in the x or y direction a distance of one third the distance between the respective components of the start and goal coordinates. The rest of the path is determined by the creation of a path with three waypoints. The paths of four waypoints are shown in figure 4.3.5.

Paths with five waypoints are handled in the same way as paths with four waypoints. The only difference lies in the fact that the path is created by a path with zero waypoints, and then a path with four waypoints. These possible paths are shown in figure 4.3.6.

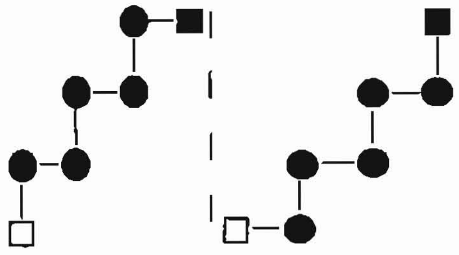


Figure 4.3.6 – Possible paths with five waypoints

This solution for generating paths with various numbers of waypoints is not the ideal one. There are arguments to be made for changing many aspects of the system. A more robust system would allow for travelling in more directions, and for maneuvering around obstacles. However, as this project merely calls for creating a variety of options for the strategy system to choose from, this system is sufficient.

4.3.6 Linux Software – Primary Learning System

Originally, I intended that the primary machine learning system on the Linux system was to be a different type of machine learning. I ultimately decided that the large scale learning system on the main Linux computer would be a neural network, in order to conserve the overhead of time involved in implementing a new machine learning system, and to be consistent with the rest of the project. The decision to remain with neural networks created a need to recast the representation of the system so that I could feed it into a neural net in a meaningful and useful way.

The neural network is given a number of different paths on which to send the robot for data collection. A representation of these paths is put into the net, and the system compares the output of each path to see which path will theoretically produce the most useful information. When a path is chosen, it is uploaded to the Handy Board. The robot then proceeds along this path and returns data from its trip. This data is used to update the current picture of the environment. The actual usefulness of the path is then determined based on the information gathered. This value of the actual usefulness, along with a measure of the success of the trip, becomes part of the training set for the net, and the net adjusts itself accordingly. Clearly there will be a correlation between a failed trip and low level of usefulness. However, the difference lies in paths that have high expectations for success, yet different values of usefulness.

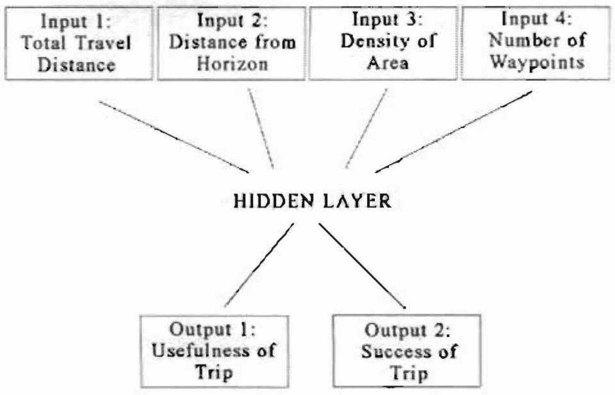

Input 1: Total Travel Distance
Input 2: Distance from Horizon
Input 3: Density of Area
Input 4: Number of Waypoints
HIDDEN LAYER
Output 1: Usefulness of Trip
Output 2: Success of Trip

Figure 4.3.7 - Input and Output of the Primary Neural Network

There are several aspects of a path that need to be input to the network. These are broken down into four major categories that compose the input nodes. The first aspect is the distance from the starting coordinate to the goal coordinate. This is measured by the distance that the robot will need to travel. While it can be argued that a straight-line distance would also be appropriate, this measurement does not necessarily convey the information that would have an effect on the success or failure of the trip. It would also undermine the usefulness of the path, as a larger travel distance entails covering more ground, and presents more opportunity to gather useful information. Determining this distance is a very simple task, and involves simply adding up the distances between the waypoints, including the start and goal coordinates.

The second input is a similar measure of distance. This measurement is from the horizon to the goal coordinate. For the sake of simplicity, if the robot crosses the horizon more than once, the distance is measured from the first crossing. The system assigns an initial horizon to the area immediately surrounding the starting coordinate, as the robot occupies this space to begin with. This also guarantees that the robot will cross the horizon. This input represents the amount of uncharted territory that the robot is travelling in. The distance covered in this measurement is the area that will be most useful to the system, as it has not been mapped yet, and may have a large impact on the accuracy of the mapping excursion. This is also a relatively simple value to compute, and involves adding the distances between the first place the robot crosses the horizon, any remaining waypoints, and the goal coordinate.

The third input to the net is a measure of the density of known objects in the area covered by the proposed path. The density of the area clearly can have a direct impact on the number of waypoints necessary to maneuver around obstacles, and can also have an effect on the estimate of success of the path in general. This information is included as a path through an area with high density may be less likely to be successful, due to the larger number of chances for the robot to run into an object where it doesn't expect one. This could serve to lower the predicted success of the path. The density for a particular path is determined by examining the ratio of filled grid squares to the total number of grid squares over the relevant area. The relevant area in this situation is defined as the rectangle formed by making the starting coordinate the lower left hand corner, and the goal coordinate the upper right hand corner. To make sure that this actually includes some information, a buffer of ten grid squares is added all around that rectangle. This process includes much error checking to ensure that the system does not try to step outside the boundaries of the map. The function iterates through each square contained in the rectangle, and increments a counter, depending on whether or not the square is occupied or empty. Finally the function returns the ratio determined by dividing the number of filled squares by the total number of squares in the rectangle.

The fourth input to the net is the number of waypoints contained in the path. This is important as it, in combination with the density of the area to be covered, may have an impact on the success of the path. A path with a large number of waypoints may introduce more opportunity for the robot get off track and become lost. The number of waypoints also has a direct correlation to the amount of data that can be collected, and hence affects the usefulness of executing that path.

While the actual training of this network could not take place due to the problems with the vector compass, I had a plan for this part of the project that should receive some attention. Whenever the robot returned to the Linux machine, it would report back information that the Linux system would then interpret. Part of the interpretation was to add the results of the trip to the training data for the network, by adding the input and the actual result to the training set. The usefulness of the trip is determined by taking a scaled value of the number of grid squares visited, and dividing this number by the number of seconds that the robot was on the excursion. The time spent on the trip is measured from the time the last element of the path to be traveled is uploaded to the Handy Board, until the Handy Board re-establishes a communications link with the Linux machine. This means that I need to be quick and consistent in attaching and detaching the serial link between the two computers. The number of squares visited is determined by counting all of the squares that the robot passed through on the path, including those covered by the width of the robot. The number of squares covered by the sensor sweep at the end of the path would be determined by counting every square within range of the sensor. Those squares that were already counted by the robot physically moving through them are subtracted from the count created by the sensor sweep. Every square encountered, through either travel or the infrared sensor, would have a value assigned to it, based on how useful it was to map that square. If the square was not considered to be mapped yet it would be more useful than rechecking a square that had already been mapped, and would receive a higher rating of usefulness. This is computed simply by assigning a larger number to the usefulness rating for mapping an unmapped square as opposed to re-mapping a square that had already been mapped. Squares that lay behind an obstacle in a sensor sweep were subtracted from the total number of squares visited. This would avoid the situation of rewarding a trip for squares that were not actually mapped, and also avoid reducing the usefulness of the trip simply because there were squares that could not be seen.

The success of the trip is fairly simple to determine. If the robot achieves its goal location and performs the sensor sweep, the trip is assigned the highest success value. If the robot encounters an obstacle and cannot achieve its goal location, the success is determined by dividing the number of grid squares that were actually visited by the number that would have been examined had the target location been achieved, including those covered in the sensor sweep.

If the trip was not one hundred percent successful, that trip will still be added to the training set. A second trip will also be added to the training set as well. This fictitious data set is the trip that would have performed had the target location where the robot encountered an obstacle. The input for this trip is determined using the same processes applied to actual trips, and the outcome is assumed to have successfully examined all possible grid squares, including all those that could be covered in a sensor sweep, and have taken the same amount of time that the failed trip actually took.

4.3.7 Linux Software – Light Tracking Network

The final version uses two input nodes, two hidden nodes, and one output node, as previously described. Output for this node is binary, where a 0 signifies turning left, and a 1 signifies turning right. While the network is used on the Handy Board, the network was pretrained, in that the starting weights were hardcoded from the results of a network run on the Linux machine. This net will be discussed in greater detail in Chapter 5.

4.3.8 Linux Software – Infrared Data Interpretation Network

The neural network to interpret infrared sensor readings is similar in structure to the light tracking network, except that this net runs exclusively on the Linux system, and uses more hidden nodes. The net contains one input node, twelve hidden nodes, and one output node. The input value is the reading from the infrared sensor. The output value is the distance in terms of clicks of the shaft encoder. The structure, results, and experiments associated with this net will be discussed in greater detail in Chapter 5.

Chapter 5
Experimentation

5.1 Introduction

There were three neural networks that I designed and implemented in this project. All of the nets utilized the same underlying code, and varied only by the number of nodes used in each layer, and the number of iterations of pre-training. The first net served as a light-tracking system for the robot, the second was a system for interpreting the data returned by the infrared ranging sensor, and the third net was a system for creating a strategy for efficiently mapping an unknown environment.

Each of these networks had its own design and implementation issues that I had to confront. There were some overlapping issues and problems that applied to the nets, however the solutions were generally unique to the specific situation. Successfully completing these nets involved some degree of trial and error and experimentation.

5.2 Light Tracking Neural Net

5.2.1 Structure of the Net

The light tracking neural net is the only network that was run on the Handy Board. It is also run on the Linux machine however, in a pre-training process. The structure of the network is the same on both computers, despite the necessary implementation differences. On both machines the final version of the net consists of two input nodes, being the left and right light sensors, two hidden nodes, and a single output node, consisting of the direction to turn.

Earlier versions of the net, however, utilized all three available light sensors. These versions of the net had three input nodes, one for each light sensor with the left sensor being the first node and moving to the right. The network at this point had more hidden nodes as well to allow the three input values to be fairly represented. The number of hidden nodes ranged from six to twelve. I also experimented with the number of output nodes. I toyed with the idea of having the network output a binary number signifying the direction to turn, followed by a second output, which was the number of degrees to turn in that direction. However as the compass was not yet implemented I could not pursue this approach. Another structure of output nodes that I attempted was a system that would have two outputs, which would represent power sent to the two motors. As it turned out, the structure of the output nodes may have been viable options for a two-input system, but the three-input node system turned out to be too complex to train. This was due to reasons of the training set involved, and will be examined in more detail.

In the final version of the net, with two input nodes, two hidden nodes, and one output node, I also varied the number of hidden nodes during the experimentation process. I started this version of the net with more than two hidden nodes, and gradually worked my way down through a process of experimenting with the structure of the net. Clearly a smaller network is preferable to a larger one due to memory and computation limitations. I was fairly surprised to observe that it was actually easier and faster to train a network that contained two hidden nodes as opposed to some larger number.

5.2.2 Pre-Training

Running the net on the Handy Board took a large amount of time. I initially started training the net with a very idealistic view of the system. I planned to run the robot through many actual scenarios, providing a supervised learning system by telling the robot which way to turn by utilizing the start and stop buttons on the Handy Board. I did this for both the three and two-input node versions of the net. With the three-input version, the robot had three options for travel: left, straight, or right. With the two-input version I limited the options to either turning left or right. While not perfect, the robot does not turn very far in a single move, such that the robots inability to go straight is not problematic.

The process to this point had been to place the robot in the environment, let it take readings, tell it the direction to turn, let it loop through the training loop a few times to speed up the training process, and then have it run the input through the net and turn in the direction prompted by the output, regardless of its correctness. The robot would then move a short distance in that direction. I rapidly got tired of waiting for the robot to turn and move, so I cut this part out of the system, and chose instead to place the robot in realistic positions to create the training set.

My hope was that this process would quickly begin to reveal that network was learning which direction to turn in, and that I would observe a shift towards the robot making better decisions about which direction to turn in. It quickly became clear that this process would take much too long to be useful. As a time-saving innovation, I opted to pre-train the network. I did this first on the Handy Board, but it became clear that the Linux system could perform this pre-training much faster than the Handy Board was capable of. When I did pre-training on the Linux machine, I needed to get the final weights from the Linux system to the Handy Board. As the communications software was not yet fully implemented at this point, I did this by hand, entering the weights on the Handy Board as the initial values of the weights for the Handy Board's net. I could also test the net on the Linux machine, and did so by running through a series of input scenarios. Once I had a working network, I experimented with the system by starting the training process over and reducing the pre-training that was done. Finally I had a net that was pre-trained as little I found to be necessary, and I put this less extensively trained net on the Handy Board. From there I continued training on the robot, and was then able to see improvement in the decisions that the network made.

5.2.3 Training Set

The original training set consisted of actual data gathered by the robot. As previously discussed, this proved to be much too time consuming. The next training set that I used was generated by the robot, by placing the robot in realistic positions and recording the sensor readings at those points. I quickly realized that this wasn't really necessary, and began to generate my own data points by extrapolating from the real points. This was necessary as it became clear that I was going to need more than a handful of data points in my training set.

Eventually it seemed as though this was not going to be sufficient. At this point I introduced a series of completely fictitious data points to the training set. This began when I was still using three input nodes instead of the final version consisting of just two. I abandoned my actual data points, and replaced them by looping through a series of artificial data points. I did this by looping through possible input values with various increments between the input values. For instance, for a series of points with an increment of ten, the data points that would have the robot turn left would look like 20, 10, 0; 30, 20 ,10; and so on. Points having the robot go straight would look like 0, 10, 0; 10, 20 10; and so on. This was done for values within the range of zero to two hundred fifty five (the output range of a light sensor), and for increments of ten, twenty, and thirty. This worked very well for the cases where the robot had to turn either left or right. However, in more than half the scenarios where the robot needed to move straight ahead, the net would tell it to turn one way or the other. This was troublesome and time consuming to attempt to track down. Therefore I moved to the two input node version of the net.

I also used my same incremental pre-training model on the two input version of the net as well. This obviously involved only two inputs, and so it was much easier to produce the training set and train the net. Upon examining actual data from the light sensors however, it appeared that the angle between the left and rightmost sensors was such that there would not usually be a difference as small as ten between the two readings. Therefore I changed my training set to run in increments of twenty, thirty, and forty. This version seems to work very well.

5.2.4 Training the Net

Training the actual network took a surprisingly small amount of time. In the versions of the network and training set that I came up with prior to the final version, I increased the number of iterations through the training set in an attempt to gain better results. However as I approached the final version of the net, I was surprised to observe that the number of iterations needed to train the network was much lower than I thought would be necessary. With each iteration I would run through the entire training set once. This is a sizable amount of information. Eventually though, I determined that it was only necessary to run through one hundred iterations of the training set to train the network. Any larger number of iterations would only serve to reinforce the function that the net had already learned. To put this in perspective, one hundred iterations through this training set would take less than a minute, which is substantially less than other networks that I was running in this project.

5.3 Infrared Sensor Interpretation Neural Net

5.3.1 Structure of the Net

This network served the purpose of interpreting infrared ranging sensor data by putting it in terms of a distance in units of clicks of the shaft encoder. With this in mind, it is clear that there would be one input node; the value of the IR sensor, and one output node; the distance in terms useful to the map. With the experiments of the previous net showing that fewer hidden nodes can often be preferable, I used this approach to begin with. However, due perhaps to the complexity of the function, lower numbers of hidden nodes did not seem to generate better results for this network. This function is made complex by the extremely noisy nature of the data. The training set is not a one-to-one relationship at various points, although the network is attempting to create a one-to-one function during the training process.

Once I determined that fewer nodes would not provide the solution that I needed, I focused my experiments on slightly larger numbers of hidden nodes. My experiments ranged from ten to twenty hidden nodes. The final version uses twelve hidden nodes. My procedure consisted of training the net, and then running a test set through the network. I then compared the outcome of this test set with a sample of the training set to see how close the two were. I did this by graphing both together, and comparing the lines graphed. The actual training set would contain some noisy data, and some conflicting data. My measure of success of the net was based on how close together the two graphs were. Where the training data became noisy and contained conflicting data, I looked for the trained net to follow the average of this data.

It is difficult to draw conclusions based solely on varying the number of hidden nodes. The number of iterations at which it was necessary to train the network was such that it took anywhere from three to eight hours to train the network and have data that was worth graphing. Due to this, I was unable to vary one parameter at a time with every attempt at running the net. Fewer numbers of hidden nodes meant a shorter training time, however often not significantly so.

5.3.2 Pre-Training

Like the light tracking net, this network is pre-trained. However, unlike the light tracking net, this net is entirely pre-trained. In the light-tracking system I allowed for the robot to continue the training process during real-world situations, whereas this net is pre-trained with the training set and then thought to be entirely static. I assume that both the IR sensor and the shaft encoder will present constant findings over time. However, it is necessary to note that this process must utilize the original findings of the size of a grid square. This system was run with no other sensors enabled or running, and little other code running to take up processor time and memory space. As previously discussed, the rate of clicks returned from the shaft encoder is about halved when the complete system is running on the robot. But the data collection for this network was performed without the complete system running, as it hadn't been implemented yet. Therefore it is important to remember to use the original size of a grid square when using the output from this net.

5.3.3 Training Set

The training set for this net was gathered by using the robot. I gathered quite a bit of information for the data set, and most of it was fairly consistent. The IR sensor has an effective maximum range, which is fairly apparent when examining a graph of sensor readings versus distance as measured by the shaft encoder (Figure 5.3.1). As can be seen in the

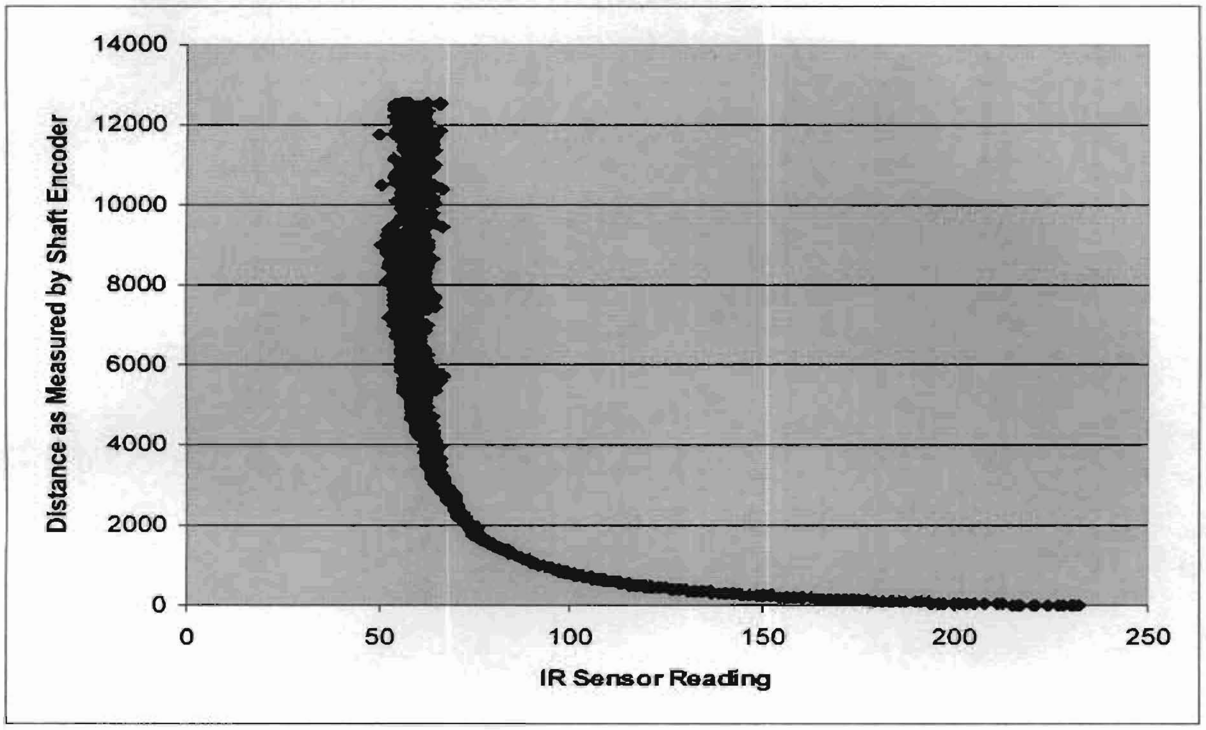


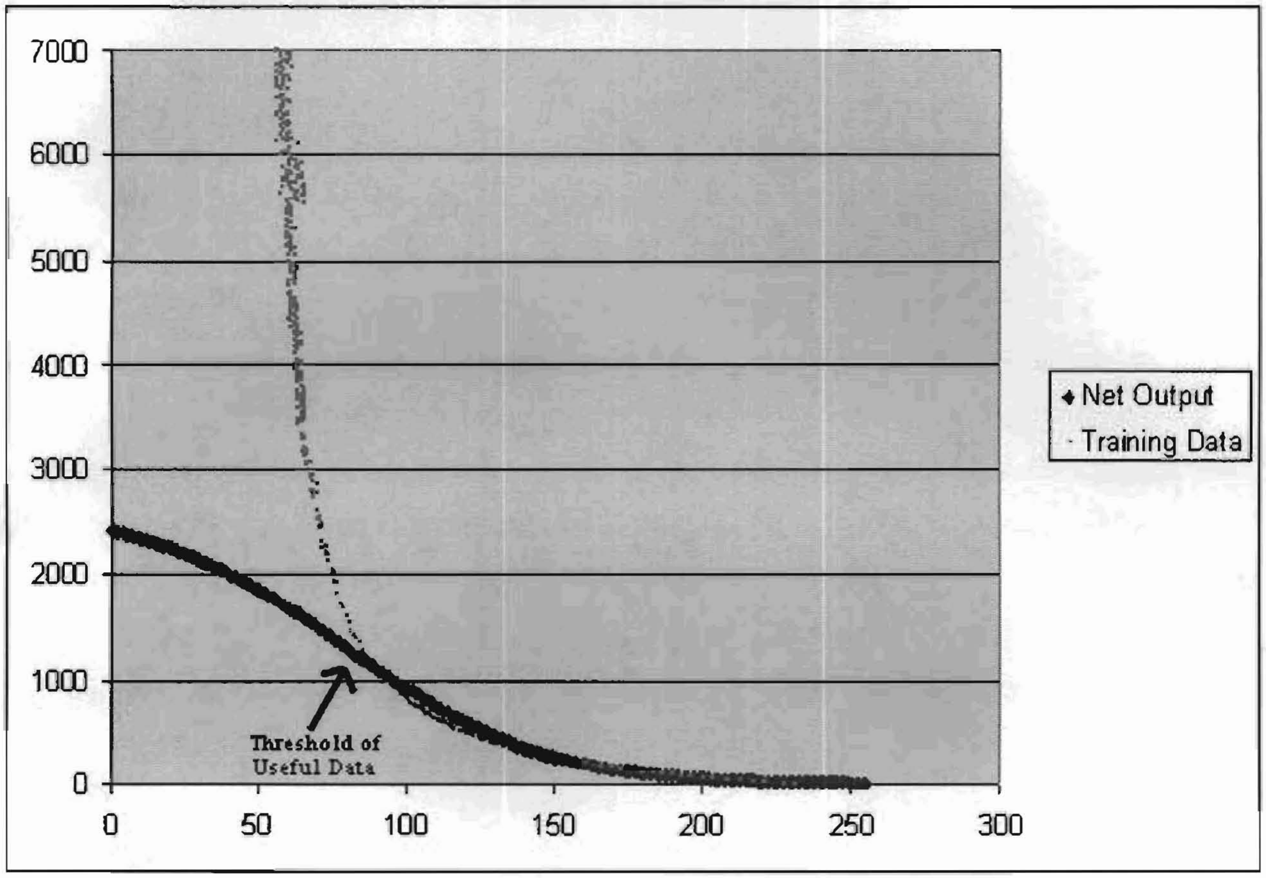
Figure 5.3.1

graph, the data is fairly consistent through an IR sensor reading of about seventy. That is the point at which the sensor data becomes consistently noisy and difficult to interpret. As I attempted to train the network, I found that the results were being shifted up the graph, and I attributed this to the fact that the data set was not representative of the area that I cared about the most. This was due to the way that I gathered data.

I gathered the data by having the robot record the distance traveled every time the IR sensor changed its reading. However, the sensor changed quite a bit more often in the noisy range, so there were many more data points in this range. The sensor gathered more data in the noisy range as it would often receive many different readings when located at the same distance away from an obstacle. As the robot moved into the range where the sensor started to return values of seventy and above, the correspondence between IR sensor and distance from the obstacle became one-to-one, so there was only one distance for each IR reading, and hence fewer data points to consider.

Once I realized this fact I started to count the important range of data, where the IR sensor read seventy or greater, many more times, in order to have both ranges count equally as much, and to make the training set more representative.

As time progressed though, I shifted more and more importance onto the range between seventy and greater. Finally I cut the other values out of the training set all together. There were



5.3.2 – Output of IR Net vs. Training Data

still many issues with the net, but I was finally able to obtain a net that produces meaningful information over the useful range of IR sensor readings, as shown in Figure 5.2.2. I determined the useful range of the sensor to be for readings greater than seventy.

5.3.4 Training the Net

Training the net was a long and time-consuming task. With every change that I made to the network, I would need to train the net and check the output. As training the net took anywhere from three to eight hours, and better results were not necessarily guaranteed, this was often a discouraging task. The number of iterations ranged from five thousand to one hundred twenty thousand. I was attempting to get as precise an output as possible from this network, so adding a few thousand more iterations in order to get a single unit closer to the target data was worth it. It quickly became apparent that as I trained the net more, I had to make more drastic changes in order see any effect. For instance, the difference in results between running the training set for one hundred twenty thousand iterations and ninety thousand iterations is very small.

5.4 Mapping Strategy Neural Net

It is impossible to discuss this network in great detail, as the problems with the electronic compass limited the results that I was able to gather. The structure of the network has been discussed in the previous chapter. I never settled on the internal structure of the hidden nodes for this net, as I could not experiment without a real data set. My plan for this network was to run the training set through the net after each data gathering excursion performed by the robot. I anticipated that I would run the training set through several iterations each time it was run. I also planned on creating a system to cycle through the training set, as the initial trips would be over-represented in this training scheme.

I contemplated the idea of training this net on data generated by me, but chose not to pursue this approach. I could easily generate locations to travel to, and create a series of data to train the net with, however this would not take into account the hardware discrepancies that I anticipated would be present on the robot. As this is a major point of interest for me, I chose not to continue along this path.

CHAPTER 6
Conclusions

6.1 Introduction

A research project such as this one can never really be considered to be completed. There is always some aspect left unfinished, or some component which can be expanded or enhanced in some way. My project is no different than any other in these regards. Despite these areas that can and should be expanded or completed, there are many other areas which are complete, and a great number of lessons which have been learned.

There are two main aspects to reflect upon, the first of which is what I learned. When I examine these issues, I focus on passing on the lessons that I learned to someone else, rather than listing the numerous topics that I learned about. The second aspect is that of what is completed, and what needs to be done. I gear this section primarily towards future researchers, so that others will know of the issues I am facing, and in the hopes that others will apply their perspectives and ideas to further my research.

6.2 Lessons Learned

6.2.1 Robot and Hardware

Given that I had never had any instruction in creating hardware systems such as the ones incorporated into my robot, this was an area involving a great deal of research as well as trial and error. The first strategy that I would recommend involves giving a lot of thought to the design of a robot. There were many issues and problems that arose with my robot that I could not have foreseen, and it seems as though this is a general truth. So to minimize this issue, I strongly recommend putting much time and energy into thinking about the demands that will be placed on a robot, and implementing and testing prototypes whenever possible. This will decrease the number of long-term issues that will need to be confronted, and make for a more robust robot system in general.

Along a similar line, giving a lot of thought and planning to choosing which sensors to use is another time-saving recommendation. I put a good deal of thought into the requirements of my robot, and this helped in my choosing sensors to purchase and implement. Along the same line, I whole-heartedly advise getting hardware components which are best suited for one's skills and abilities. In my case, I would have been much better off buying sensors that required less construction. While my goals for this project involved learning about wiring and soldering electronic components, not all of my sensors were quite within reasonable grasp of my skills. I sank a great deal of time into the implementation of my sensors, and in the case of the electronic compass, was not entirely successful. While I am grateful for the opportunity to learn what I did about soldering and wiring, too much time was spent on these hardware issues, and ultimately more software and results would have been achieved had I been able to eliminate these hardware issues in a more timely fashion.

Similarly, I have found that it is much more beneficial to seek out help from those more experienced rather than attempting to force through some issues. While it is not helpful to anyone to simply ask for help from the start without making some sort of effort, there are many resources which can serve as educational tools. Without any background in electrical engineering, it was essentially impossible for me to interpret the electrical schematics of sensors without external help. Sources such as the internet and the Handy Board Mailing List[24] were invaluable to me in determining how to wire the sensors that I used.

6.2.2 Neural Networks

As with my background in hardware topics, I was relatively new to machine learning and neural networks at the beginning of this project. When I refer to the size and complexity of a network, I am referring to the number of hidden nodes in a net. In my experiments, the number of input and output nodes was fairly obvious for a problem, and thus was not really variable. Thus the only variable left for the structure of the network is the number of hidden nodes.

The biggest piece of advice that I can offer to someone experimenting with neural nets is to start small and build up from there. This is true for several reasons. First of all, it simply makes sense to start with a simple design and build complexity into it. By starting with a simpler design and adding to it I was generally able to watch the results get better as experimentation progressed. Then it was a fairly simple task to add complexity to the net until the performance of the network was satisfactory, and the performance increases yielded by further complexity were negligible. Another benefit of starting small is that a smaller net takes less time to train and examine the

[24] Handy Board Mailing List

results. As the complexity of a net increases, the training process takes longer, and the process of gathering and comparing results thus takes more time.

The truly difficult part of experimenting with neural nets is recording the differences between the nets and the results of each. As I varied such components as the number of hidden nodes and the number of iterations of the training process, each different net would yield a set of results. First of all, it is difficult to determine whether a change in the number of hidden nodes in a net or the number of training iterations made a change in the results of a net. Therefore it is important to vary only one of these variables at a time, which can be rather time consuming depending on the size of the net and number of iterations for the training set. I have found that keeping a good set of notes for each change made to the net is critical to being able to track the effects of changes. This is especially true if a test of the network takes several hours or days.

6.2.3 General Lessons

Generally, my most significant piece of advice is to write about sections and systems as they are completed. I did this to some degree, and increased my policy of this as I progressed in the project. When doing background research it is easy to make write summaries and small topic papers along the way, and these mini papers can be plugged into a final paper with relative ease. This is somewhat more difficult when creating a hardware system, but design notes and brainstorm sessions are a good way to track the design and thought process behind designing a robot. Keeping track of changes when writing code may be the most difficult task of all. It is very difficult to write about code before it is completed, given the large number of problems that arise and changes that end up being made to code before it is complete. I have found that the best solution here is to keep a good system for commenting code. Well commented code can not only be understood by others who might wish to read the code, but it also serves as an outline for a paper. Comments in code spell out the process and thought behind the code in a concise and straightforward way, and in my case could often be put directly into a paper.

6.3 Future Work

Many aspects of the project that I originally set out to do have been successfully completed. Some others are currently held up by hardware implementation issues. The completed topics are certainly not trivial, and are all discussed earlier in this paper. Among these are such large-scale topics as building the robot, building the software system to control the robot and tie together the learning systems, and implementing three nets and testing two of them. As mentioned at various points in this discussion, there were several topics that were not completed, or were not completed to my satisfaction.

Clearly the problems with the electronic compass prevented some of the project from being completed. Also, some components, such as the path generation system, were implemented only to the point of being sufficient for the current state of the system. Topics such as these could be furthered to be more complete and operate with greater efficiency. These current issues and my hopes for the project help to create an impressive list of topics for future work.

My first hope for future work is to complete the system as it lies now. This primarily entails working on the electronic compass more, and making it work correctly. The proper operation of the compass would yield the ability to test the rest of the code that I wrote, including the primary learning system.

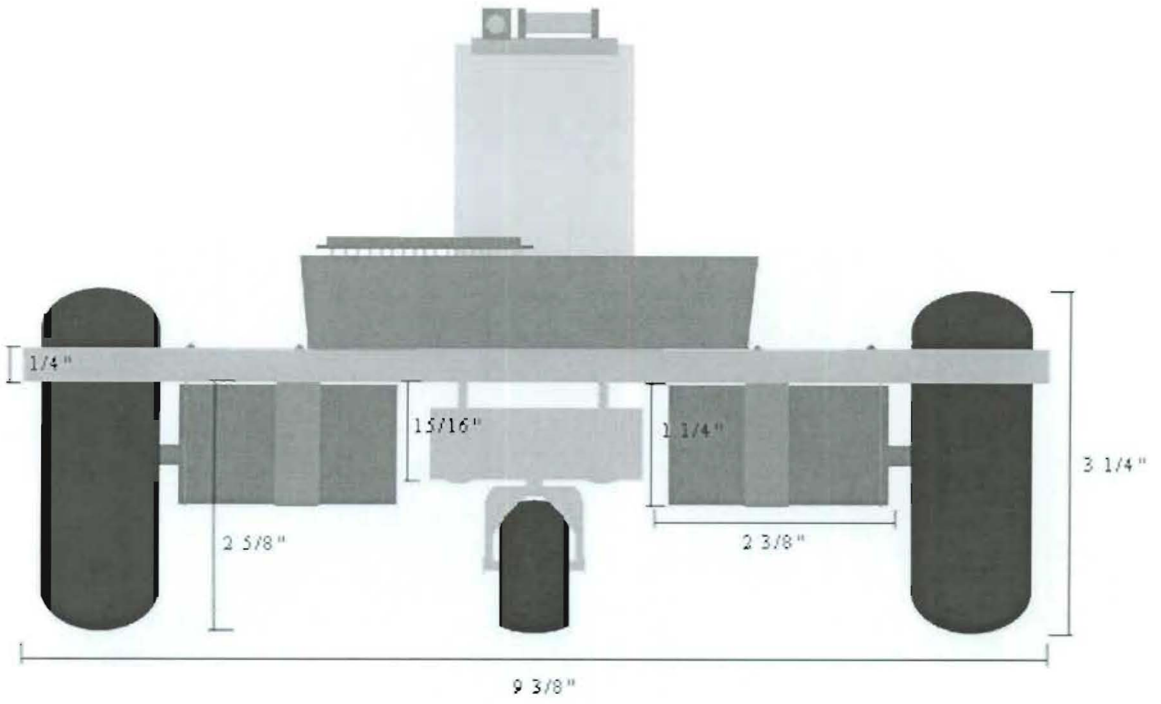
Another topic for future research entails introducing multiple robots into the environment. This includes a surprising number of problems and opportunities for more research. The first issue is that there is now one or more robot in the environment at a time, essentially creating multiple moving obstacles in what was previously a static environment. There are several possible solutions to this problem, all of which include further subsystems. There is the possibility of having more than one starting coordinate, with each corresponding to a different robot. This would involve having sections of the map assigned to a specific robot, but that implies some sort of fore-knowledge of the map. Another solution would be a scheduling system, involving a central computer assigning tasks to robots in such a way as to avoid collisions. This is a somewhat dull and imperfect solution however. A more interesting system might involve inter-robot communications, both for avoiding collisions, and perhaps also for a more intelligent and accurate system of mapping based on comparing two perspectives of the same area of a map.

Another interesting possibility with the introduction of multiple robots is giving individual robots different skills and different tasks. This could require some sort of cooperation between robots, as one robot may be assigned the task of mapping an area that it is not capable of mapping for some reason (perhaps due to different terrain in the area, or restrictions placed on a robot). This would be a particularly interesting system to apply machine learning to, to examine any emergent behavior in the relationships between the individual robots.

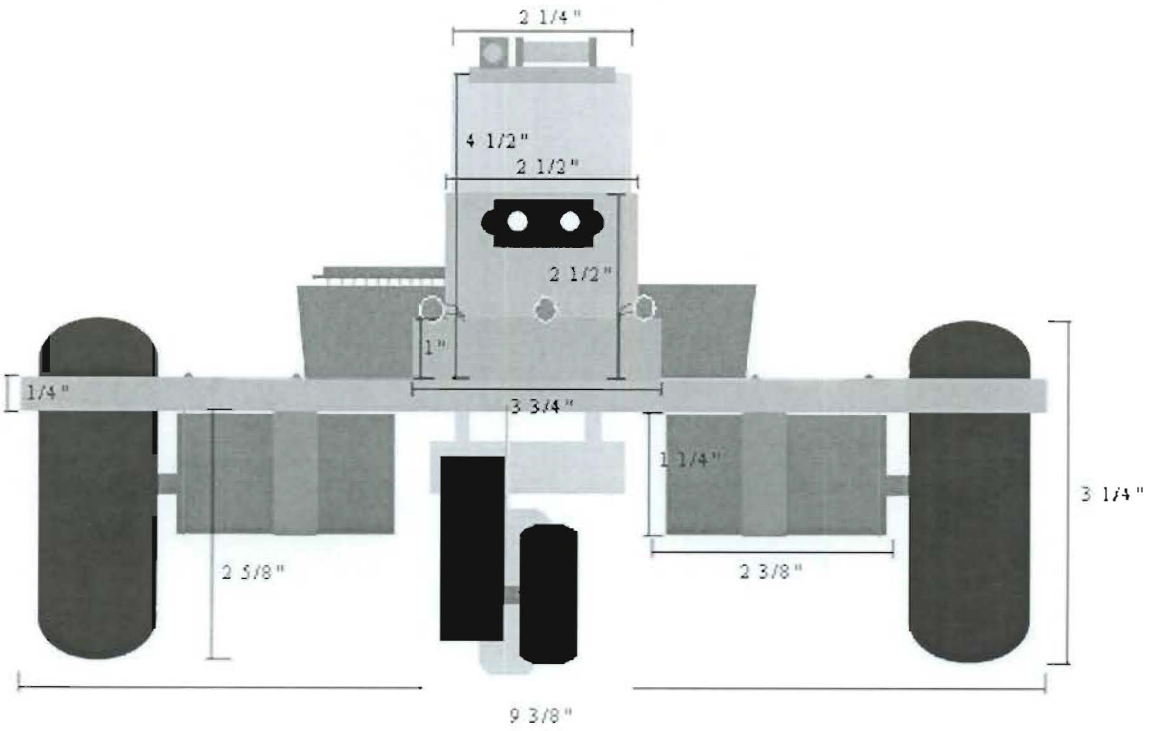
Another hardware issue to be addressed is the serial communications link between the Handy Board and the Linux computer system. It is truly irritating to have to connect and disconnect the telephone cord between the two computer systems. It would be possible to implement either an infrared communications system, or a wireless radio system between the two computers. This could also be a longer range connection, such that the robot may not always have to return to the starting coordinate to report back its findings and receive new instructions.

Beyond the hardware issues, there are several software implementation topics which could be enhanced or increased. The obvious issue is that of the path-generation system, on which I cut some corners in order to have a working system. The system could be marginally enhanced by incorporating more options for paths into this system. Additionally, I would like to increase the presence of machine learning systems in the whole system. There are many more areas where machine learning system could be incorporated, and it would be interesting to examine the effects of putting more of the system into the control of a learning system. Finally, I believe it would be quite interesting to create a successful system with one type of machine learning, and then re-implement the components of this system with different machine learning systems. Comparing the results could perhaps lead to combining the systems to create a very efficient mapping and navigation system based on multiple machine learning approaches.

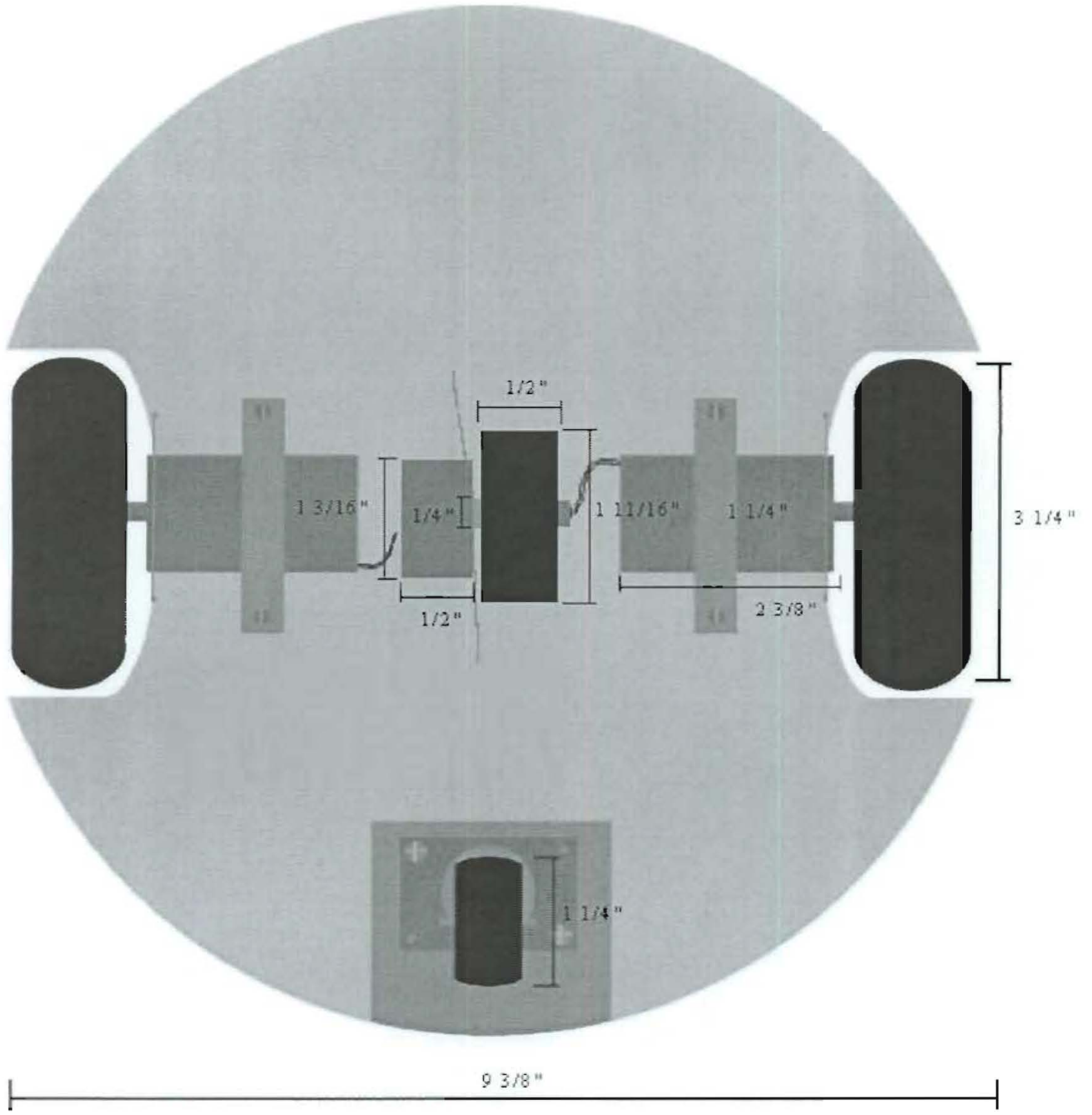
Appendix A: Robot Base



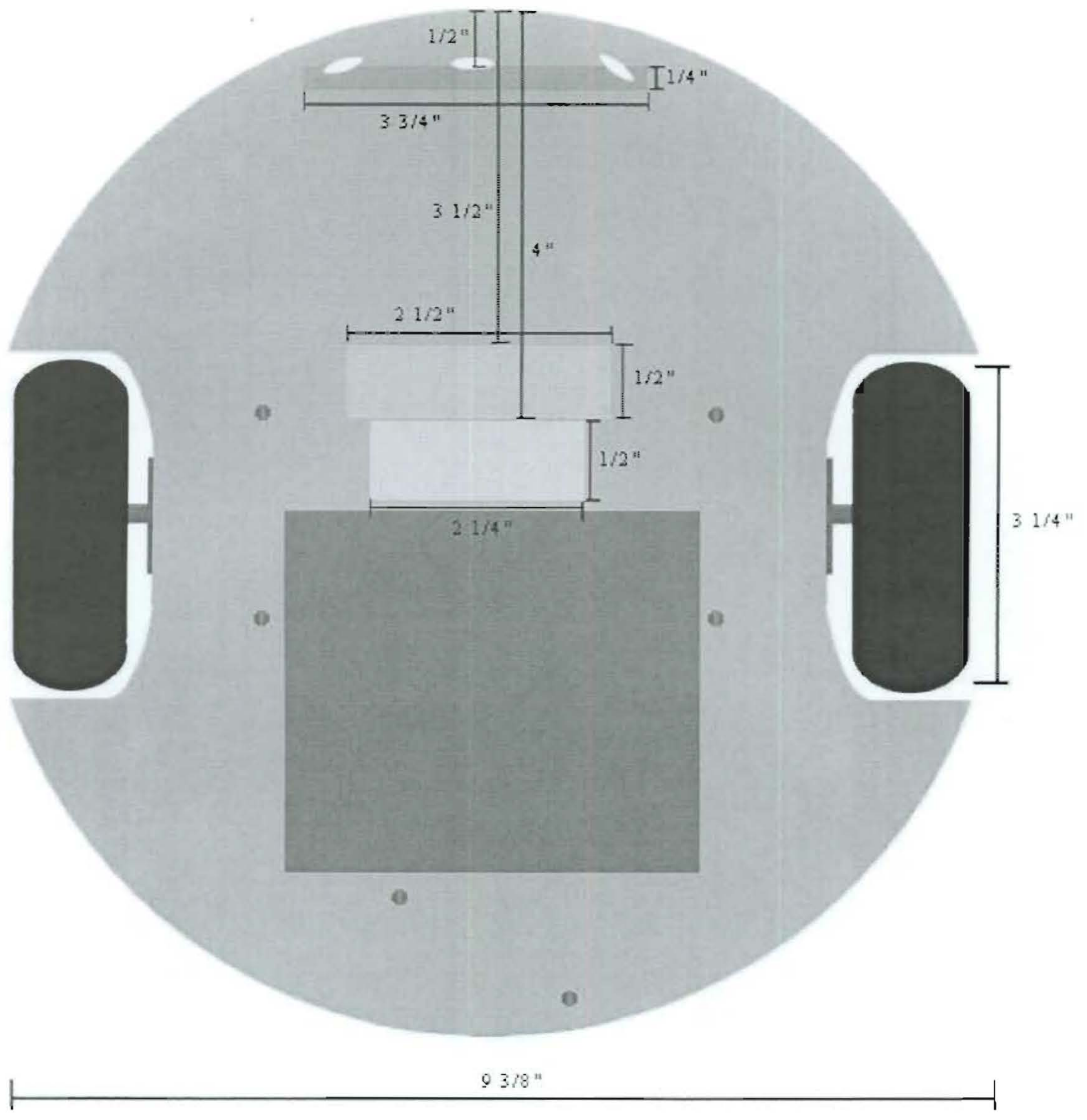
Robot Base – View from the Back



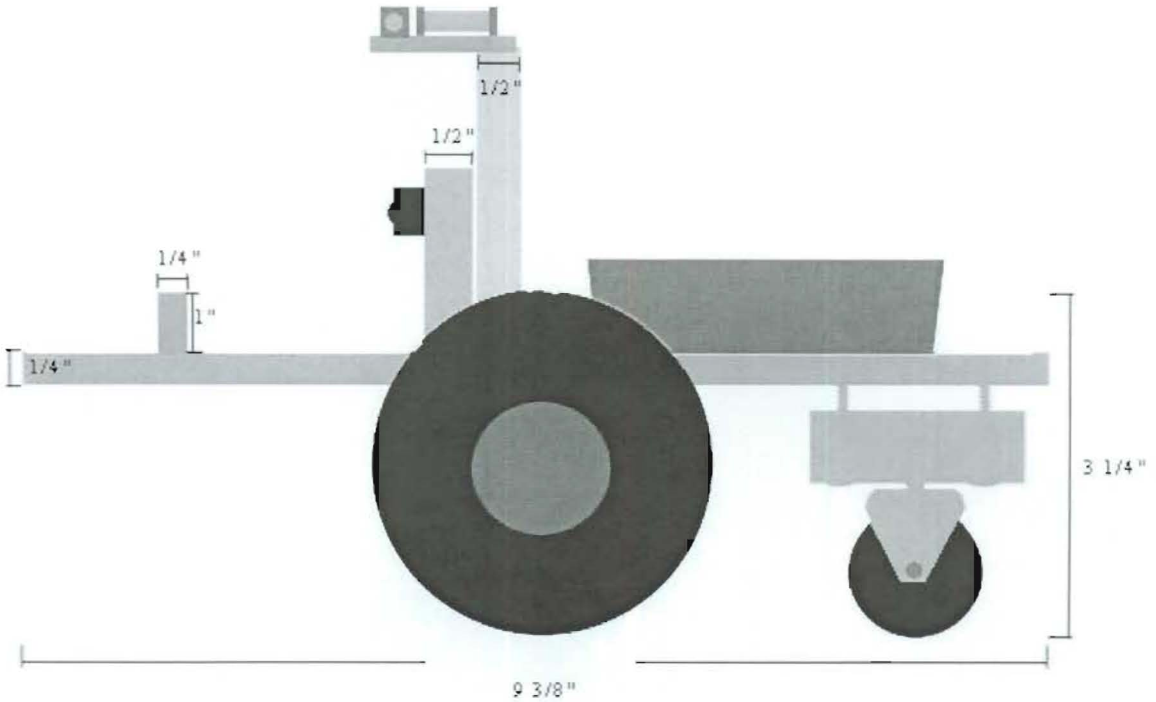
Robot Base – View from the Front



Robot Base – View from the Bottom



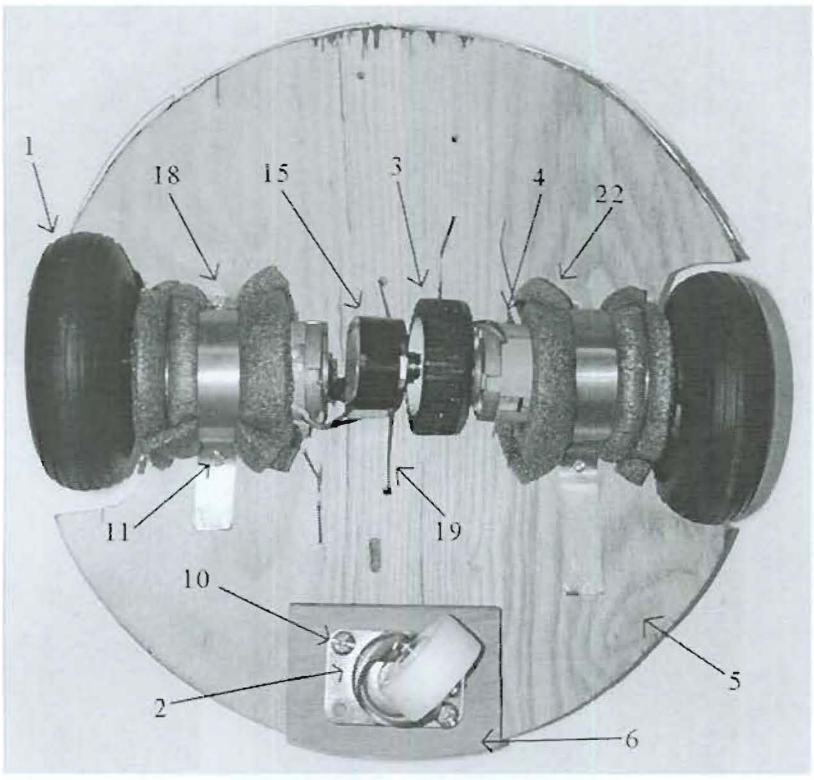
Robot Base – View from the Top

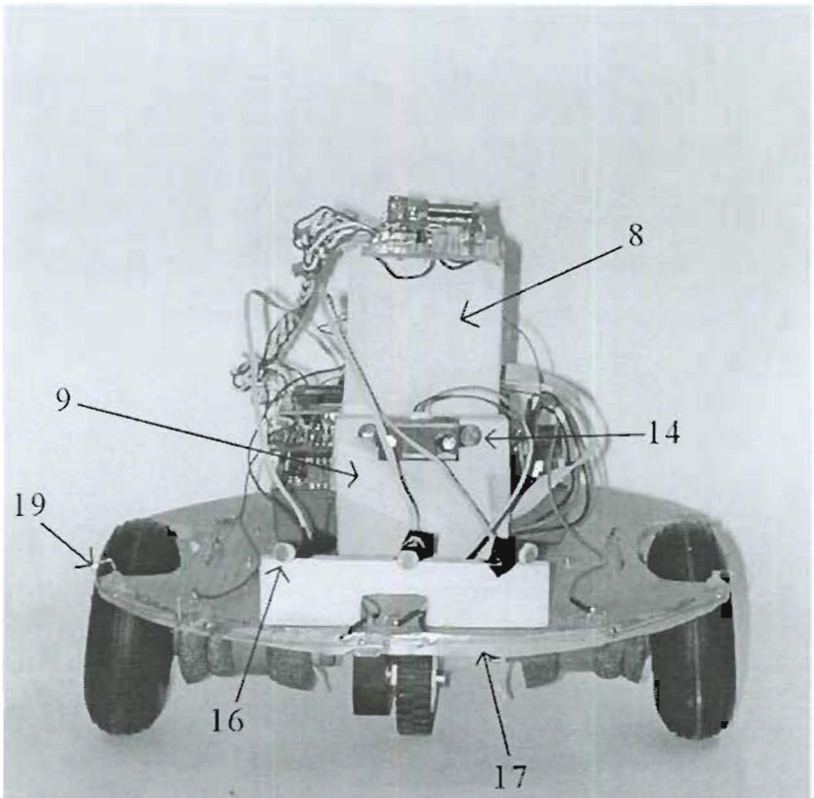


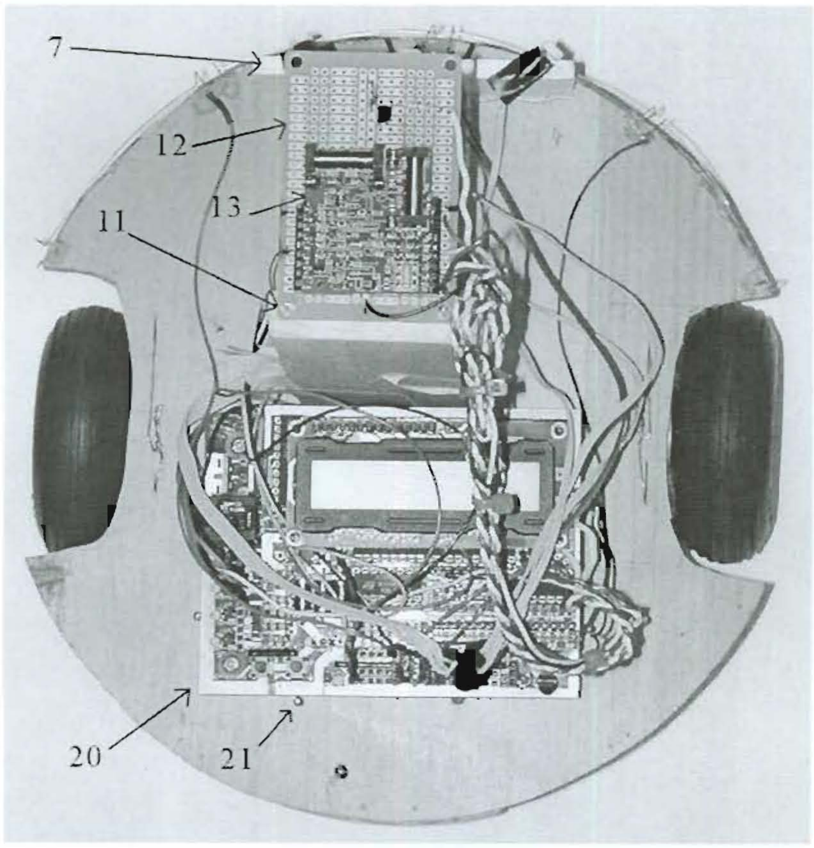
Robot Base – View from the Side

Parts List: (Corresponds to 3 Figures Following this Chart)

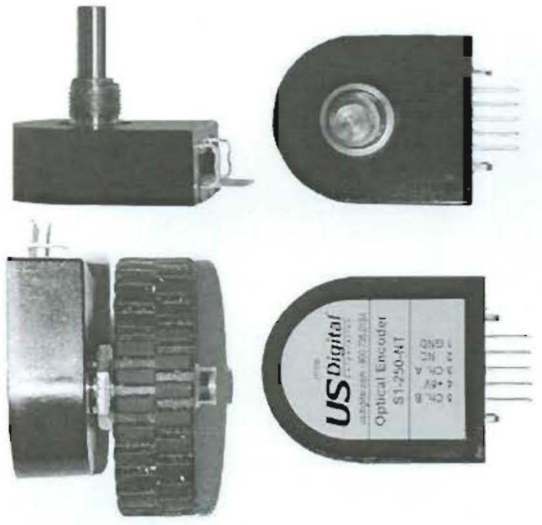
Part:	Num.	Supplier Information	Reference Num
Drive Wheels – 3 1/4"	2	DuBro	1
1 11/16" Castor Wheel	1	Hardware store	2
Lego Wheel – 1 11/16" x 1/2"	1	Lego Robotics Technology Kit	3
Geared Motors	2	Herbach & Rademan Co., 16 Roland Ave., Mt. Laurel, NJ 08054, (856)802-0422 Part #: TM90MTR1166, "25rpm 12vdc" ($27.95 ea.)	4
9 3/8" x 9 3/8" x 1/4" plywood	1	Scrap wood	5
2" x 3" x 1/2" plywood	1	Scrap wood	6
3 3/4" x 1" x 1/4" pine	1	Scrap wood	7
2 1/4" x 4 1/2" x 1/2" pine	1	Scrap wood	8
2 1/2" x 2 1/2" x 1/2" pine	1	Scrap wood	9
1" aluminum screws	2	Hardware store	10
1/2" brass screws	8	Hardware store	11
Perf Board	1	Electronics store	12
Precision Navigation Electronic Compass	1	Jameco, 1355 Shoreway Road, Belmont, CA 94001, 1-800-831-4242 part #: 126703, "sensor, magnetic compass elect." ($49.95 ea.)	13
Sharp GP2D02 Infrared Ranger	1	Acroname, Inc., PO Box 1894, Nederland, CO 80466, (303)258-3161 part #: R19-IR02 ($21.00 ea.)	14
Optical Shaft Encoder	1	US Digital, 1110 NE 34th Circle, Vancouver, WA 98682, (360)260-2468 part #: S1-250-NT, "softpot optical shaft encoder, sleeve bushing version, with no added torque, 250 CPR", ($49.95 ea.)	15
Light Sensors	3	Electronics store	16
Strips Brass Foil	2	Hardware store	17
Strips Brass	2	Hardware store	18
Brass Wire	6	Hardware store	19
Handy Board	1	Gleason Research, PO Box 1247, Arlington, MA 02474 Part #: GRHB - Mac ($299.00 ea)	20
1" Brass Nails	10	Hardware store	21
Pipe Insulation	NA	Hardware store	22
Wire	NA	Electronics store	NA
Shrink Tubing	NA	Electronics store	NA
Male/Female Strip Connectors	NA	Electronics store	NA

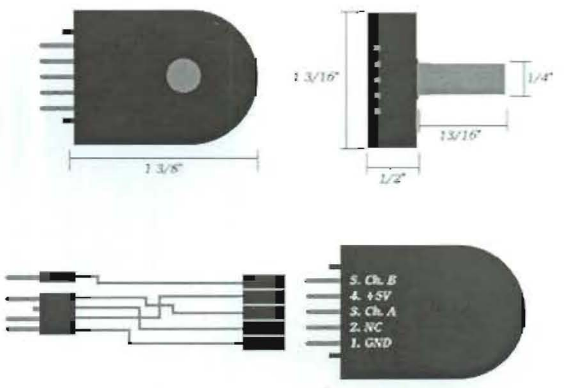

1
18
15
3
4
22
11
19
10
5
2
6


8
9
14
19
16
17


7
12
13
11
20
21

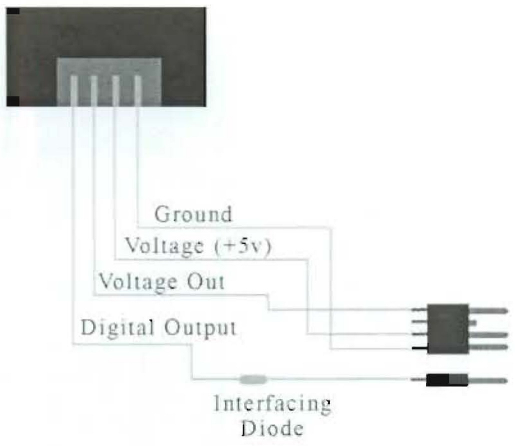
Appendix B: Hardware and Sensors


Shaft Encoder

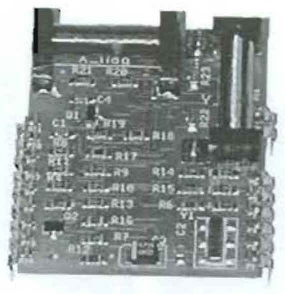


Shaft Encoder Wiring Diagrams

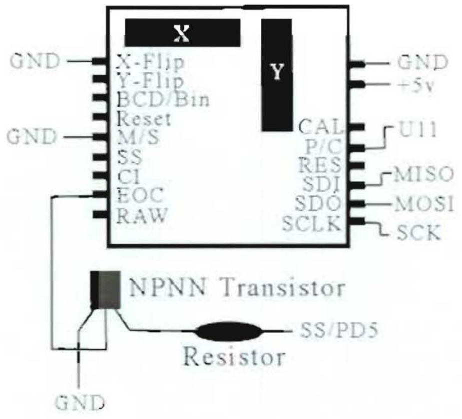
Infrared Sensor



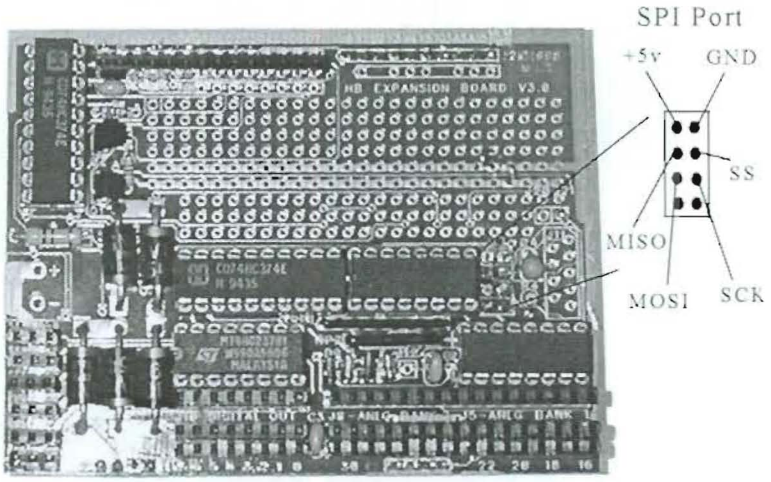
Infrared Sensor Wiring Schematic



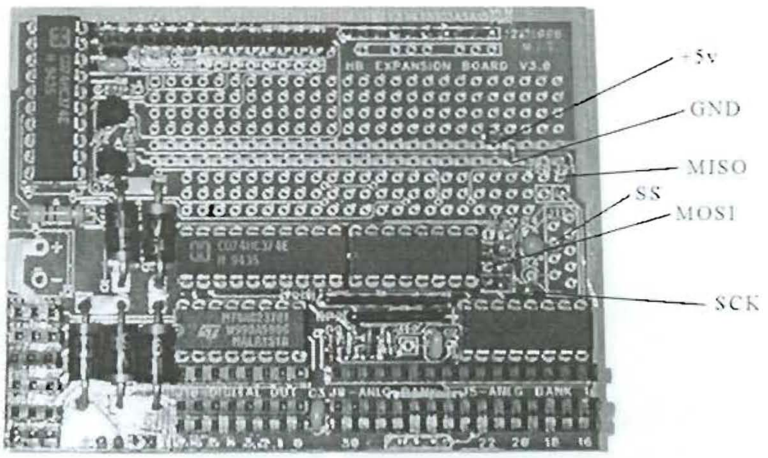
Electronic Compass



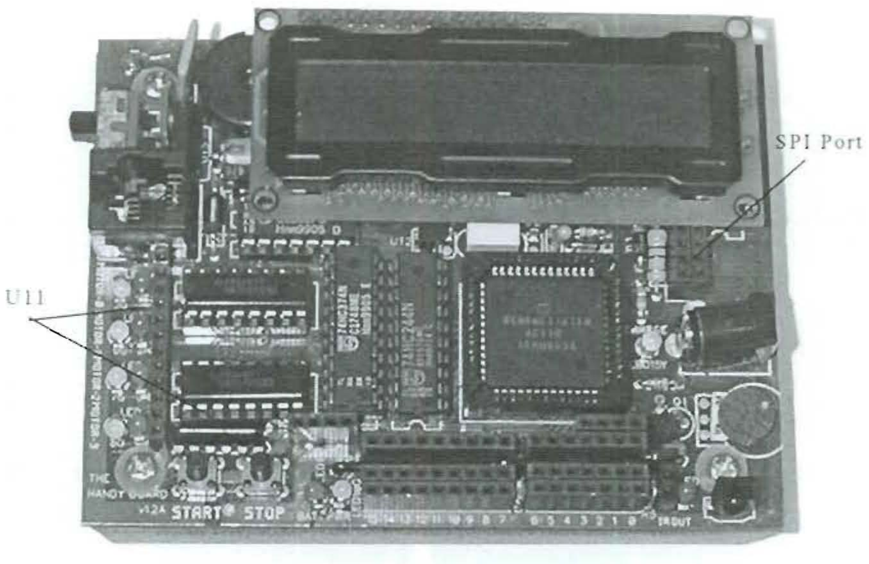
Electronic Compass Pin Out



Electronic Compass: Wiring to Expansion Board



Electronic Compass: Alternate Wiring to Expansion Board



Electronic Compass: Wiring to Handy Board

Bibliography

American Association for Artificial Intelligence; [http://www.aaai.org]

Braitenberg, Valentino, "Vehicles, Experiments in Synthetic Psychology", MIT Press, Cambridge, Mass., 1984

Brusehaver, Tom; Handy Board contributed code repository [http://el.www.media.mit.edu/groups/el/projects/handy-board/software/contrib/tomb/] (Brushaver's code, available from the Handy Board contributed code repository)

Cheeseman, Peter, et. al., "AutoClass: A Bayesian Classification System", from *Proceedings of the Fifth International Conference on Machine Learning*, 1998

Congdon, Clare, "A Comparison of Genetic Algorithms and Other Machine Learning Systems on a Complex Classification Task from Common Disease Research", PhD thesis, University of Michigan, Department of Electrical Engineering and Computer Science, February 1995.

Detroit, Barry, [http://reality.sgi.com/barry_detroit/GP2D02_1.html] (Detroit's code linked from the Handy Board code repository)

Drushel, Richard F.; Handy Board contributed code repository; [http://el.www.media.mit.edu/groups/el/projects/handy-board/software/contrib/drushel/serialio.c] (Drushel's code, available from the Handy Board contributed code repository)

The Expansion Board web site; [http://el.www.media.mit.edu/groups/el/Projects/handy-board/hbexp30/] (Information available from this site was used as instructions for constructing the Expansion Board)

Fisher, Douglas, A., "Knowledge Acquistion Via Incremental Conceptual Clustering", *Machine Learning*, 2: 139-172, Kluwer Academic Publishers; Boston, 1987

The Handy Board code repository; [http://el.www.media.mit.edu/groups/el/projects/handy-board/software/encoders.html] (Standard code for the Handy Board, available from the Handy Board software repository)

The Handy Board Mailing List, [http://www.lugnet.com/robotics/handyboard/] (Repository of messages from the Handy Board Mailing List)

Heidel, Thomas, [theidel@advis.de], personal communication, Monday, November 8, 1999

IS Robotics web site; [http://www.isr.com]

Knotts, Ryan; Nourbakhsh, Illah; Morris, Robert, "NaviGates: A Benchmark for Indoor Navigation",
[http://www.ri.cmu.edu/pub_files/pub1/knotts_ryan_1998_1/knotts_ryan_1998_1.pdf]

Kunz, Clayton; Willeke, Thomas; Nourbakhsh, "Automatic Mapping of Dynamic Office Environments",
[http://www.ri.cmu.edu/pub_files/pub1/kunz_clayton_1999_1/kunz_clayton_1999_1.pdf]

Meeden, Lisa, "An Incremental Approach to Developing Intelligent Neural Network Controllers for Robots", appeared in *IEEE Transactions of Systems, Man, and Cybernetics, Part B: Cybernetics*, June 1996, Volume 26, Number 3, pages 474-485

Meeden, Lisa; Kumar, Deepak, "Trends in Evolutionary Robotics", appeared in *Soft Computing for Intelligent Robotic Systems*, edited by L.C. Jain and T. Fukada, Physica-Verlag, New York, NY, 215-233, 1998

NASA, Mars Pathfinder web site; [http://mpfwww.jpl.nasa.gov/default.html]

The Neural Network FAQ, [ftp://ftp.sas.com/neural/FAQ.html] (Comprehensive resource and source on neural networks)

Mitchell, Tom, M., "Machine Learning", McGraw Hill, New York, 1997
[http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ftp/ml-examples.html] (Neural network code accompanying textbook)

Shu-pui, Patrick Ko; [http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/systems/bpnn/]
(Neural net code from this site used as an introduction to neural networks)