A Flexible and Innovative Platform for Autonomous Mobile Robots

by

Jessica Anne Howe

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

[February 2004]
January 2004

Author .....
Department of Electrical Engineering and Computer Science
January 29, 2004

Certified by .....
Rodney A. Brooks
Professor, Department of Electrical Engineering and Computer Science
Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Flexible and Innovative Platform for Autonomous Mobile Robots

by

Jessica Anne Howe

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The development of the CREAL programming language and the STACK hardware platform by the members of the CSAIL Living Machines research group has lead to a foundation upon which roboticists may build and control autonomous mobile robots. This platform, however, has been used only within this research group on a minimal number of projects. As is, there has been no published discussion on the application of CREAL and the STACK to various types of control architectures commonly used in controlling autonomous mobile robots, or any personal accounts of the successes or failures of using such a system on a hand-built robot.

In this thesis I focus on these two points. I go into depth on the use and expansion of CREAL to support multiple control architectures, as well as a personal account of the construction and use of a robot that uses these architectures and hardware to accomplish various tasks. The work to be undertaken will describe the process of design, construction, debugging and implementation of a hand-built robot that performs example tasks that are similar in nature to tasks commonly performed by autonomous mobile robots within the robotic research community.

Currently, CREAL does not provide any abstract framework to facilitate implementation of neural net architectures by users. The work described in this thesis includes a set of macros that expand the CREAL language to allow user-friendly creation of neural nets. This abstraction framework is then put into use in an implementation that uses the neural net tools developed in order to achieve a fixed goal.

The second architecture to be discussed is that of subsumption, an architecture that is extremely well suited to be implemented in CREAL. To demonstrate the suitability of CREAL, a subsumption implementation will be described that performs a complex robot behavior An account will be given of creating a subsumption base behavior and passing through multiple stages that increment the behavioral capabilities of the robot. This will include a description at each stage of how the subsumption architecture is expanded to bring the behavior of the robot closer toward the goal behavior.

Through the implementation of the above tasks I hope to show to be true what we have claimed: that the platform consisting of the CREAL programming language and the STACK hardware is an effective, flexible, powerful and desirable platform to use in designing autonomous mobile robots.

Thesis Supervisor: Rodney A. Brooks
Title: Professor, Department of Electrical Engineering and Computer Science

Acknowledgments

Holy bejesus, where do I start? Mom. Thank you so much for always having the answers. Without you I really don't think I would have been able to make it to this point. You are amazing.

Rod and Una-may, thank you so much for your guidance and help along the way. I am forever honored to have had the opportunity to work with you.

Da, Amila, Kentus. Whoa, huh? Yeah, whoa. Thank you so much for being there and believing in me. Lil' Bit. Your sister thinks you're the best and biggest bit in the world. Hi to Monkey Pants too.

JessB, Lijin, Kathleen, thank you for keeping me sane during those long long long nights working and helping me in so many ways. Your advice was always taken to heart and helped me see the light at the end of the tunnel. Seriously. Actually that goes for pretty much all you anti-socialites up here on 9. Beep, Maddog, the damn fine officemates. Eduardito, Mad Man CK, the Lovely Miss Paulina, Aaron, Jeff, Pfitz, Artur, Martino Martin. A big fat thanks goes to all of you.

Grandmas, Grandpas, Aunts, Uncles, Cousins... your love was felt from across the country and I'm blessed that you are all part of my life.

Machuga!!! Bedonk. The Bun! Bun. Miss Maggie! I love you. Silly Adam. You too. FLOORPI!!!!!

To everyone else out there that even had a teeny tiny part in helping me get to where I am a sincere thank you from the bottom of my heart.

Contents

List of Figures

Chapter 1

Introduction

When planning the construction of an autonomous mobile robot there are many decisions that must be made along the way. Choices such as deciding what the robot is intended to do, how best to achieve that task, which programming language you intend to use, which processing platform you should use, how the control architecture should be designed, and the morphological structure and appearance of the robot all must be considered in great detail. There are a lot of questions that must be answered. Additionally, there are a multitude of different combinations of answers to these above questions that may lead the roboticist toward their desired goals. There is no 'right' combination of answers to these questions, but some choices may be made that will make the planning, construction, programming, and debugging of this robot easier for the builder, as well as influence the degree of success achieved in the project. When making these decisions it is wise to look at the implications of each possible choice in terms of the difficulty in design, planning and upkeep, the robustness that the system will have as a result, the modularity that the system components will have, the ability to easily perform changes or upgrades, and the reusability of the architectural design platform.

It is the presence of these numerous choice factors that shows the need for a single easy-to-use, robust, modular, reusable architectural design platform for autonomous mobile robots. Over the past year the Living Machines group at MIT's former Artificial Intelligence Laboratory (now the Computer Science and Artificial Intelligence

Laboratory) have been designing a new platform for autonomous mobile robots that is intended to meet all of the needs described above. This system is flexible, stable, easy to use, and has been designed for creative and flexible manipulation by users. My research demonstrates the constraints and challenges faced by scientists in the field and show that within our system these goals have indeed been met.

The system is composed of two distinct parts: a programming language called CREAL and a hardware platform that it may be run upon, commonly referred to as the STACK. The programming language caters specifically to behavior based autonomous robots and there is great flexibility offered in both the types and numbers of peripherals that may be used with this system. The research in this thesis demonstrates that the system is stable and robust through the construction and use of a hand built robot that performs various tasks. As described within this thesis, the system has also been expanded for greater flexibility and ease of use when designing and building. My own work has focused on the creation of an abstract framework so that there is a simple way to implement various control architectures within our system.

When looking at the current state of autonomous mobile robotics one sees a number of interesting trends. The first is that there are many control architectures used, but often these control architectures are specific to either one particular task, or one particular person (usually the person that designed the architecture). Examples of such architectures include the Action Selection Architecture designed by Pattie Maes [28], the DAMN Architecture designed by Rosenblatt [38], the Circuit Architecture designed by Kaelbling and Rosenschein [39], and the Motor Schemas Architecture used by Arkin [3].

There are also a limited set of architectures that are commonly used by many people on a range of robots. One such architecture is Subsumption, a behavior based architecture designed by Rodney Brooks [9]. The basis of the subsumption architecture is to have immediate reflex-like reactions to sensory input as it is read. Independent modules that represent outwardly visible behaviors run in parallel, each one acting when appropriate input values are received. Further stratification is pos-

sible by the low-level and the high-level behaviors. Low-level behaviors essentially act at all times while high-level behaviors, whose input triggers are tapped much less frequently, have the power to override, or subsume, the outputs of the low-level behaviors. This architecture has been used to control robots such as autonomous walking hexapods [7], a map-building wheeled robot [32], a robotic tour guide [20], and an autonomous vacuum cleaner [22].

Another approach that is commonly used to control autonomous mobile robots is to use a Neural Network control architecture. Neural networks are computational structures that model the actions and structure of neurons in the brain and the passing of signals from one neuron to its neighboring neurons. Neural nets are similar in structure: sensory input flows into nodes of the network and weighted signals are passed on to neighboring nodes. This signal passing continues from input nodes, through the neural net, and finally to the output nodes. The signals retrieved from the output nodes are then sent to the output actuators. Neural Nets have been used to control robots whose tasks have ranged from navigation [36] and manipulation [40] to interactive group behavior [2] [35].

Unfortunately, choosing a control architecture for a robot is one of the many necessary tasks facing the roboticist. From this follows the questions of which type of processing hardware to use, which programming language will be best to accurately implement this architecture, and how the body of the robot is to be built to best achieve the desired goal. The work presented in this thesis explains the CREAL and STACK platform described above and its use in implementing various control architectures on a hand-built robot.

1.1 Problem Overview

The development of CREAL and the STACK has lead to a foundation upon which roboticists may build and control autonomous mobile robots. This platform, however, has been used only within the Living Machines research group that created it on a minimal number of projects. This development platform has great potential for use

in many other types of robotic projects.

As is, there has been no published discussion on the application of CREAL and the STACK to various types of control architectures commonly used in controlling autonomous mobile robots, or any personal accounts of the successes or failures of using such a system on a hand-built robot. In this thesis I propose to focus on those two points: go into depth on the use and expansion of CREAL to support multiple control architectures, as well as a personal account of the construction and use of a robot that uses these architectures and hardware to accomplish various tasks. The work to be undertaken will describe the process of design, construction, debugging and implementation of a hand-built robot that performs example tasks that are similar in nature to tasks commonly performed by autonomous mobile robots within the robotic research community.

Figure 1-1: The autonomous mobile robot described within this thesis and used as the base for all experiments conducted herein.

The first architecture to be approached is that of neural networks, a control system commonly used in the control of autonomous mobile robots [36]. Currently, CREAL does not provide any abstract framework to facilitate implementation of neural net architectures by users. Just as high level programming languages and graphical tools exist so users do not have to program everything in assembly code, a set of abstraction tools for neural nets within CREAL will assist users in implementing this architecture.

The work described in this thesis includes a set of macros that expand the CREAL language to allow user-friendly creation of neural nets. Without these macros the

user would have to create the node structure, node communication protocol, data passing, and all other details from scratch to implement a neural net within CREAL. This abstraction provides to the user access to these vital components of neural nets through a set of macros designed to automatically create the lengthy CREAL code that is required to create such a control system.

This abstraction framework is then put into use in an implementation that uses the neural net tools developed in order to achieve a fixed goal. The behavior aimed for is that of wall following, a task that many robots need to perform in order to achieve their greater goals [40] [2] [35]. Wall following is especially prevalent in navigation problems[36] or in situations that involve a mobile robot in an environment meant for humans, such as in an office setting [6].

The second architecture is that of subsumption [9]. A discussion will take place that will describe how CREAL is extremely well suited to implement a subsumption architecture within a mobile robot. As CREAL so adequately implements subsumption, no separate abstraction tools are needed to implement a subsumption architecture.

To demonstrate the suitability of CREAL, a subsumption implementation will be described that performs a much more complex robot behavior than wall following. Although wall following will be involved, the greater goals of the system include navigation in search of people, with which the robot is to interact. An account will be given of creating a subsumption base behavior and passing through multiple stages that increment the behavioral capabilities of the robot. This will include a description at each stage of how the subsumption architecture is expanded to bring the behavior of the robot closer toward the goal behavior.

Through the implementation of the above tasks I hope to show to be true what we have claimed: that the platform consisting of the CREAL programming language and the STACK hardware is an effective, flexible, powerful and desirable platform to use in designing autonomous mobile robots.

1.2 Background and Related Work

As stated above, there are numerous decisions to be made when building an autonomous mobile robot. One option may be for scientists to build the entire robot from scratch, designing the control system and building the body. Alternately, mobile roboticists could buy a pre-built pre-programmed robot that requires only minimal user intervention to 'build'. There are, of course, advantages and disadvantages to each of these options. In building the entire robot yourself you have total flexibility in design and get very familiar with the hardware that is in use, but it takes much longer to implement and could leave the builder with a faulty machine, particularly if the user has limited skills and experience in this area. A pre-built machine will take almost no time to set up and is almost guaranteed to work, but the user is limited to the parts that the manufacturer provided.

Most robot builders fall somewhere in between the two extremes. Some use either a pre-built body and hardware platform that requires only programming, but are constrained in the types of control system architectures to choose from. Others commonly take a more hands-on approach and use the pre-built control hardware system that best accommodates their chosen control architecture model, and fully design the body of the robot, but are often limited to marketed body parts. Due to the constraints of cost, labor and time, few people build both the hardware and body from scratch.

The following section will briefly describe the commercial platforms and programming languages available to use in the construction and control of an autonomous mobile robot, as well as the target audience and the advantages and disadvantages of each system. In the following two chapters this set of commonly used platforms and languages will be compared to the CREAL programming language and the STACK hardware system.

Note that in this paper the phrase 'robot' will refer strictly to autonomous mobile robots, not remotely operated or pre-programmed robots which are commonly found in factories. Additionally, I will be limiting my discussion to robots that are 'small',

excluding systems as large as an autonomous vehicle and including systems that are within the size range of being lifted by a person.

1.2.1 Commercial Robotic Platforms

There are many commercial products that people may buy to help them along in their robot-building adventure. Some robots are sold complete with on-board sensors and actuators, as well as the hardware with which to control the robot. Others are sold as either the control hardware alone with no standard sensors, actuators or morphology, or the robot body alone, lacking the computation abilities necessary to control such a robot.

Lego Mindstorm

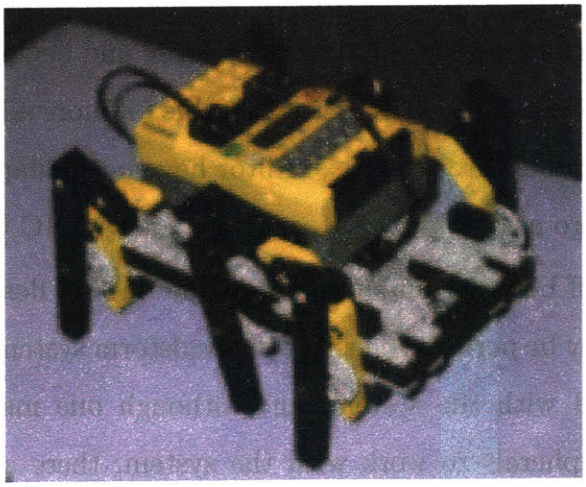


Figure 1-2: A hexapod robot built with a Lego Mindstorm.

The Lego Mindstorm is an extremely popular platform that is both inexpensive and especially suitable for beginners and amateurs [34]. It provides a simple graphical programming interface and facilitates the interaction of sensors, actuators and the control system. Additionally, the majority of robot bodies built using the Mindstorm are built with Legos, the common children's building block. This gives the builder flexibility in the style and construction of body morphologies. The Mindstorm package

comes complete with sensors and wheels that may be used with the control and easily attached to Lego blocks.

The Mindstorm RCX, also known as the Lego *Brick* is the computational device that controls the actions of the robot [33]. It interfaces seamlessly with the sensors and actuators provided with the kit, and allows for simple programming accessing these accessories. The standard Mindstorm kit comes with simple rotation motors that can control things such as the wheels of a mobile car. The speed of the motors are controlled through programming of the Brick. The Mindstorm kit also comes with various basic sensors, such as bump sensors, switches, and light sensors, and has more advanced extension peripherals such as cameras for simple vision work. Additionally, there are many non-standard sensors and actuators that can be retrofitted to properly interact with the Mindstorm as well, such as accelerometers or gyroscopes.

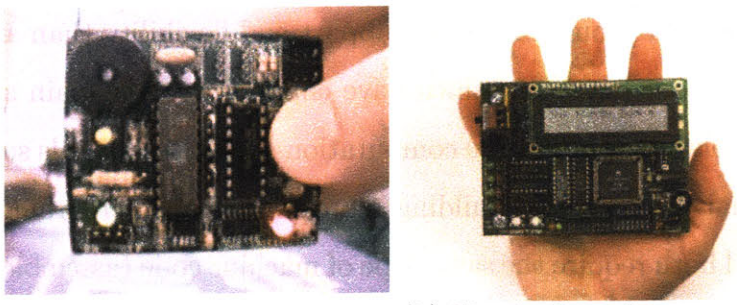
RCX Code, a visual drag-and-drop representation of a limited scope language, is provided with the Mindstorm kit and is the default language for programming the Mindstorm. This language is suitable for children and beginners but its graphical approach and limited capabilities make it unsuitable for more advanced programmers. Languages programmable through a text editor have been adapted to be compatible with the required Lego assembly format, such as Not Quite C [4] and Java [26].

There is a tradeoff between simplicity of use and lack of flexibility in the types of computation that may be performed using the Mindstorm system. Minimal peripheral devices may be used with the system, and although one may adapt many types of non-standard peripherals to work with the system, there are limitations on the extent to which this ability applies. Similarly, the processor within the brick is not very powerful. It is powerful enough to control a robot with basic computation and minimal numbers of peripherals but is limited beyond that.

Mindstorms are extensively used as an introduction platform to teach both children [34] and adults [14] the basics of robotics. Very few professional roboticists use Mindstorms for research, but this platform has been used for research projects [15] as advanced as robotic soccer [27]. In using Mindstorms people learn aspects of morphological design, control planning, debugging, and the overall process of building an

autonomous robot.

Crickets and Handy Boards



(a) Cricket (b) Handy Board

Figure 1-3: A Cricket and a Handy Board, two technologies created at the MIT Media Lab that are often used to control autonomous mobile robots.

The Handy Board [29] and Cricket [30] are control systems designed at the MIT Media Lab that allow more flexibility than the Lego Mindstorm but are also fairly accessible to the beginner roboticist. All of the same sensors and actuators that may be used with the Mindstorm may also be used with these platforms, as well as more complicated peripherals. These systems support a much greater number of peripherals than the Mindstorm.

Unlike the Lego Mindstorm, there is no standard design strategy for the bodies of robots using these control systems. The Cricket and Handy Board are hardware packages used for the control of the robot and do not come with a standard building block such as the Lego or a robot body. Of course, people may use Legos in conjunction with these control systems, but often choose to build a more advanced robot body.

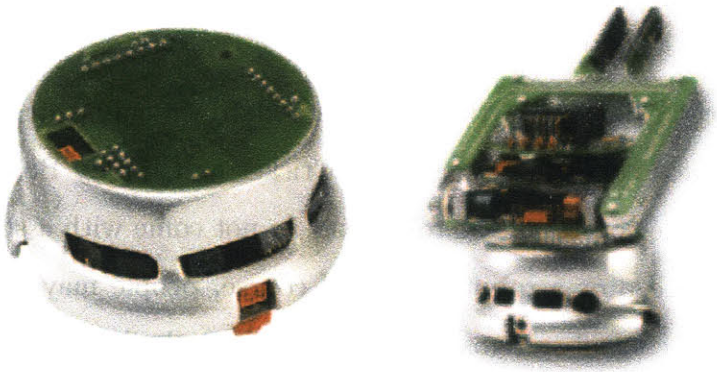
The Cricket is a small device that allows only two sensors and two actuators to be connected, but is fairly inexpensive, is very easily used and is quite small. It can be programmed using using the language Not Quite C [4] as well as through Cricket Logo [31], a graphical block-style programming interface as per the Mindstorm. The Cricket is exceedingly simple to program and easily interacts with peripheral devices, but lacks the processing power that would be necessary to control more than these

basic sensors and actuators.

The Handy Board offers much more flexibility in the quantity of peripherals that may be used and the computation power available, but is also more expensive and bigger than either the Cricket or the Mindstorm. Nonetheless it is often used by university students who would like more support and flexibility than are offered with the other two mentioned system (which have children as their main audience). The Handy Board is able to support more computation-heavy peripherals such as cameras, and the additional memory aids building more complicated programs. The Cricket and Handy Board both require the same type of machine code (assembled code written in a higher-level language), so an upgrade of platforms or a communication protocol between two different platforms is possible.

Crickets and Handy Boards have a fairly wide audience. They are accessible enough for children to use and powerful enough for university students [21] and researchers [13] to use.

Kheperas



(a) (b)

Figure 1-4: (a) A Khepera base system. (b) A Khepera base with extension boards and an arm.

The Khepera is an extremely popular pre-built commercial robot commonly used within the computer science research community [24]. It is very small (Diameter: 70 mm, Height: 30 mm [24]) and is a fully functioning programmable mobile robot. It

contains wheels that are used as its primary source of locomotion, has many built in sensors, and has room for attachment of other peripherals. It is flexible in the types of peripherals and computation it supports, but its size limits the types of environments in which it may be used and the sizes of peripherals that may be attached to it. It is usually both too expensive (in the $2000 range) and too complex for students to make beginner usage practical.

The platform comes equipped with eight infrared and ambient light sensors with up to 100mm range [24], and has the capabilities of supporting many other types of sensors [24]. It is also flexible and powerful enough to support cameras running vision algorithms and arms used for object manipulation [18]. The computation device that is built into the khepera is a Motorola 68331, running at 25MHz [24]. It can also be attached to an external computer for greater real-time computation power, but this requires either a tether or a wireless communication protocol to be set up.

Kheperas may be programmed in a variety of programming languages using plug-ins provided by K-Team, the Khepera manufacturers. Supported languages include GNU C, Matlab and SysQuake, as well as WEBOTS, a simulation environment that models the dynamics of a Khepera in an environment with user-specified objects [25]. Kheperas also have multiple hardware extension packages that allow an increase in the number of peripheral sensors and actuators, a few compatible cameras, and a gripper arm built specifically for the Khepera [24].

Various control architectures are used in programming the Khepera as well. Traditional AI sense-plan-act strategies usually require quite a bit of computation power and are often not chosen due to the restraints of the system. Neural networks are commonly used, and have been used to perform a variety of tasks such as navigation [36] and manipulation[40]. Evolutionary algorithms have been used to calculate the appropriate weights of the neural nets as well [16] [12]. In most cases the evolutionary process is done in simulation on a more computationally powerful machine and later downloaded to the Kephera, but there has also been work in evolving the system parameters through testing on the actual robot as opposed to in simulation [36].

Large 'Garbage Can' Robots

Many commercial research platforms are available for building various mobile robots. Two examples are shown in Figure 1-5. They are generally quite large and expensive, limiting the pool of users (usually) to research and academia. These mobile bases usually use a multiple wheel synchronous drive system. They come with onboard power and processing modules and provide both hardware and software systems for motor as well as sensor (sonar, IR, tactile, etc) control.

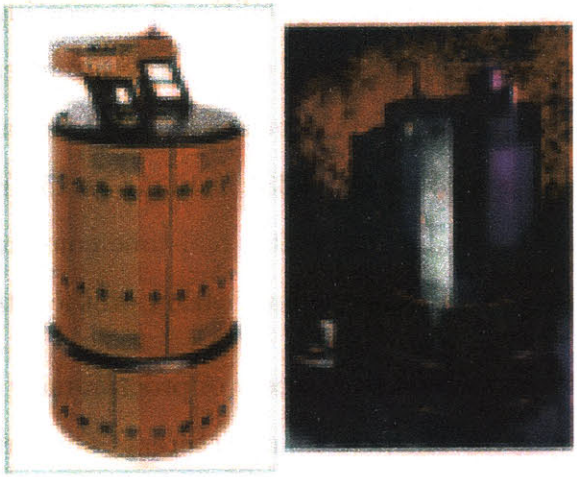


Figure 1-5: The left robot is the B21r from IRobot and the right robot is Nomad 200 from Nomadic Technology

1.2.2 Programming Languages Commonly Used for Mobile Robots

As would be expected, there are a wide range of programming languages used to control autonomous mobile robots. Some perform better than others on given hardware platforms, and some platforms only support a single language. In robots, particularly those of the quality used for research as opposed to those aimed primarily at an amateur audience, there are often multiple language environments that are compatible with the hardware of the system. The control architecture that is chosen to accomplish the desired task is often extremely influential in the set of programming languages that are available.

Graphical programming interfaces are aimed at children and are used with the most basic robotic platforms. The Lego Mindstorm kit comes complete with RCX Code, a visual interface specifically designed for the Mindstorm [33], and Logo is a language that has had a visual programming tool adapted for use on Crickets [31]. Most robots programmed with these languages are fairly simple based on the fact that these languages do not support programs of much complexity (such as ones that contain variables.) Due to the fact that most of these basic platforms may be programmed by higher-level programming languages, more advanced roboticists tend to use a non-graphical language to program these hardware platforms.

Programming languages such as C, C++ and Java are popular languages to use in programming robots of all levels of complexity, from Java on the Mindstorm [26] and Not Quite C [4] on the Cricket, to GNU C on the Khepera [25] and many languages on larger robots used for professional research. There exist multiple problems with using these languages to design distributed control architectures such as neural nets or subsumption. One major flaw is that it is difficult to control the timing of many concurrent threads of computation necessary for these architectures that are running within the program. Timing management must be explicitly coded into the program by the user to ensure that each thread of computation gets an even share of CPU usage and to make the multiple nodes or modules really run concurrently. These languages are also quite large in that compilation into assembly code (that will eventually be downloaded to the processing hardware) will often produce extremely lengthy files that are difficult to run on platforms with little memory or limited computational abilities.

Lisp, Scheme and related programming languages are also commonly used to program autonomous mobile robots as well [37]. These languages are much more lightweight than languages such as Java, but there are still difficulties in controlling these languages with the precision and flexibility one often needs. There have been many derivatives of these languages, such as L [11] and Behavior Language [8].

1.3 Organization of Thesis by Chapter

- **Chapter 1: Introduction**

- **Chapter 2: STACK and Robot Hardware**

  Chapter 2 contains two large sections. The first section is an in-depth look at the hardware STACK: the overall characteristics of the STACK as a whole and details of how its many parts work together, a description of each of the individual components in the stack, their function and usage details, and the overall capabilities and limitations of this hardware. The second section is a detailed description of the morphology of the robot constructed for this thesis, including information on the sensors, actuators, power supplies and STACK components on board.

- **Chapter 3: CREAL**

  Chapter 3 gives an overview of the programming language CREAL. This language is used to program the robot in all experiments within this thesis. The interaction between CREAL and the STACK hardware will be described, as well as a comparison between CREAL and other languages that could be used to program autonomous mobile robots.

- **Chapter 4: A Neural Net Expansion to CREAL**

  Chapter 4 gives an overview of the commonly used neural net architecture and describes the abstraction framework that was built upon CREAL. This framework allows users to easily access the computation capabilities of this architecture.

- **Chapter 5: A Neural Net Implementation**

  Chapter 5 describes an implementation of a neural net architecture upon the robot. Included in this description are details of the use of the CREAL neural net abstraction framework within this implementation and an overview of the steps involved in working towards the goal.

- **Chapter 6: A Subsumption Implementation**

  Chapter 6 gives an overview of the subsumption architecture in the field of autonomous mobile robots and describes an implementation of this architecture using CREAL and the robot. This implementation shows the capabilities of CREAL in supporting a more complicated behavioral task.

- **Chapter 7: Future Work**

Chapter 2

Stack and Robot Hardware

2.1 The STACK

Over the past year the Living Machines research group within the AI Lab has been designing an embedded processing system for robotic applications. This hardware system, referred to as the STACK, was inspired by a wide array of expandable architecture alternatives for autonomous mobile robots. The small footprint system is based on an embedded 8-bit Rabbit 2000 processor module [1] which controls, and literally sits above, a *stack* of up to 16 peripheral boards. These peripherals employ PIC microcontrollers to read sensors and command actuators. Because they all hang off a shared bus to the main processor, each peripheral is software addressable, allowing it to be uniquely referenced as a buffer which can be both written to and read from. Bus communication between the boards is dictated by a 9-bit RS-485 protocol. A full STACK description may be obtained in the STACK Manual [19].

The STACK was designed in conjunction with a programming language called CREAL [10] (for *CREAture Language*) which is written in EmacsLisp and supported on the Rabbit with its own architecture/application dependent operating system (see Chapter 3). The CREAL environment also specifies a graphical user interface to allow easy monitoring and debugging of STACK processes.

Together, the STACK and CREAL serve as a small, yet powerful computational system for long-running and responsive autonomous robots.

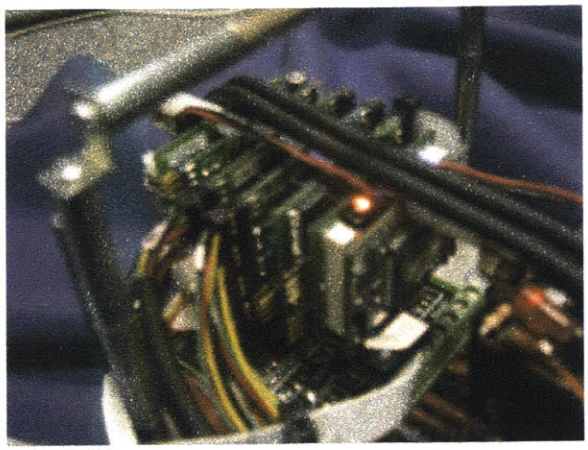


Figure 2-1: A photograph of the STACK that is on the robot. These boards control the robot autonomously once code has been downloaded to them.

Although a particular STACK can comprise any combination or types of peripheral boards, there are a few core boards that must be included in every stack: a top Rabbit Module followed by a *Carrier Board* and a *Power Board* [19] (see Figure 2-1).

- **Rabbit 2000 Module**

  STACK design began with a search for a compact, yet powerful commercial processing board that would be well-suited for a wide variety of real-time robotic applications. The final choice was the RabbitCore RCM2300 [1], a module that incorporates a high performance Rabbit 2000 microprocessor, 256K flash memory, 128K SRAM, 29 general-purpose I/O pins, and 4 serial ports on a 1.60 x 1.15 x 0.47 inch PCB. The module also has a programming header on it for direct download of assembled CREAL code.

  The onboard Rabbit processor runs CREAL, controlling the robot by managing communication with the rest of the STACK network and reporting on the health and status of the peripherals. However, in the implemented protocol, the main processor only communicates directly with the SPI SLAVE on the *Carrier Board*(see below) via 115.2Kbs serial interface. [5]

  Furthermore, the CREAL bios sends update data packets to the STACK at a rate of 64IIz.

- **Carrier Board**

  The *Carrier Board* is an integral part of every STACK, as it serves as the interface between the Rabbit and any other peripheral boards. It's two onboard 16F876 PICs, denoted the SPI SLAVE and the SPI MASTER, receive data packets from the host and reply with a continuous stream of data packets from the network [5]. Communication between this board and the peripheral devices occurs at 250Kbps.

  The *Carrier* is powered by the STACK bus.

- **Power Board**

  The system would not be complete without a *Power Board.* This board provides a jumper-selectable 5 V DC from either a regulator or an optoisolated DC-DC converter to the rest of the STACK through the stack bus. The *Power Board* is connected to a power supply of voltage between 9 and 18 Volts. In the case of autonomous mobile robots batteries would serve as this power supply. The board also has test points for debugging and a power/reset switch.

- **Peripheral Boards** Various types of software addressable peripheral boards have been built for use in the STACK, though the architecture could support numerous other feedback and actuation devices. Any new peripheral board need only conform to the STACK footprint, form-factor, and standardized RS485 bus, programming header, and auxiliary power connectors. Both firmware and hardware templates have been created for this purpose.

  The *Analog Sensor Board* interfaces up to 16 8-bit analog sensors. The *General-Purpose Sensor Board* has 8 analog ports as well as 8 ports that can be configured as either analog or digital.

  The *Servo Board* uses PWM (Pulse Width Modulation) to control up to 8 RC servos. The board is powered by the stack bus, though servo power is externally provided by an optoisolated supply. The *Motor Board* and *H-Bridge* are a pair of boards that control DC motors. The *Motor Board* sends PWM signals to the

H-Bridge that amplifies this signal so that a DC motor may be run off of it.

Information on the STACK, its protocols, its boards and its PIC programming details may all be found in the STACK Manual [19].

2.2 The Robot

A wheeled robot has been built that serves as the primary platform upon which all experiments described within this thesis were performed. This robot was constructed using hand-machined body parts, various sensors and actuators, and the STACK hardware described in Section 1. All experiments performed by the robot were programmed in the CREAL programming language.

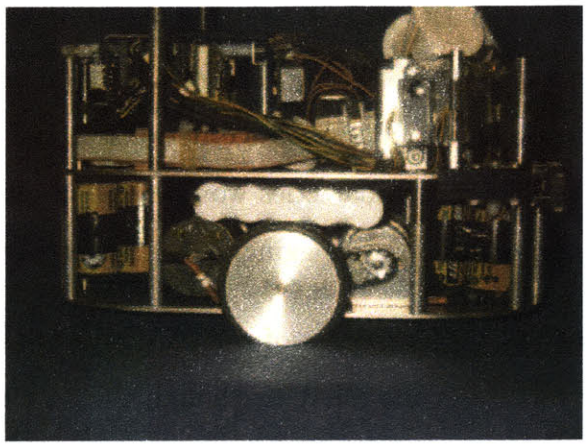


Figure 2-2: A side view of the autonomous mobile robot. Visible within this image are the sensors in the upper left, the white batteries, and the gearing that connects the wheel to the axis of the actuator.

In building and using this robot I am able to provide to future users a guide to what goes into designing, building, programming and debugging a robot. As a principal user of the CREAL and STACK system the robot was tested for functionality, robustness, practicality and ease-of-use of the hardware and software platform

The following sections will go into detail on the design of the robot. This will include descriptions of all commercial products that are used on board as well as a discussion about each of the STACK boards used and their purpose in relation to the functionality of the robot.

2.2.1 Morphology

The robot built for this thesis is a small dual-wheeled robot that is roughly 9 inches long, 6 inches high, and roughly 2 pounds in weight. The base of the robot is oval shaped machined aluminum with two bi-directional wheels placed in the center of each of the long sides. There are castor points at the front and rear for stability. This wheel and castor point arrangement allows the robot flexibility in the types of motion that may be achieved (such as having the ability to spin in place.)

The robot is made of two tiers: the lower tier containing batteries and motors, and the upper tier containing the control hardware and all of the sensors. There are separate battery packs for the motors (which require 7.2 volts) and the control hardware (which require 5 volts.) The robot contains touch sensors, infrared distance sensors and pyroelectric sensors, all of which are used by the on-board STACK (described in Section 2.1) to control the output movement of the robot.

A separate computer is used to write and compile CREAL code that is then downloaded onto the robot processors through removable cables. Once the robot is turned on it is completely autonomous and no longer relies on the separate computer for information, power, or commands. All sensing is done locally and is relative to the current position and actions of the robot.

Locomotion

There are two actuators on board the robot: one to control each of the wheels. The actuators are brushed DC motors, geared down to provide lower top speeds and higher torque. This provides the robot with more consistent movement and an available speed range that is more consistent with what is desired for this particular robot. The maximum speed of the robot with geared-down motors is roughly 30 feet per second.

Each of these actuators is combined with an encoder that can measure the rotational speed and position of the axis. The encoder data is used by the control STACK to both fine tune the speeds of the motors and to recognize situations when

the desired motor speed is far from the actual current motor speeds. An example of this type of situation would be when the robot is stuck trying to move through an stationary object unobserved by its sensors. In this situation the recognition of this difference in current and desired motor speeds will result in a control sequence that will change the desired action into one that will free the robot.

Power

For small mobile robots such as this, it is often decided that batteries are the simplest solution to solving power issues [23]. They are fairly cheap, often rechargable, and can be surprisingly light. In the case of this robot batteries were indeed chosen as the method of power supply. Another option for this robot would have been to tether the robot to a stationary power supply, but this defeats our purpose of having the robot be completely autonomous and free to roam over a wide area.

In choosing batteries there are also many options that must be considered, due to the fact that power is eventually exhausted when running an electronic device off of a battery. Larger batteries lead to longer running time, but are also much heavier and more difficult to place within a fixed-size robot frame. Smaller batteries are lighter and smaller, leaving more room within the robot frame, but lose their charge much more quickly. There are also many types of batteries to choose from. If disposable batteries are an option Alkaline batteries may be chosen, but in most cases rechargable versions such as Nickel Metal Hydride (NiMh) or Nickel Cadmium (NiCd) will be used. This is the most common type of battery used in small mobile robots and model cars, airplanes, or boats [23]. Another battery option is a Lithium Ion battery, which is generally much smaller, lighter and with a longer charge life, but is also much more expensive and much more difficult to recharge than NiMh or NiCd batteries. For larger robots Lead Acid batteries (like those found in cars) are often chosen, but are generally too large for *small* robots [6].

The robot used in this thesis contains two separate power sources: one for the motors and one for the processing STACK. The battery pack included for the motors provides 7.2 Volts of direct current, while the battery pack included for the hardware

STACK provides 12 Volts of direct current. It would have been more convenient to just use one battery pack, but the motors use a fixed 7.2 Volt input, while the STACK uses an input voltage between 9 and 18 Volts. In both of these cases the batteries are Nickel Cadmium.

Sensing

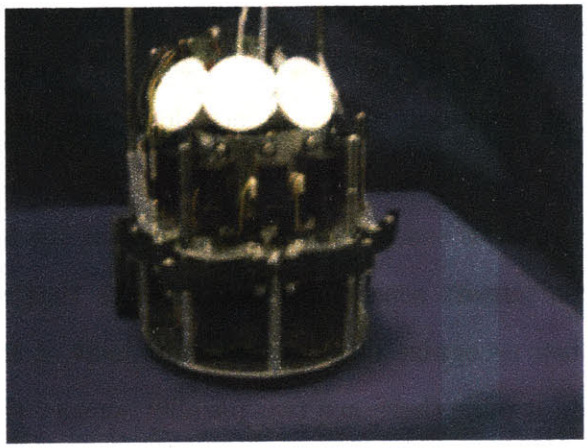


Figure 2-3: The front of the robot showing its three types of sensors. From top: pyroelectric (four large white discs), infrared (seven black rectangle boxes), touch sensors (four horizontal black rectangles hinged toward the center of the robot.)

There are three types of sensors on board the robot: bump sensors, infrared range finders, and pyroelectric heat sensors. These sensors, along with the encoders mounted on the axes of the wheels, are the only information that the control system receives about the environment of the robot.

There are eight bump sensors mounted on the robot: four in the front and four in the back. Each of the bump sensors is mounted in conjunction with a whisker-like bumper, to detect touch anywhere along a fixed length of the bumper. Each of the bumpers has one end mounted on an axis on the robot and has a spring-mounted bumper protruding out to a fixed distance from the edge of the robot base. These sensors are responsible for detecting when the robot has made contact with another object such as a wall or even the leg of a bystander.

On the front of the upper tier there are seven infrared (IR) sensors, mounted radially to cover just over 180 degrees in front of the robot. The IR sensors are Sharp

GP2D02 object detectors with a detection range between 10 and 80 cm. IR sensors are two part: an emitter that sends out a pulsed infrared signal, and a detector that reads these signals when they are returned. The signals sent out bounce off of an object and are returned in a time linearly related to the distance of the object in relation to the robot. These sensors will detect any hard surface, but have trouble with certain surfaces with low reflectivity. Due to the radial separation of the IR sensors it is possible that a narrow object such as a table leg could come near the robot without being detected. Fortunately most objects are wide enough that they will be aligned with at least one of the object detectors and be noticed before a collision occurs.

There are four pyroelectric sensors (pyros) mounted at the front of the robot near the IR sensors. These sensors sense motion of objects that are at a certain heat level. In particular they recognize objects that emit heat at the temperature of an average person. Inside a pyro there are two parallel sensors that are perpendicular to the projected line of motion. The act of a heat-emitting object passing in front of the pyroelectric sensors results in one of the two internal sensors detecting the object slightly before it is noticed by the other internal sensor. It is this difference in detection time that leads to the value returned by the sensor. This value does not correspond to the distance between the sensor and the detected object (as the IR sensors do), but instead correspond to the speed at which a detected object is moving past the sensor.

2.2.2 Control Hardware

The control hardware onboard the robot is a STACK comprised of seven boards: one of each STACK board described in the STACK description in Section 2.1, with the exception of the servo board. This section will briefly describe each of the boards present and their role in controling the robot. An off board computer is used to compile CREAL code that is then downloaded directly to the STACK. Once the STACK is activated the robot runs completely autonomously.

As described in the beginning of this chapter, the current STACK implementation

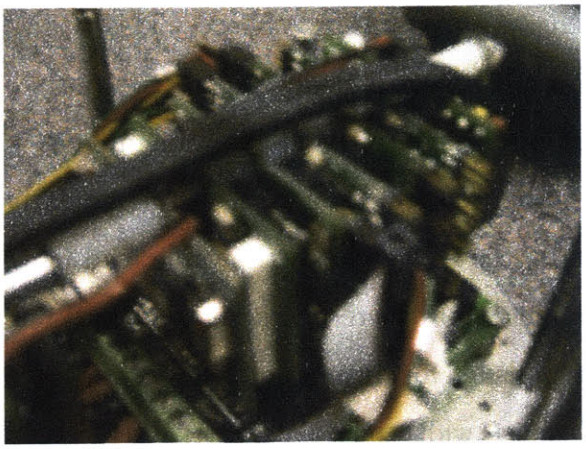


Figure 2-4: A view of the STACK on the robot. Farthest to the left is the Rabbit board connected to a carrier board, four peripheral boards and a power board.

uses a Rabbit 2000 processor for control. In order to have a functioning basic STACK there also need to be two other boards: a carrier board and a power board. The carrier board controls communications between the Rabbit and the peripheral boards on the STACK, ensuring that the code on the Rabbit is implemented correctly and accesses the correct peripheral boards. The power board is connected to the 12 Volt STACK batteries on board the robot and supplies a constant regulated 5 Volt power supply to the rest of the boards on the STACK bus.

The first peripheral board on the robot is an analog board. This board is responsible for detecting activation on any of the bump sensors mounted around the perimeter of the robot. When in natural open state, these sensors break the connecting circuit from analog board port to bump sensor and a voltage of zero is reported by the analog board. When the sensors are compressed, values nearing 255 (the maximum reading on an 8 bit analog board port) are seen. There are currently eight bump sensors on the robot, all of which are connected to the analog board.

There are also four pyroelectric sensors attached to the analog board. These sensors also return values between zero and 255. When the sensor detects no motion within its range field a value of 128 is returned to the analog board. Detected motion of an object to the left or right returns a value that is either slightly above or below 128, with the difference representing the magnitude of the motion reading and its movement direction.

Next there is a general-purpose sensor board. This board controls the eight IR sensors on board. Each IR sensor is connected to two ports of the general-purpose sensor board: one configured as output and one configured as input. The output port is responsible for triggering the IR sensor to send the regularly timed IR signal, and the input port returns to the sensor board the value read by the IR sensor.

To control the two actuators there is one motor board and one H-bridge board. The motor board is responsible for turning integer motor speeds specified by the main Rabbit processor into pulse width modulation (PWM) signals that control the motors. The motor board is also responsible for processing the values returned from the encoders on the motor shaft into integer numbers for the processor to use. The PWM signals are sent from the motor board to the H-bridge, where signals are amplified and passed on to the two actuators. The motor board is part of the STACK and uses communal STACK power and the STACK bus for communication with the rest of the STACK, while the H-bridge has an eight-line signal connection to the motor board and is powered by the attached 7.2 Volt motor batteries. The actuators are each connected to the H-bridge by one line through which the pulse width modulation (PWM) signal is sent to control the actuator speed.

Chapter 3

CREAL

CREAture Language, known as CREAL, is a programming language designed for use in conjunction with the STACK hardware described in Chapter 2. CREAL was recently written by Rodney Brooks [10]. It is designed to be used in many types of robotic applications, specifically autonomous mobile robots. The motivation behind the creation of this language is to have a platform in which fast processing can take place on small embedded processors, drawing little power and having the ability to run autonomously for long periods of time. CREAL supports large numbers of modules to be executed in parallel, making it ideal for the control of autonomous robots. The software package has capabilities to be used with many different types of processors and to support many types of peripheral components that may be present on a mobile robot, as will be described in Section 3.1.

CREAL is composed of three main components: a compiler that turns creature language code into assembly code, an assembler that turns assembly code into binary, and an operating system that may be loaded onto and used with a particular processor in order to perform the desired actions of the CREAture Language program. These three components run on a variety of different processors and on any operating system (Windows, Mac, Unix-based, etc.) that is able to run Emacs. These components are written in EmacsLisp and can be run from within the Emacs program.

3.1 The Compiler, the Assembler and the Operating System

The CREAL compiler has two distinct parts, a front end user interface and a back end that tailors the assembled CREAL code for a particular processor. The front end is completely independent of the processor. It is supported by a back end that is targeted to the architecture of a specific processor the CREAL code will run on. The job of the compiler is to turn CREAL programs into compiled assembly code that is suitable for the architecture of the back end processor.

The current implementation of CREAL is targeted to a Rabbit 2000 processor. Full details on retargetting the processor are available in the CREAL manual [10]

The operating system is architecture dependent, as it runs on the processor and is responsible eventually running the CREAL code. The specifics of the processor architecture directly influence the types and quantities of calculations that may be performed, and will therefore influence the options that the operating system are able to offer.

Details of how these components are configured for the Rabbit 2000, as well as details on how to reconfigure these components for an alternate processor, are thoroughly explained in the CREAL manual [10].

3.2 Installing the CREAL System

The act of installing the CREAL system on a desktop or laptop is straightforward for the computer scientist familiar with Emacs packages. This simply involves copying the necessary files to the desired machine, editing the *.emacs* file as described in the CREAL manual [10], and making sure the desired machine has a few required support systems, as will be described below. In order to install the CREAL distribution files, one must copy the distribution directories into some easily accessed space on the destination hard drive. The directory structure is all that is needed and does not require a more complex software installation.

Emacs must be installed in order to run CREAL. This ensures that EmacsLisp is properly functioning on the installation machine. The associated *.emacs* file must be altered to include the directory path where the CREAL directory structure has been installed, and calls that load the assembler and compiler. These updates are explained in detail in the CREAL manual.

Once these changes are made to the *.emacs* file the user will have the ability to assemble and compile files from within Emacs using either a command line approach or shortcut keys. When Emacs is opened an IELM (*interactive emacs lisp mode*) window will open. This provides a prompt in which commands can be given to compile a *.creal* file or to assemble a *.casm* (CREAL assembly) file.

```
ELISP> (compile-file "foo.creal")
ELISP> (assemble-file "foo.casm")
```

Another option for achieving the same result as above is to assemble or compile the files while editing them in Emacs. If a *.creal* file is open within Emacs, the shortcut command *c-x c-a* will automatically compile the CREAL code into assembly code. Similarly an open *.casm* file will automatically be assembled into binary.

3.3 Structural Design

3.3.1 Modules and Wires

A top priority of CREAL was that it support multiple threads of computation. Additionally, to support the architectural designs commonly used in autonomous mobile robots, CREAL has *modules* and *wires*.

Modules are computational entities that can contain many internal threads of computation. These local threads all exist within the shared environment of the module, sharing module variables, module input and output ports, and monostables (which hold a value of true or false for a fixed amount of time). The coordination of threads within a module allows a larger behavior to emerge, and to have the module itself capture some basic behavior.

In producing more complicated programs, many basic computational structures interact. Each module may contain input and output ports, and the interaction between modules takes place through wires that connect a source port to a destination port. Wires send *messages* which are essentially numbers of some predetermined type. The content of these messages and their relevant importance is completely determined by the code of the connected modules and how that message is to be interpreted. Wires have the ability to pass messages from one source port to multiple destination ports, to suppress another wire (replace the contents of the suppressed wire with the contents of the message sent by the suppressing wire), and to inhibit output from an output port of a module. When a source port is inhibited no signal passes from that port.

Modules may also contain ports that are local to the module. Wires are again used to connect local input and local output ports, but these wires and ports may not be suppressed or inhibited by non-local wires.

3.3.2 Threads and Events

A thread within a module is a completely independent process of computation. Many (perhaps thousands) of threads may run in parallel, with the distribution of processor time distributed over the threads by the processing operating system. Threads may perform such actions as waiting on an event, performing some act of arithmetic computation, assigning values to variables, sending messages through the output ports of the module, or triggering some sort of event within the module. As it turns out, a large portion of the thread's time is usually spent waiting for some particular event to take place so that it may perform its action.

An event is considered to be a situation in which a variable or port changes state. Examples of this may be a message arriving at an input port, a thread waking up from sleeping, or a monostable (which will be described shortly) changing state from true to false or vice versa.

When a CREAL program is initiated specified threads are spawned. These threads my dynamically end themselves, but no new threads may be spawned after the initial

creation of all threads.

3.3.3 Types

There are three types of variables available within CREAL: unsigned 8 bit variables and both signed and unsigned 16 bit variables. This variation leads to a conservation of necessary program size and allows easy interaction with various parts of the system. Peripheral sensors often return 8 bit values, and arithmetic computation within CREAL takes place on signed 16 bit values. If computation is attempted on a variable that is not a signed 16 bit type it will be converted to that type in order to continue with the computation. It is also possible to typecast a variable from one type to another if necessary.

Similarly, input, output and local ports are also designated as one of the above types. This type specification identifies which type of variable value is intended to be sent as a message across the connected wires.

A monostable, as briefly discussed above, is a data type that can either have the value of true or false. This alone makes it similar to a boolean value seen in other programming languages. The difference is that a monostable must be initialized as either true or false, and its state may be changed by an event that triggers it. CREAL does not contain representations for first-hand true and false objects, so a monostable may not be directly set as true or false after initialization, only through triggering an influencing event.

The other component to monostables is that they also act as independent timers. A monostable may be set to change its state upon the occurrence of some triggering event, but after a predetermined time it will revert to its default state. If a monostable is triggered and then retriggered before reverting to its default state, it will stay in its changed state until the predetermined amount of time has passed since the last triggering.

3.3.4 Interaction with Peripherals

Within CREAL a port on a peripheral device may be addressed and used in the same way as a variable with a predetermined type. In the definition of the peripheral, a unique peripheral board ID was given, along with a unique board name and a naming of all of the ports to be accessed. A simple conjunction of the peripheral name and the port name that correspond to the desired device will allow the value associated with that device to either be set or read from within CREAL.

In reality the sharing of information between the processor and the peripheral devices takes place in buffers that reside on the carrier board. These buffers are accessible by both the processor and the peripherals, although the two access the contents of these buffers asynchronously. When a new sensor reading becomes available the value is sent to the carrier board which places that value within the appropriate buffer. When the processor needs to find out the current value of that device it simply looks in that buffer. It is quite possible that the buffer had been updated many times between readings by the processor. Similarly the processor may write control signals to buffers that correspond to output devices such as actuators. The buffer value is frequently checked by the peripheral device but it is possible that the value was updated multiple times by the processor between readings.

There are two types of buffers that may be defined: fast buffers and slow buffers. Fast buffers indicate that the values contained within should be checked and updated as often as possible, at a rate close to 100 MHz. Slow buffers indicate that the values are not critical. These buffers are checked and updated at roughly 25 MHz.

3.3.5 Low-Level Processor Configuration

In order to run CREAL programs a processor must contain a BIOS that can process CREAL programs appropriately. This BIOS contains a clock, is responsible for communication with the carrier board (which communicates with the rest of the peripheral boards), and performs basic operations.

On top of the BIOS an optional ROBOS, or robot operating system, may be

installed. This is only necessary if CREAL controls hardware devices other than the boards included in the STACK. The last item needed on the processor is the assembled CREAL program.

The BIOS must be assembled and installed independent of the other two sections, and must be installed on the processor before either of the other two sections are installed. The ROBOS may be assembled and installed only after the BIOS has been appropriately loaded. After these two items are in place the CREAL program may be compiled, assembled, and installed on the processor.

3.3.6 Debugging

There are two ways to debug a CREAL program. Both rely on connecting the serial port to a host machine. The first strategy involves sending text messages. Text may be sent as often as desired and may include pieces of information such as sensor values or a warning of some type when a particular state is reached within one of the modules. Obviously, this interaction is only in one direction: from the CREAL processor to the host machine. This format is very useful when data must be collected from the STACK.

The second strategy is a graphical user interface that allows the user to watch the values of variables and peripherals through a "thermometer" which shows the current value. For instance, a bump sensor returns a value that is either near 0 or near 255. When the value is near 0 the thermometer is almost all red. When the bump sensor is compressed the thermometer shows the value jumping high by the slider being mostly blue, with only a slight bit of red on the far end (if the compressed value is not exactly at 255.)

Thermometers give the user increased runtime flexibility by allowing some values to be altered by the user as the CREAL program runs on the processor. User-manipulated thermometers are used on variables and some output values such as the control signals that are sent to actuators. In this case the user may simply use the mouse to maneuver the slider to represent the intended numerical value (which is conveniently shown off to the side in either decimal or hexidecimal). This option is

often used when many types of settings must be tested at one sitting and intense data collection from one particular setting is not needed.

Both of these debugging methods may be used with a Bluetooth wireless serial connection which allows untethered communication between the host machine and the STACK.

Chapter 4

A Neural Net Expansion to CREAL

A neural net is modelled after the complex network of neurons in the human brain. These biologically inspired computational networks are used to perform a similar style of distributed computation within computer programs. Large systems of simulated neurons exist in which each node of the network is responsible for performing some small bit of computation and passing the resulting value to its neighboring network nodes. Sensory input flows into nodes of the network and weighted signals are passed on to neighboring nodes. This signal passing continues from input nodes, through the neural net and finally to the output nodes.

As described in Chapter 1, neural nets are an extremely popular control architecture for use in autonomous mobile robots [36] [40] [2] [35]. It is the strong presence and popularity of this trend, combined with the complexity of writing CREAL code that uses neural net structures, that motivates the creation of a set of tools that allow users to easily create neural nets within CREAL.

This chapter will lead through a discussion of the structure and control present in neural nets as well as the important design features of this control method that would be desired by a user using a neural net tool. As CREAL does not provide any abstract framework specific to the implementation of neural network architectures, a set of abstraction tools that facilitate the creation of neural nets within CREAL will

be discussed.

4.1 Neural Net Control Description

4.1.1 Structural Organization

One of the simplest ways to view a neural network is as a set of similarly structured nodes (also referred to as *modules*) that interact through simple connections. Each structure performs some small amount of computation upon input information and proceeds to pass modified information on to neighboring modules. Certain modules may have slight differences in the computation that is performed within them, but all will have the same basic structure containing input from some neighboring nodes, a central computation that is performed on input data, and output to some neighboring nodes.

Depending on the design of the neural net, the connections among modules may be arranged in a variety of fashions. The simplest type of neural net is a *perceptron.* When a perceptron neural network is used to control a mobile robot each of these nodes has direct input from all of the available sensors and each node outputs a value that is not used as input by any other nodes. This output value is usually used as direct input into an actuator. Information flowing through the sensors is immediately processed by the layer of nodes and actuator values are set.

A more complicated neural net structure would contain two layers of nodes rather than the one layer present in a perceptron. The lower layer would be identical in structure to a perceptron layer, each containing multiple input ports and one output port that is directly linked to an actuator. The difference lies in that the inputs come from another layer of nodes as opposed to directly from sensors. The upper layer nodes take inputs from every sensor and produce outputs that go to every node in the lower layer.

This double layer design is commonly referred to as a neural net with *hidden nodes* (see Figure4-2.) This term is appropriate for this type of design because as the robot

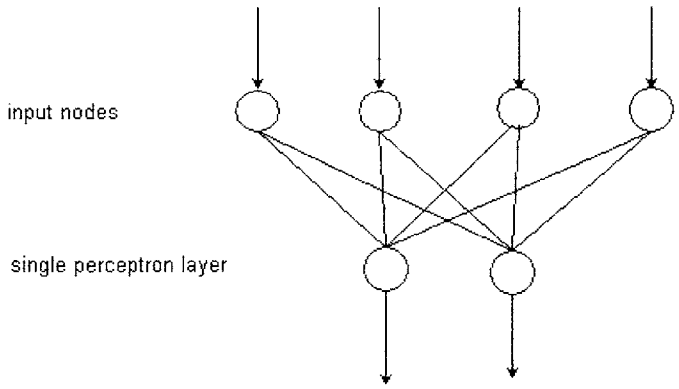


Figure 4-1: A single layer perceptron. Input values flow in from the top through the input nodes, on through the perceptron layer then out through the bottom.

using the neural network runs in the environment, observers may only observe the inputs to the neural net (in the form of sensor data) and the output values (in the form of actuator data). An observer has no direct knowledge of the values internal to the robot that are used to produce the witnessed behavior. Therefore, this top layer that has hidden output data, is considered a hidden layer. Depending on the complexity desired by the designer, a multiple number of hidden nodes may exist.

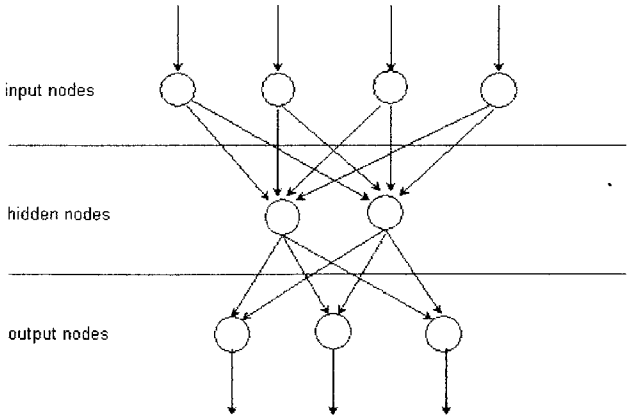


Figure 4-2: Multiple layers of a neural net, including an input layer, a hidden layer and a computation layer.

4.1.2 Neural Computation

Along with the basic structure of nodes and connections, the connections are assigned a *weight* that influences how the signal passed along it is used in computation at the arrival node. When a node performs its neural computation, it uses a summation of *weighted signals* from all input nodes. A weighted signal is the product of the signal value passed on the wire and the fixed weight of that wire. This summation is fed into the *activation function* of the node, producing the output value for that node. It is this set of connection weights which makes a neural net unique and which will influence how input data is converted into output data.

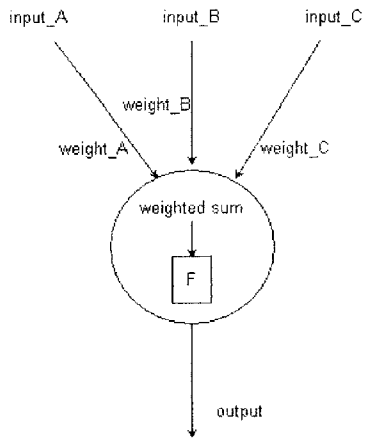


Figure 4-3: Input and computation of a node within a neural network. Weighted input signals are combined into a weighted sum that acts as input into the internal computation function F.

The three basic activation functions seen in neural nets are the *linear function*, the *threshold function* and the *sigmoid function* (see Figure 4-4.) All three functions take the weighted sum of node inputs. The linear function returns the weighted sum value and the threshold function returns 1 if the weighted sum is a positive value and 0 otherwise. The sigmoid function returns a nonlinear value that is bounded between 0 and 1, with a value of 0.5 returned with an input sum of 0. See Figure 4-4.

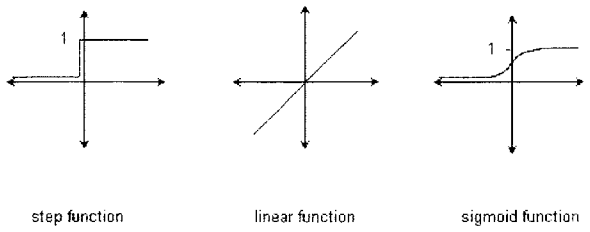


Figure 4-4: Internal node computation functions.

4.2 CREAL and Neural Nets

There are many aspects of CREAL which make it an extremely useful and appropriate language for expressing a neural net architecture. The core design of a neural net includes many nodes running in parallel, each performing some localized bit of computation at the same time as neighboring nodes are computing other values. CREAL, as described in earlier chapters, automatically schedules the timing of multiple threads of computation running in parallel while some other languages do not. With this framework the user does not need to separately plan how to conduct the processing timing of each of these threads, just specify what the threads of computation are. In CREAL, each of these simultaneously computing nodes may be represented by a CREAL module. Each module has a set of internal threads that will perform the node computation function, while CREAL wires will be used to pass information from module outports to other module inports. Using CREAL's macro facility, it is straightforward to design a neural net. This avoids the tedium of doing so manually which is repetitive.

A toolkit has been built that allows users access to neural net computation in CREAL through a set of macros that create this complicated yet structurally repetitive control code. With these new tools users need only specify the structure and computation style desired within the neural net and the lengthy CREAL code is created automatically.

Before explaining the details of the toolkit we should look into the variations that

are often seen within neural nets. In order for this toolkit to be truly useful it is necessary that users have the flexibility to create control systems that accurately represent their ideal control design.

4.3 Necessary User-Specified Components

Seeing that there are so many variations in the use of neural nets, it is important that key elements are identified and included in the neural net abstraction interface available to users. Each user must have the ability to create the control architecture that is specific to their task. The key variations in neural nets are described below.

- Structure / Topology

  As described at the beginning of this chapter, there are many different structural forms that neural nets may take. Perceptrons, for example, have nodes all contained within the same layer. That is to say, each node has access to the network inputs and each node has exactly one output that corresponds to a unique network output. Even though perceptrons all have only one layer, there may be a variable number of nodes within that layer, depending on the task which is to be accomplished and the design desired by the user.

  Neural networks may be made up of multiple layers of hidden nodes. As with single layer neural networks, each layer may have any number of nodes within it, depending on the complexity and style of control that is desired. Information on determining the ideal number of layers and / or nodes within a layer can be found in many neural net text books [17]. The structure of a network will greatly depend on the specific task that is desired.

  There may also be variation in the inputs that are fed to particular nodes. For instance, there may be a set of nodes which receive input from sensors of type A, while another set of nodes receive input from sensors of type B. These two disjoint sets of nodes may act as a single layer and in turn be the input for a separate layer of nodes.

- Input Weights

  It is critical that the user be able to specify the weights that correspond to the paths connecting nodes. One network with a particular weight distribution will most likely produce vastly different output than a network with identical node structure and different weight values. When neural networks are used as the control architecture within autonomous robots it becomes easy to see that slight differences in output values can lead to drastically different behaviors.

- Function in Node

  As mentioned in Section 4.1.2, there are multiple types of computation that may take place within nodes of a neural network. The most common types of computation that is performed within nodes are the threshold step function, the linear function, and the sigmoid function. It is, of course possible that a user may want to use some other type of function, but this is fairly uncommon. For most practical purposes these three functions will be the primary choices for node computation.

  Users may also choose to create a neural network in which some nodes use one type of computation function while others use another. This is also not very common, but nonetheless is an aspect of user functionality that should be addressed.

4.4 Macro-Interfaced Abstraction Layer

There are two macros that have been written so that CREAL users may simply and easily create a neural net to their desired specification.

4.4.1 Defining Input Nodes

`macro: (define-input nodeName buffer)`

The first component is responsible for creating an input node that references a particular input (such as a sensor) and making that value available to subsequent

computation nodes within the network. As described in Section 3.3.4, peripheral ports (such as those that may contain connected sensor values) are referenced within CREAL as a buffer position within the STACK carrier board. The input node macro described here takes as an argument a buffer location such as a peripheral port that it will constantly reference.

The most common input signals would be retrieved from ports on either an analog or digital STACK board, representing values from a connected sensor, but values may also be retrieved from globally accessible user-created buffer slots within CREAL. The advantage of a user-created buffer locations as node inputs is that this gives users the option of pre-processing raw sensor values into some other format that is more suitable as input into the neural net. An example of this would be setting the post-processing (and node accessed) buffer slot with a value that represents if some threshold value had been reached by the sensor data, thereby converting integer sensor values into a binary input value.

When these modules are created, threads internal to the input node module will continuously attempt to retrieve values from these input locations. The module is also created with an output port that can be accessed by other modules through a connecting wire. The internal threads will will continuously attempt to send the input value out through this designated output port.

Note that this alone will not result in any values being sent to other modules. In order for a separate module to receive this value a wire must be created that connect this continuously updated output port to an input port of the separate module.

The syntax of creating an input node for a neural network is simply the macro call, the name you wish to give the input node, and the input location. An example of a macro call could be

```
(define-input INPUT-1 (bump-board port1))
```

where (bumpboard port1) is a peripheral board port or hand-created buffer slot that has been previously created in CREAL. This macro call will result in the following expanded CREAL code:

```
(defmodule INPUT-1
   :outports (out)
   :threads ((whenever :true)
             (send out (* 0x0100 (bump-board port1)))))
```

Notice that the multiplication performed on the value stored upon the bumpboard value is essentially converting the 8-bit integer value to the format of a fixed point 2-byte number through what amounts to an 8-bit arithmetic shift to the left.

4.4.2 Computation Nodes

`macro: (define-node nodeName nodeFct (&rest (parentNode weight)))`

The second component in the CREAL neural network toolkit is a macro that is responsible for creating the non-input nodes of the network. It is these nodes that take the weighted sums of all input values and perform an internal computation function upon it. The computation node macro described here takes as arguments the names of the nodes that it wishes to receive weighted inputs from, the weights of each of those inputs, and the computation function that is to be used within the node.

As with the input nodes, the computation nodes created by this macro will contain one single output port to which calculated values are sent. It is with this structure that other nodes within the neural network are able to use this the output of this macro-created node as input to their node. An output port from a node is represented within CREAL as a buffer location that other modules may query (through the use of a wire), just like a peripheral port or user-defined buffer port.

Each computation node must contain one input port for each of the nodes that passes a value into it. Just as an output port, input ports are represented in CREAL as a buffer position that other nodes may write to through the use of a wire. If there were just one input port then a number of output values from neighboring nodes would all try to write their value in that same memory location.

Similarly, each computation node must contain weight values for each of its input signals. These weights will be used when calculating the weighted input sum that is

used by the computation function. This computation function must also be specified in the macro call so that the correct CREAL code can be generated to perform the desired calculation.

The syntax of creating a calculation node for a neural network contains the macro call, the name to be given to the computation node, and a list of pairings of input nodes and signal weights. An example macro call of

```
(define-node NODE-A linear ((INPUT-1 0.5)
                            (INPUT-2 0.8)
                            (INPUT-3 0.5)
                            (INPUT-4 0.15)))
```

will result in the following expanded CREAL code:

```
(defmodule NODE-A
   :inports ((inp-INPUT-1 0)
             (inp-INPUT-2 0)
             (inp-INPUT-3 0)
             (inp-INPUT-4 0))
   :outports (out)
   :localvars ((sum 0)
               (send-flag 0)
               (w-INPUT-1 0.5)
               (w-INPUT-2 0.8)
               (w-INPUT-3 0.5)
               (w-INPUT-4 0.15))
   :threads
      ((whenever (and (receivedp inp-bump1)
                      (receivedp inp-bump2)
                      (receivedp inp-bump3)
                      (receivedp inp-bump4))
          (setf sum 0)
```

```
            (incf sum (*fp inp-INPUT-1 w-INPUT-1))
            (incf sum (*fp inp-INPUT-2 w-INPUT-2))
            (incf sum (*fp inp-INPUT-3 w-INPUT-3))
            (incf sum (*fp inp-INPUT-4 w-INPUT-4))
            (setf send-flag 1))
        (whenever (= send-flag 1)
            (setf send-flag 0)
            (send out sum)))
(defwire INPUT-1-to-NODE-A
   :from (INPUT-1 out)
   :to (NODE-A inp-INPUT-1))
(defwire INPUT-2-to-NODE-A
   :from (INPUT-2 out)
   :to (NODE-A inp-INPUT-2))
(defwire INPUT-3-to-NODE-A
   :from (INPUT-3 out)
   :to (NODE-A inp-INPUT-3))
(defwire INPUT-4-to-NODE-A
   :from (INPUT-4 out)
   :to (NODE-A inp-INPUT-4))
```

4.5 Evaluating CREAL and its Use for Neural Net Definition

There are many reasons why CREAL is an appropriate programming language with which to implement a neural net architecture. As discussed earlier, there is usually some difficulty in programming the timing of a distributed system with many concurrently running threads of computation. These timing difficulties are completely removed from the list of difficulties the user faces, as they are implemented automat-

ically within CREAL. Similarly, the structural design of CREAL makes the implementation of many simultaneous computation modules with interconnecting paths quite straightforward.

The tools that have been built allow for flexibility in creating a neural net that suits the needs of the user. The user may control the structure of the net, the style of computation that takes place within the nodes, and the weights of each wire. The tools allow for straightforward and easy access to a neural net representation.

Unfortunately, there are also some shortcomings to the neural net capabilities of CREAL. Some of these shortcomings result from the design of CREAL while some are aspects that may possibly be implemented in the future, but which have not been done so already.

The first major shortcoming is that CREAL cannot express the sigmoid function. The sigmoid function is one of the most commonly used functions in neural nets. The base of this problem comes from the fact that the Rabbit hardware that is currently on the STACK does not support division. The fixed point data type allows for multiplication and the use of decimal points, but not division explicitly. One approach that has been suggested is to create a lookup table of the output values of the sigmoid function with input values surrounding zero (the most common input values). Rather than being able to perform the actual computation of the sigmoid function for a given input, the nearest value held in the lookup table may be returned for a close to accurate output. In the examples used in this thesis the sigmoid function was not used.

4.6 The Macros

```
(defun convertfp (x)
  (logand 65535 (round (* 256.0 x))))
```

```
(defmacro define-clock (clock-name sleep-time)
  (let ((x ()))
    (push `(defmodule ,clock-name
     :outports (out)
     :threads ((whenever (sleep ,sleep-time)
 (send out 1))))
  x)
    x))

(defmacro define-input (node-name input-sensor clock-name)
  (let ((bob ()))
    (push `(defwire ,(intern (format "%s-to-%s"
     clock-name node-name))
     :from (,clock-name out)
     :to (,node-name inp-clock)) bob)
    (push `(defmodule ,node-name
     :inports (inp-clock)
     :outports (out)
     :threads ((whenever (receivedp inp-clock)
 (send out (fp (+ 0 ,input-sensor))))))
  bob)
    bob))

(defmacro define-node (node-name parent-list)
  (let ((inport-list ())
(localvar-list ())
(recp-list ())
(weighted-input ())
(final ()))
    (do ((p-list parent-list (rest p-list)))
```

```
((null p-list))
      (setq p `(,(car p-list)))
      (push `(,(intern (format "inp-%s" (caar p))) 0)
    inport-list)
      (push `(,(intern (format "w-%s" (caar p)))
      ,(convertfp (cadar p))) localvar-list)
      (push `(receivedp ,(intern (format "inp-%s" (caar p))))
    recp-list)
      (push `(*fp ,(intern (format "inp-%s" (caar p)))
,(intern (format "w-%s" (caar p))))
    weighted-input)
      (push `(defwire ,(intern (format "%s-to-%s"
       (caar p) node-name))
       :from (,(caar p) out)
       :to (,node-name ,(intern (format "inp-%s"
(caar p))))) final))
    (push `(defmodule ,node-name
     :inports ,inport-list
     :outports (out)
     :localvars  ,(append '((sum 0))
  '((send-flag 0))
  localvar-list)
     :threads ((whenever ,(append '(and) recp-list)
 (setf sum 0)
 ,@(mapcar #'(lambda (parent)
       `(incf sum ,parent))
   weighted-input)
 (setf send-flag 1))
       (whenever (= send-flag 1)
 (setf send-flag 0)
```

```
(send out sum))))
 final)
   final))
```

Chapter 5

Implementation of a Neural Net Architecture

This chapter is aimed at showing the applicability of the newly built neural network tools in CREAL through the implementation of an autonomous mobile robot controlled by a neural network architecture. The use of the neural net tools described in Chapter 4 will be detailed and explained, as well as a personal account of the successes and difficulties faced in this experience.

Many neural net architectures implement complicated tasks that often require an extremely advanced robot morphology to be able to perform the task, or a complex method to accurately learn the required neural net weights. In order to show the functionality of the newly constructed neural net tools a complicated performance example is not necessary. The emphasis during this experiment is on the abstraction layer giving the user a simple interface to complicated neural net capabilities rather than the task itself. It is assumed by the author that if a user were to use these CREAL tools to create a neural net driven robot that performs complicated tasks, offline learning of some sort would take place outside of CREAL to learn the network weights.

5.1 The Task

With the exception of autonomous mobile robots that perform in barren landscapes or environments unsuitable for humans, most mobile robots live in the world built and used by people. Interaction with humans, be it directly or indirectly through human-occupied environments, is a requirement that most robots need to address.

Walls, for instance, are a result of human construction and are found in all places that host people. Seeing that human-occupied environments are often the same environments that are occupied by robots it is no surprise that walls are a common occurrence in robot implementations.

Wall following is a task that many mobile robots need to perform for a variety of reasons. In almost any navigation or exploration experiment wall following is a tool that will help the robot move through its environment in order to achieve its ultimate navigation or exploration task, and it is a necessary trait for mobile robot whose task is to smoothly interact with humans.

This task alone is not novel or original in any way, but its prevalence in pre-existing systems shows its validity as a task. Although to a viewer it may seem like a simple action, it is a sufficient action to show the functionality of a neural net architecture. It is this task that will be used to demonstrate the functionality and accessibility of a CREAL neural network control architecture constructed using the tools created and described in Chapter 4.

5.2 The Neural Net Control Architecture

The task of wall following depends only on the values of the IR sensors that are angled towards the wall. One form of wall following is achieved by fine tuning parameters so that the robot is constantly switching between veering towards the wall and away from the wall. If the IR sensors read values that are too high the robot should turn away from the wall. Likewise, if the IR values are too low the robot should move slightly towards the wall. Straddling this fine line can result in a behavior that bystanders

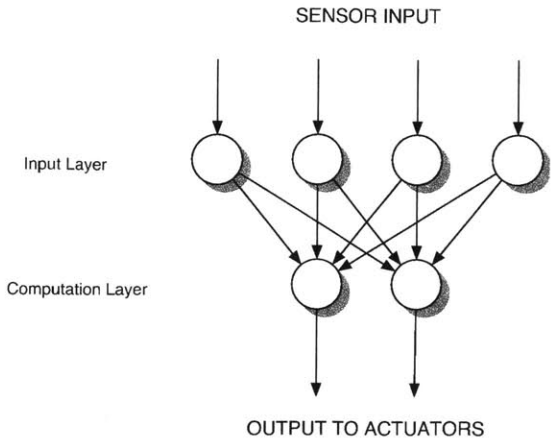


Figure 5-1: Original planned neural net structure for the implementation of wall following, weights not yet calculated.

would observe as wall following.

In order to perform the wall following task, one must know the relationship between the sensor data being collected by the robot and the speed of the two wheels. The intensity of the sensor readings (indicating distance to nearby objects) is directly related to the difference in motor speeds. It is this difference that controls the strength at which the robot turns away from objects it perceives to be in its path or somewhere dangerously close.

With the goal of having the robot follow a wall on one particular side of its body rather than on both sides it is possible to limit the number of sensors that will act as input to the neural net control. With seven IR sensors on the robot only four would be angled toward or parallel with the wall that is to be followed. The pyroelectric sensors are not applicable to this task due to the fact that they only detect objects with a human-level temperature and are blind to objects such as walls. Likewise, the bump sensors are not applicable because they indicate when a mistake has already occurred. The addition of a response sequence (such as detection of an impact, back up, turn away) would add many new complexity issues. The original goal of demonstrating the functionality of the CREAL neural network tools would be met in achieving the simpler task of wall following alone.

The neural net would only require two output values: the desired motor speed for

each of the two motors. Note that the desired motor speed is different from the motor speeds observed by the encoders on each of the motor shafts. The desired motor speeds are values that the control system sends to the motor board, which in turn are sent to the motors. Quite often, the observed values of the motors are not in fact equal to the desired values. This is due to differences in acceleration, friction, and physical aspects of the motors being in contact with the floor, as well as internal control parameters such as gain that are present in the pulse width modulation motor control algorithm. Strictly collecting observed motor speeds would not give the information that would be necessary for a neural net structure to accurately mimic the sample action. The required data are the *desired* motor speeds, which in turn produce the *observed* motor speeds and the resulting wall following behavior sequence.

With this information in mind it we decided to build a neural net with four input nodes (to correlate with the four infrared sensors on the right side of the robot body) and a single perceptron layer containing two nodes (that would correlate with the two motors.) This decision alone would not produce a wall-following robot; the values of the weights within this neural net would somehow need to be calculated. The next section will discuss the various strategies tried in order to determine appropriate weight values.

5.3 The Approach

5.3.1 Method 1: Data Collection and Offline Learning

The first method approached was two part: The first step involved collecting sensor and motor data printed as the robot was successfully performing wall following. This data would then be used in offline learning on a separate machine to calculate the neural net weights that produced output most successfully matching the test data.

There were two methods tested in order to collect sensor and actuator data: manually pushing the robot and remote control of the robot. There were major problems with both methods.

Manual Motion

The first attempted data collection scheme allowed the robot to be manually pushed with the desired wall following motion (as determined by the observer responsible for pushing the robot), while IR sensor and observed motor encoder data were output to a file. This approach worked surprisingly well in that the encoders accurately output the observed rotational speeds of the motors, as was hoped.

An extensive data collecting session took place, only to soon realize that the motivating logic for this method was incorrect. The true difference between motor speeds desired by the STACK control program and actual motor speeds witnessed and output by the actuator encoders was realized and the flaw in this method was exposed. To explain, a neural network controlling a robot would read sensor values and at each step calculate the desired motor speeds. These motor speeds would be attempted to be reached by the motors but in reality the actual motor speeds observed by the encoders would usually be off by a sizable amount.

An attempt was made to try to calculate a set of rules that would output the witnessed motor speeds if given the desired motor speeds. This strategy was quickly abandoned after discovering that accurately mapping the desired motor speeds to the observed motor speeds would be an extremely difficult task due to the presence of multiple control parameters and environmental influences.

Remote Motion

To collect the required data a new system was put in place. This approach included adding stack hardware and control software such that the robot could be controlled remotely by a joystick of the type often used to maneuver remote controlled airplanes. By controlling the robot through a joystick as opposed to manually, the desired motor speeds were able to be collected as the input sensor data was read. It is this desired motor speed that we would like the final neural net to output when given the witnessed input data.

With this joystick-driven method, multiple runs worth of data were collected. The

basic action that was performed by the operator in each of the runs can be described as follows: When no obstacles are within range veer slightly to the right; when an obstacle is seen veer to the left enough to avoid collision with the object; if parallel to the wall continue at a fixed distance.

At this point data was collected from multiple wall following runs in which sensor and desired motor values were output to a file at a rate of 20 Hz. It is this data that was used in the offline learning of the neural net weights.

Offline Learning of Neural Net Weights

Matlab has various toolkits associated with it, including the Neural Network Toolkit. This package is aimed specifically at performing the computation related to different types of neural nets such as construction, simulation and learning. Matlab allows many different types of learning to take place, allowing the user to decide which weight-learning algorithm to use, the number and types of runs to be performed, and many other variables in the calculation algorithm. This program was planned to compute the necessary weights to be learned offline.

Numerous attempts were given at using this program to find weight values for the collected data, but every attempt failed miserably. When automatic learning of connection weights was tried the error rates immediately became exceedingly large.

During many early attempts it was believed that there was a problem with how this tool was being used, but a look into the collected data showed that a fixed set of input sensor value could produce a wide variety of resulting output values. When the full extent of the one-to-many mapping from input to output values was understood, it was decided that the weights to the neural network should be found in another way.

5.3.2 Method 2: Manual Adjustment of Weights

The failure of the offline learning approach described above led to some brainstorming about other ways in which an acceptable neural net could be created in order to

achieve the behavior of wall following. As an observer it was easy to tell when the robot was behaving properly and when it was behaving poorly, but trying to find the calculations that would achieve this behavior without having direct and immediate observer feedback would be quite difficult. An approach was needed that would allow an observer to manually adjust the computation parameters controlling the robot behavior, and be able to immediately see the changes made.

The approach that was decided upon allowed the robot to be remotely operated through a wireless Bluetooth serial connection. This operation would not be as straightforward as the joystick control described earlier, for the person controlling the robot would have direct control over only the parameter values within the neural net currently running on the robot. There are many weights in a neural net of this type, and only through trial and error is the user able to come up with a collection of weight values that will result in a successful final behavior. With a fairly restricted range of weight values that would produce the desired behavior, the trials would undoubtedly result mostly in errors and erratic, unpredictable behavior. This is true, but it is in fact a way that would both allow the observer to confirm that successful behavior is being achieved and find the neural net weights that would be required to produce such a behavior.

Watching and recording values is most easily done through a simple text output from the CREAL program. This method allows for output from the program to be sent to a text screen, but with this alone the user may not input or change values internal to the CREAL program. The graphical user interface to CREAL, on the other hand, allows the user to directly adjust variable values as well as watch a graphical representation of the values held. As described earlier, thermometers, or graphical sliders, are used to perform this visualization, showing the value held by the parameter byte. It is through this tool (and the Bluetooth connection) that the user would be able to adjust and control the weights of the neural net currently running on the robot.

5.4 Improving Performance

As described earlier, there are many possible combinations of weight values that may be given to control a robot. Unfortunately, most of these combinations will control the robot to perform nothing like the final behavior desired. It is, in fact, a very narrow window of weight values that will lead to the achievement of the desired behavior. Knowing that this window is very small implies that trial and error weight testing will pass much more painlessly if the initial weight values are chosen wisely so that a correct set of weights may be found quickly.

Typically it would be very difficult to guess the weights to a neural net so that the output values roughly correlate to the desired output values, but luckily wall following is a rather simple case. The perceptron model used here has only four inputs (the IR sensors along the right side of the robot) and only two outputs (the left and the right motor output values.) If there were a hidden layer to the neural net, multiple types of sensors or more output nodes the rough guess of weight values would be much more difficult to find.

Looking through the wall following data taken with the joystick it was quite easy to see the IR values rise (when the robot got close to the wall, presumably) and the resulting motor speeds as the robot moved away from the object. Using this visual data, it was possible to come up with a rough correlation of how the values of the IR sensors mapped to the output motor values. The visually chosen data, of course, fit the general trend in many easily observable situations in the data but came nowhere close to producing the recorded output in all of the situations.

The assumption that the robot would always be following a wall on its right side allowed for great simplification of the problem. The only capabilities that the robot would need to have in order to perform this task would be to slightly veer toward the wall when no obstacles were seen and to be able to turn from the wall with varying intensity when an obstacle was observed.

These behaviors could be achieved by keeping the left wheel speed fixed and allowing the right wheel speed to vary depending on the value of the sensory input.

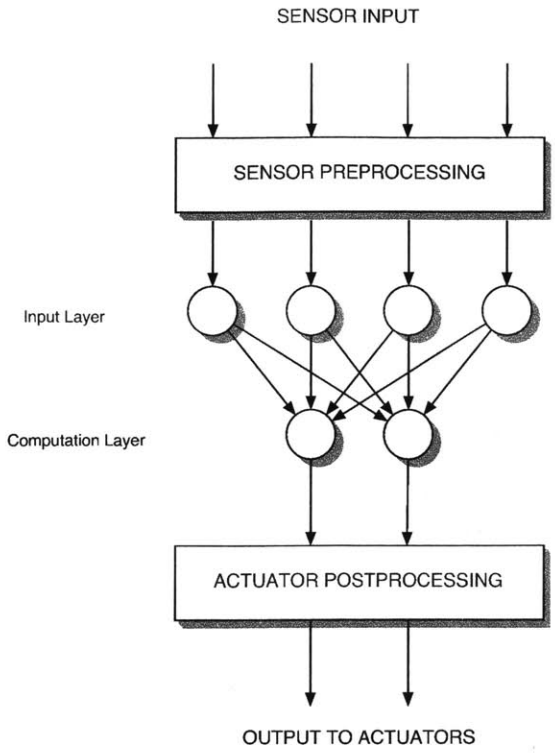


Figure 5-2: Updated structure of the neural network nodes that will implement wall following.

Similarly, the lower speed of the right wheel could be bounded because only a slight veer to the right would ever be needed. This simplification could be put into practice by using the output of the neural net as delta values that would be added to base wheel speeds rather than as exact values that the motors would be set to, as originally attempted.

It was also decided that preprocessing of the input values from the IR sensors would help to make finding the weights of the neural nets easier. Threshold values were set that correlated roughly to distances from objects detected by the IR sensors. Three threshold values were specified producing LOW, MEDIUM and HIGH threshold classifications. Values of 0, 1, 2 or 3, depending on which threshold values were met, would then be used as the preprocessed input into the neural net.

Figure 5-2 shows the modified neural net structure with preprocessing and post-processing modules.

5.5 Behavioral Results

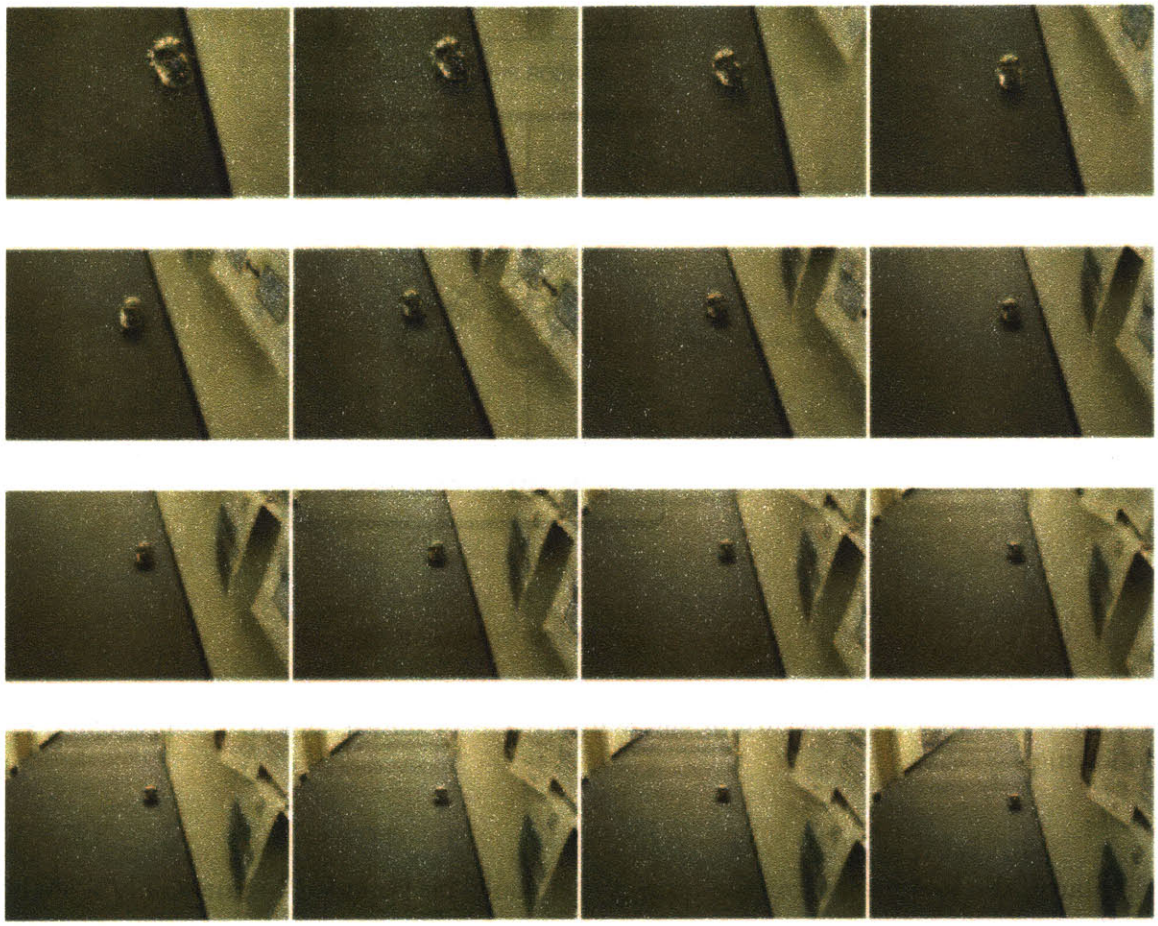


Figure 5-3: Still frames from a video demonstrating the neural net controlled wall following behavior. These images show an overall smooth behavior of wall following as the robot travels down the hallway. Upon closer inspection it can be seen that the robot is actually swerving slightly from side to side as it travels forward.

Even through the structure of the neural net had been simplified substantially it was still necessary to learn the weights required to produce a wall following behavior. The approach taken to do so involved parameter adjustment remotely by an observer in real time as the robot ran. The sliders provided by the graphical user interface of CREAL were used to adjust the values of these weights. A Bluetooth wireless serial connection was used to so that the robot may be run untethered and still be able to communicate with the host computer.

The process of trial and error weight adjustment began with setting the initial

weights to values that seemed appropriate given the simplified scenario described above. Throughout multiple runs these weight values were adjusted until a smooth wall following behavior was achieved.

Figure 5-3 shows the robot traveling quite far down the hallway at a constant distance from the wall. It can be seen in the first four frames that the robot is actually swerving back and forth slightly, but that this action repeated multiple times produces motion in a straight line. The wall following that results from this neural net structure is quite smooth.

Chapter 6

A Subsumption Implementation

This chapter will describe the use of a subsumption control architecture on a mobile robot and describe the implementation of a subsumption architecture in the CREAL programming language. The steps taken to reach the desired goal will be described, as will the author's experiences in implementing this control system within CREAL.

Subsumption is a behavior based architecture designed by Rodney Brooks [9]. The basis of the subsumption architecture is to have immediate reflex-like reactions to sensory input as it is read. Independent modules that represent outwardly visible behaviors run in parallel, each one acting when appropriate input values are received. Further stratification is possible by the low-level and the high-level behaviors. Low-level behaviors essentially act at all times while high-level behaviors, whose input triggers are tapped much less often, have the power to override, or subsume, the outputs of the low-level behaviors. This architecture has been used to control robots such as autonomous walking hexapods [7], a map-building wheeled robot [32], a robotic tour guide [20], and an autonomous vaccuum cleaner [22].

The goal of this chapter is not primarily to demonstrate the functionality of the attempted task. It is to report on the success and flexibility of CREAL in supporting subsumption.

6.1 Subsumption and CREAL

CREAL is a programming language that is extremely well suited to accommodate the creation of subsumption control architectures. This opinion applies to both the capabilities of the language in handling the computations required of this architecture and the the way in which this language facilitates programmer use.

Subsumption, as described above, is an architecture that is based upon many concurrently running modules each performing lightweight computation. These modules communicate through the environment (via sensing) and through signals passed between modules or peripheral devices. CREAL provides built in automatic scheduling of lightweight computation that is multi-threaded (See Chapter 3).

The design methodology of independently running subsumption modules correlates directly to CREAL modules running in parallel. The implementation of an architecture of this type within CREAL could easily be performed by interacting CREAL modules that each perform the computation that correlates to a module in the design architecture.

Subsumption method of communication through signals passed from one module to another is analogous to CREAL wires connecting module outports to other module inports. These wires are able to pass signals of any supported CREAL type (either 8-bit or 16-bit) and encompass the types of signals that would be passed in any subsumption system.

CREAL wires also have the ability to perform interactive communication tasks that directly correlate to subsumption communication styles. Suppression of signal content as well as inhibition of module outputs are features that are essential to the functionality of the subsumption architecture. These features are built into CREAL so that wires may suppress or inhibit other values through a fully supported and easy to use method.

These features combined make CREAL a language that will perform with ease the actions required of a subsumption architecture as well as make the experience straightforward and painless for the user implementing such an architecture.

6.2 The Task

The task is one of human companionship and interaction. As this task is being performed a narration of the experience of the robot designer will also take place. The CREAL programming language and STACK hardware platform are extremely new and have been tested and run on relatively few robotic platforms. This narration provides a user's perspective of development of a subsumption-based mobile robot from first testing through completion, as well as the successes and difficulties faces while accomplishing such a task.

It is to be stressed again that the focus of this chapter is not the novel task, but to show the functionality of using a subsumption control architecture written in CREAL to perform such a task and report its Flexibility, reliability and ease of use.

6.3 Implementation

Subsumption has been described as an architecture that is built from the bottom up: the creation of simple lower level behaviors takes place before more complex higher level behaviors are created, and these lower levels serve as the base upon which the complex behaviors are built. This is how we proceeded too, by building layers of behaviors.

Each of our behaviors are encompassed within a layer of the subsumption architecture. One or more modules are used to perform each behavior. For each layer I describe its behaviors, the modules that are necessary to perform each behavior, and the interaction that the modules within this layer will have on modules and wires within lower layers of the architecture (See Figure 6-1).

6.3.1 The Wander Layer: Straight, Turn, Wander

Structural Organization

The most basic behavior that a human-interacting mobile robot would need to perform is that of becoming mobile. In a two wheeled robot, the act of moving straight is

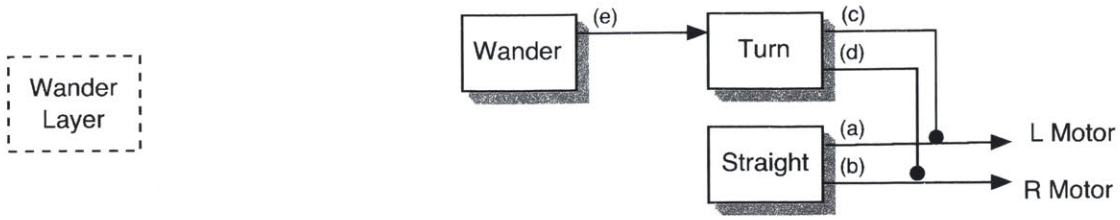


Figure 6-1: Subsumption diagram containing the modules within the Wander layer. Included are the Straight module which outputs a constant value to both the left and the right motor, the Turn module which may subsume the values output by the Straight module and force a turn, and the Wander module that periodically applies a turn of varying intensity for a varying duration.

quite simple: just output the same value to both of the two wheels and linear motion will occur.

This behavior alone is not enough to allow successful movement through an environment. In order to traverse through a large space in search of a desired object (in this case in search of a human), the robot must be able to turn at angles and travel to places that exist off of the initial fixed straight line of travel.

With the ability to move straight ahead and turn with some varying degree the robot would have all of the tools necessary for randomly wandering through an environment. The actions of moving straight and turning would have to be coordinated so that the robot wandered successfully, but with this coordination successful wandering could occur.

In order to implement the wandering behavior within CREAL, modules must be created to perform each of the tasks described above: straight motion, turning motion, and a coordination of these two tasks:

- **STRAIGHT**

  A module was created that produced a steady output of integer values for both the left robot wheel and the right robot wheel. No input is necessary for this module; it will always perform the base behavior of forward motion. For each of the motors there is a wire which sends the output value to a support module responsible for setting the speed of the motor to this value.

- **TURN**

  The task of turning is one that would arise when a more complex control situation arose requiring the robot to alter its straight motion. Examples of this would include observing an object through sensor data and a turn instigated by the module coordinating the wander motion. Turning would therefore have higher priority over the base behavior of straight motion. In a subsumption diagram this may be modelled as the output of the turn module subsuming the output from the forward motion module when turning was necessary. In the absence of a requested turn no values would be sent from the *Turn* module and the default forward speeds would be observed.

  In order to successfully turn the robot, motor control signals would need to be given so that the two wheels of the robot turn at different speeds. Based on the intensity of the turn required the difference in wheel speeds could be very slight or quite large. This difference in motor values could be accomplished through a turn module that had one input port whose received signals represented the intensity of the turn requested, and output ports with values for each of the two motors.

  Figure 6-1 shows the relation between a *Straight* module producing fixed left and right motor output values, and a *Turn* module with one input port and two output ports that each output values that subsume the correlated output values from the *Straight* module.

- **WANDER**

  The last module that is part of this layer is the *Wander* module. This module is responsible for sending signals to the *Turn* module of varying intensities and varying durations. A random number generator within CREAL made it possible to create random numbers within certain ranges repeatedly so that a turn intensity would be randomly chosen, it would be sent to the *Turn* module for a random amount of time, and finally a random amount of time would be spent in which no value was sent to the *Turn* module.

Figure 6-1 also shows the presence of the *Wander* module which has one output that is fed directly into the input of the *Turn* module.

This implementation was created so that the *Straight* module always output a default motor value of 15. This value, as found by trial and error, resulted in forward motion of the robot that was of an acceptible speed for a robot safely touring through office environments. The *Turn* module was implemented so that when a signed integer input signal was received, each of the default motor values would differ either plus or minus this amount (depending on the sign of the input.)

Resulting Behavior

In order to successfully create subsumption layers many test runs must be performed and parameters altered until the resulting behaviors appear correct. There is no one *right* implementation of a behavior such as wandering, for many combinations of parameters and code can lead to behaviors that an observer would deem correct.

In the case of this behavioral layer, tests were run and parameters were adjusted that altered the ranges of values possible for turn strength, turn duration and non-turn duration. Improper wandering parameters could lead to turns being too weak or short in duration (resulting in only a very slight change from travelling in a straight line), turns being too strong or too long in duration (resulting in the robot spinning in place), turns that take place too with little rest time in between (resulting in what appears to be extremely erratic behavior), or too much rest time in between turns. All of these parameter values may be chosen by the programmer so that the action performed by the robot is to their satisfaction.

Figure 6-2 shows images of the wandering behavior resulting from these parameter values. These images were taken from a video of the robot wandering. The displayed frames show the position at fixed amounts of time as it is performing this action.

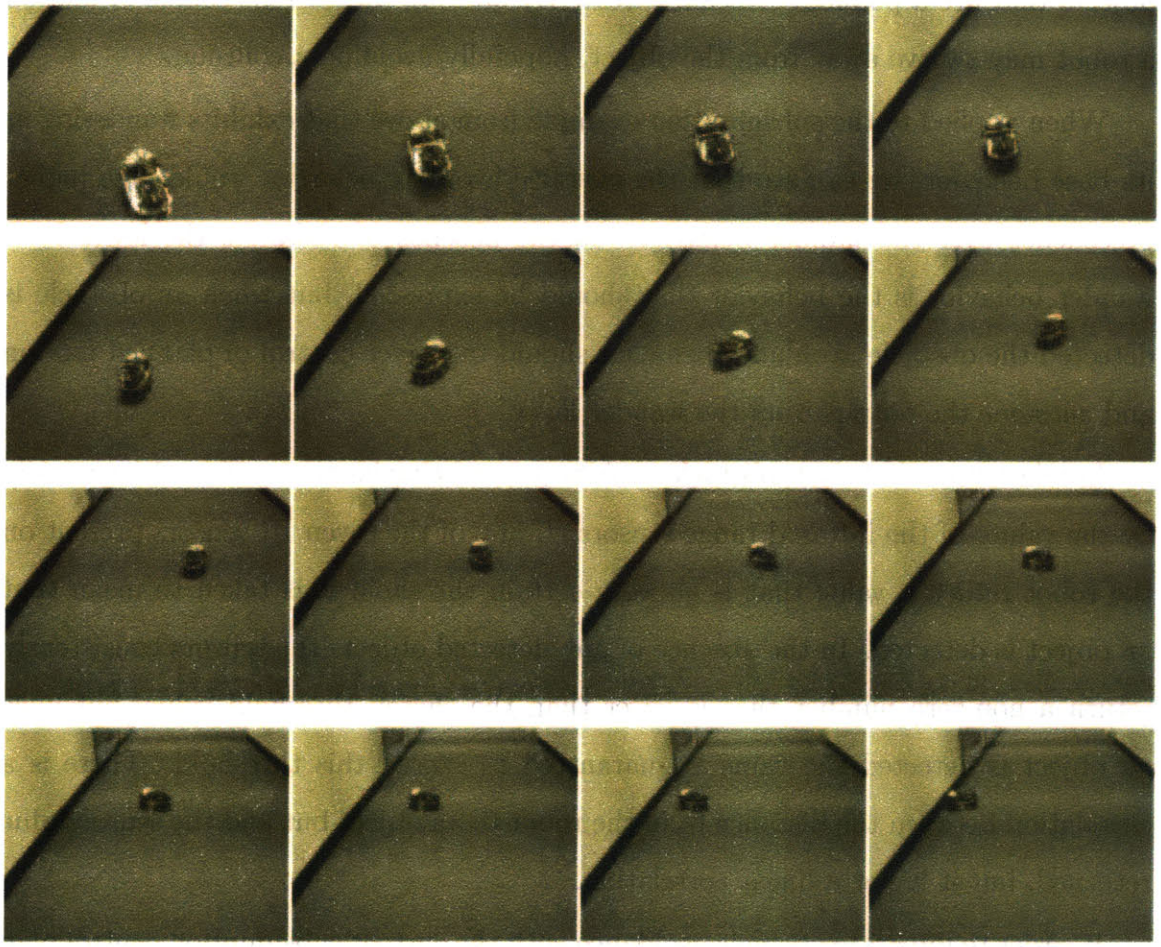


Figure 6-2: Still frames from a video demonstrating the behavior achieved by the Wander layer. Notice that the robot turns right then left, and proceeds to run directly into the wall. No sensors influence the behavior produced by by the Wander layer.

6.3.2 The Obstacle Avoidance Layer

Structural Organization

Obstacle avoidance is a behavior that is essential to most all autonomous mobile robots. This behavior allows robots to travel through environments and react to objects that are a possible collision threat. By altering the current course of motion a robot may swerve away from the object, hopefully avoiding a collision.

When applied to the subsumption example from above that exhibits wandering as its base behavior, we can see that the obstacle avoiding behavior would take higher precedence over the wander behavior. In the absence of obstacles the lower level wander behavior is the behavior that should be expressed, but when an obstacle is detected the obstacle avoidance behavior should take over control of planned motion and *subsume* the output from the wander layer.

In relation to the robot, obstacle avoidance is a behavior that will be driven solely by the values of the infrared range sensors. If any of the seven IR sensors present on the robot return a value that is above a certain threshold it is taken to mean that an object is detected. In the absence of any detected objects the sensors consistently return a non-zero number that is lower than the chosen threshold value, but once an object is detected the value is guaranteed to exceed this threshold. There is a correlation between the distance from the robot to an object but and the sensor value returned, but it is not a linear correlation.

In order to integrate obstacle avoidance with the current subsumption control plan it is necessary that an *Obstacle Avoidance* module be built such that it is able to send signals directly to the *Turn* module, having a higher priority than the signals sent from the *Wander* module. This may be achieved by creating an output wire from the *Obstacle Avoidance* module that subsumes the output wire from the *Wander* module.

The *Wander* module has no information regarding the state of the wire that receives values from its output port so it never knows when its output is being subsumed. The *Wander* module will continuously calculate and send values to the *Turn* module that would lead to a wander behavior, even if those values are in turn being

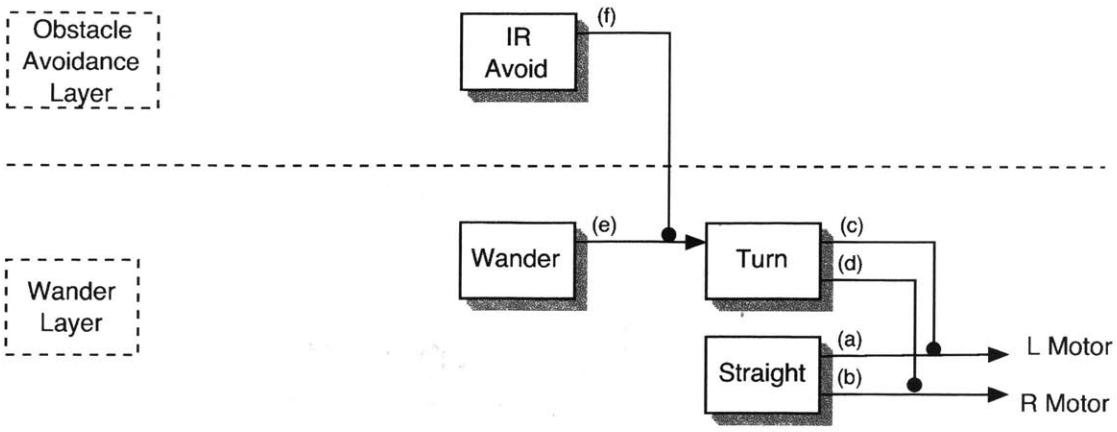


Figure 6-3: Modules within the subsumption layers of Wander and Avoid. The Obstacle Avoidance layer contains only one module: IR Avoid. This module is responsible for watching the IR values and instigating a turn when an obstacle has been detected.

subsumed. If no object is detected by the IR sensors then simply a lack of output from the *Obstacle Avoidance* module will result in the wander behavior to take place.

Figure 6-3 shows the layout of the subsumption architecture with both a lower wander layer (that is the same as before) and a higher priority obstacle avoidance layer. In this figure wire (f) exiting the *Obstacle Avoidance* module subsumes wire (e) leading from the *Wander* module output to the *Turn* input.

Resulting Behavior

In order to create a module that successfully performs obstacle avoidance it is necessary to design an algorithm that will be able to calculate the proper turn value based on the current sensory input values. As mentioned before, the *Obstacle Avoidance* module outputs an integer value that the *Turn* module uses to offset the wheel speeds. Since subsumption modules run concurrently the sensor values at any one given moment should almost immediately influence the motor values. The value that must be output from the *Obstacle Avoidance* module is therefore a measurement of desired turn intensity that will be output. Another method which this algorithm may be compared to is one in which the input values to the *Turn* module represent some length of time that a fixed turning intensity should be held.

In creating this algorithm it was necessary to first analyze the overall relationship

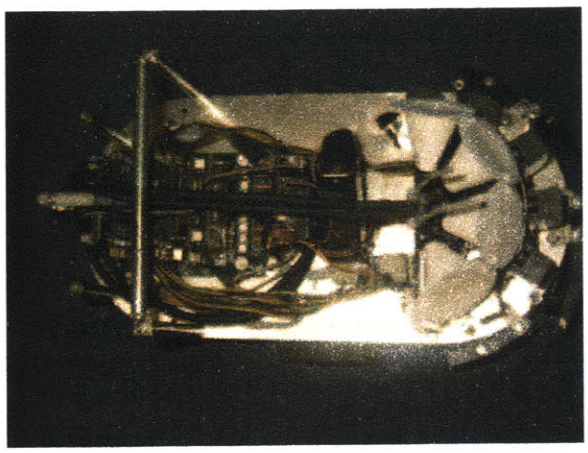


Figure 6-4: Angular placement of the IR sensors on the front of the robot. The seven IR sensors on the left side of the image point radially out from the front of the robot, covering just shy of 180 degrees.

that should exist between the possible sensor readings and the desired motor output values. As shown in Figure 6-4, the seven IR sensors on the robot are arranged so that they are evenly angled across the front 180 degrees of the robot. As an overall guiding principle we can see that if an object is detected in one of the front sensors the turn value should be quite high to avoid a collision, while IR sensor readings made by a side sensor should only result in a slight veer away from the object.

Rather than using raw IR sensor readings to calculate the intensity value at which the robot should turn away, a simplification took place in which IR data was converted into integer values between 0 and 3 that represent threshold values that were met. The thresholding system involved reading the IR sensor values and categorizing the intensity of the sensor as LOW, MEDIUM or HIGH. It was necessary to test the IR sensors to see at what distance an object was recognized and what types of numeric values were returned when the robot was at that fixed distance from the object. Threshold values were initially chosen to encompass the state of the robot in terms of inches from the wall. For instance, sensor readings above 180 (out of a possible 255) were consistently encountered when an IR sensor was brought closer than roughly 9 inches from the wall. These thresholded sensor values were given a numeric value of 0 (for no object detected), 1 (for LOW), 2 (for MEDIUM) and 3 (for HIGH).

Each of the IR sensors was also assigned a weight that correlated the value seen

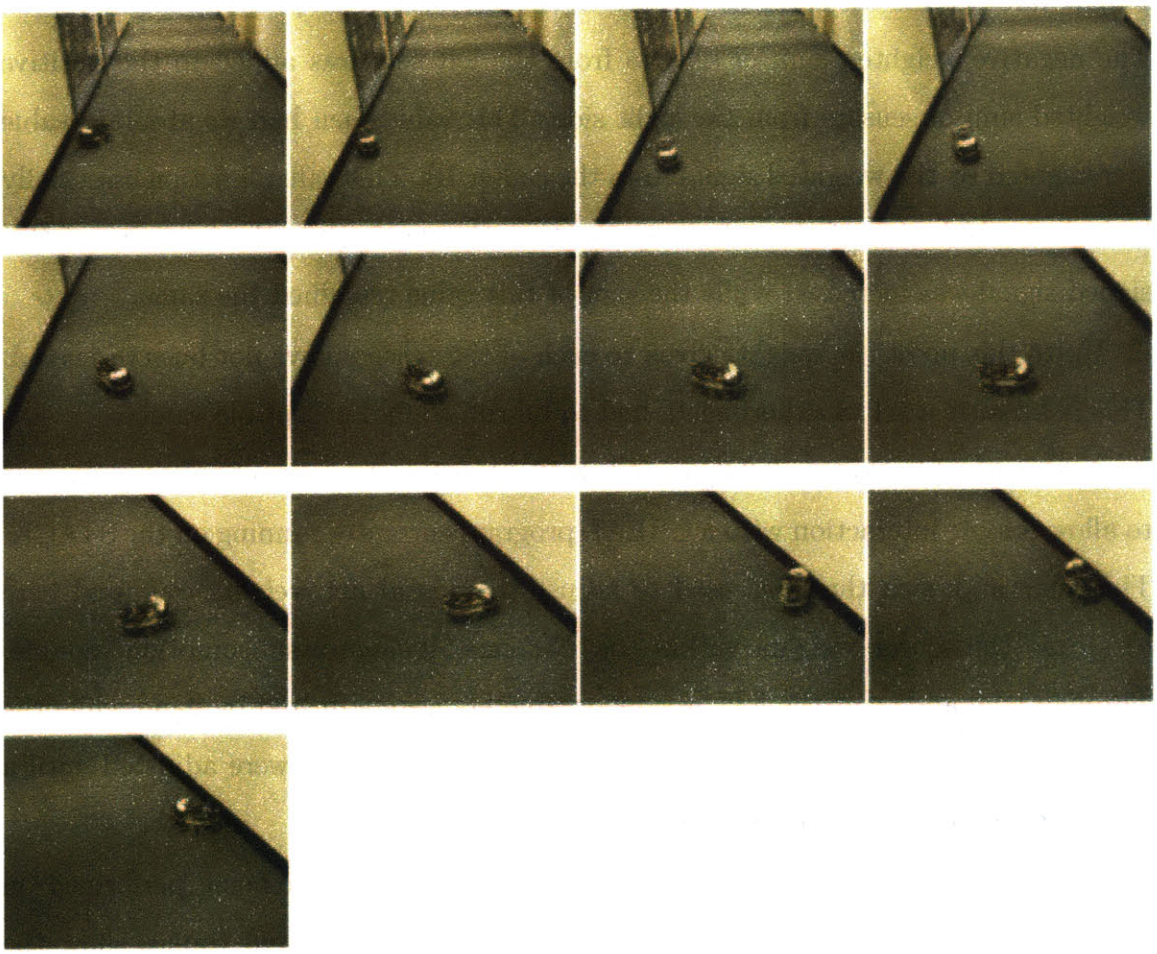


Figure 6-5: Still frames from a video demonstrating obstacle avoidance as well as wandering. The first three frames, as well as the last three frames, show the robot approaching a wall and making a turn before a collision with the wall takes place.

and the intensity at which a turn should take place. The symmetry of the IR sensor placement allowed pairs of sensors angled an even amount away from the center line to share the same weight. This symmetry would lead to no favoritism in calculating the desired turning direction and intensity.

The three IR sensors covering the right side of the robot body were given negative weight values while the three IR sensors on the left were given positive weight values. The negative weighted sum of sensors from the left side was added to the positive weighted sum of sensors from the right side. This value then had its absolute value incremented by the weighted signal from the center IR. This value function essentially looked at which side of the robot says to turn the most, then the intensity of this signed signal was increased while the sign of this value remained the same.

Up to this point the weights given to each of the sensors have not been discussed. This is because a set of experiments had to be run to find acceptible values.

As described in Section 3.3.6, a wireless Bluetooth serial connector was connected to allow remote interaction with a CREAL program currently running on the STACK. The same approach that was used to adjust the neural net weights was used here to adjust the sensor weights within the *Obstacle Avoidance* subsumption module. Through trial and error the four weight parameters were adjusted by hand while the robot was performing obstacle avoidance. These weight values were adjusted until a behavior deemed acceptible was reached.

Figure 6-5 shows images of the obstacle avoidance behavior being performed on the robot with these final parameter values used. The displayed frames show the position of the robot every half second.

6.3.3 The Wall Hit Layer

Structural Organization

The next layer to be described is one which produces a behavior that allows the robot to recover if a collision is ever made with a fixed object. The obstacle avoidance layer helps deter any possible collisions, but if one were to happen it would be necessary

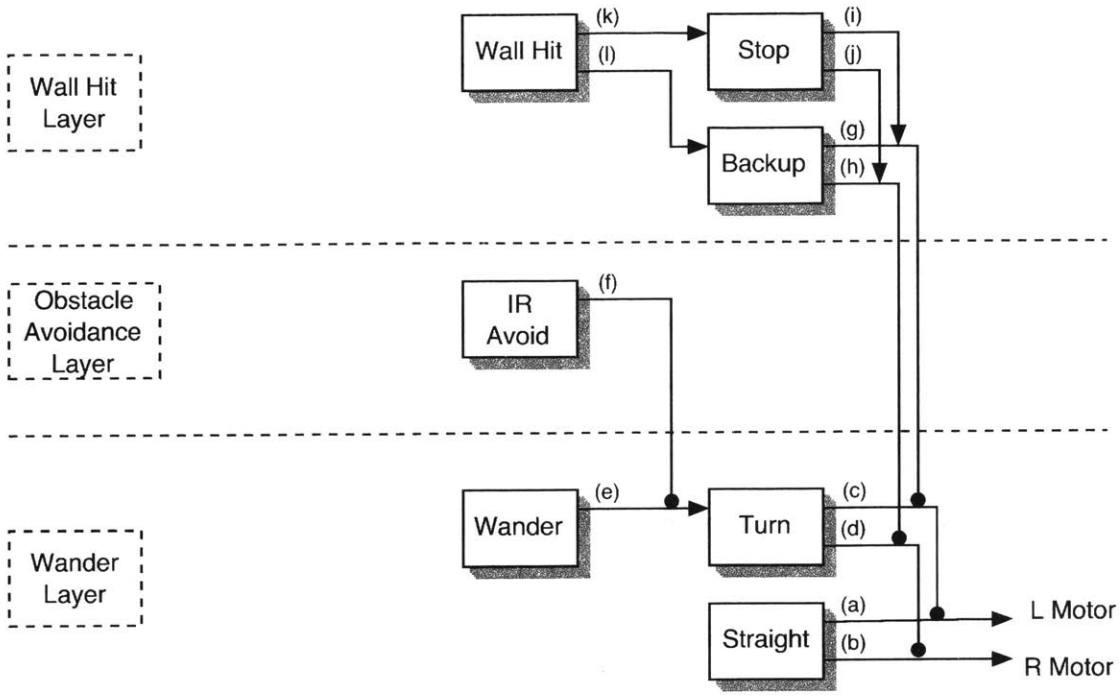


Figure 6-6: Modules within the subsumption layers of Wall Hit, Avoid and Wander. The newly added Wall Hit layer contains three modules: the Wall Hit module that monitors the status of the analog bump sensors and sends output to both the Stop and Backup modules. The Stop and Backup modules both act to subsume the values that are sent to the left and right motors from the lower layers of Wander and Avoid.

that the robot respond accordingly. When the robot is within a certain small distance of a wall it is often the case that the IR sensors return faulty data. In a case such as this the robot could hit the wall and continue to attempt to move forward due to the fact that the IR sensors do not *see* the wall. Rather than continuously attempting to move forward an appropriate behavior would be to back up to a distance in which reliable IR data could be received then let the obstacle avoidance behavior turn the robot from the wall.

This behavior may be achieved by adding a collision recovery layer that contains three nodes. This layer would monitor the state of the analog bump sensors placed along the front edges of the robot. In response to activation by a bump sensor the *Wall Hit* node would output a signal that would force the robot to move in reverse for a fixed amount of time. This output would subsume the values output by the obstacle avoidance layer and the wander layer.

Figure 6-6 shows the *Wall Hit* module sending signals to the *Stop* and the *Backup* modules. The wires (g) and (h) output by the *Backup* module suppress wires (c) and (d) from the *Turn* module, having the effect of directly modifying the output desired motor values.

Wires (i) and (j) from the *Stop* module subsume wires (g) and (h) from the *Backup* module. This design was chosen so that behavior layers higher than Collision Avoidance would be able to control the robot to stop if needed through direct input to the *Stop* module.

In summary, the *Wall Hit* module acts in response to an activated bump sensor by commanding the robot to backup for a fixed short amount of time then momentarily stop. After this control sequence the Collision Recovery goes back into a rest state waiting to act in response to another activated bump sensor.

Resulting Behavior

The behavior that results from the addition of the Collision Recovery layer is one in which in the absence of an activated bump sensor the lower behavior layers keep the robot wandering and avoiding objects. When an obstacle is hit the robot backs up and turns away from the object. It is important to note that the Collision Recovery behavior layer does not explicitly control the robot to turn away from the object, just to briefly move in reverse. Once the robot has moved sufficiently in reverse the Collision Recovery layer stops sending signals that subsume the lower layers. When this happens we find the robot to be in a position in which the IR sensors may then read correct values and notice the wall or object that was hit. The Obstacle Avoidance behavior will take over and the robot will turn away from the object.

Figure 6-6 shows a time lapse sequence of evenly spaced frames from a filmed video in which the robot performs the collision recovery behavior.

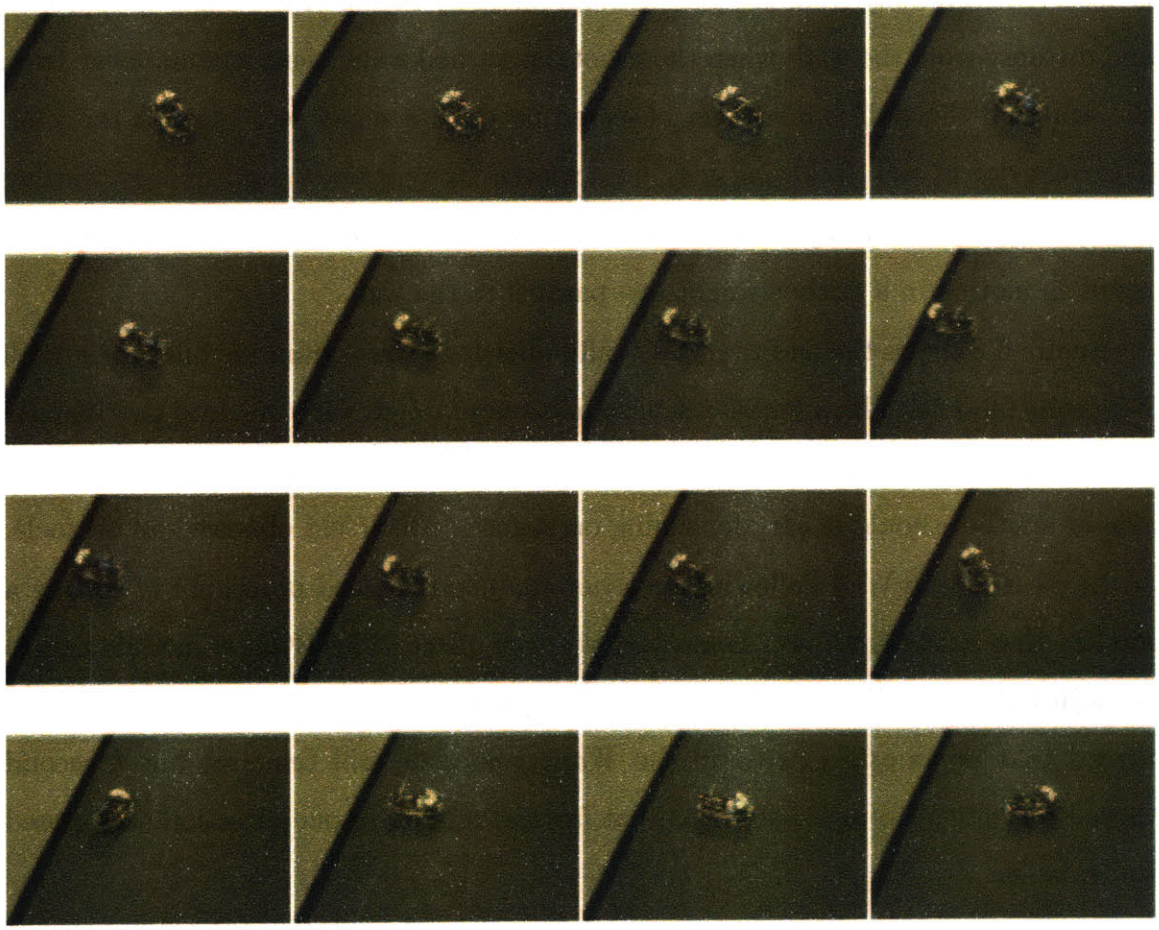


Figure 6-7: Still frames from a video demonstrating the behavior sequence that occurs when an obstacle is hit. Frame 8 shows the robot making contact with the wall, followed by two frames of the robot backing up, then three frames of a quick turn in place so the robot is then facing away from the wall.

6.3.4 The Wall Following Layer

Structural Organization

The approach used to achieve wall following is the same basic idea that was used in the implementation of wall following in neural nets. The basic overview is this: if a wall is observed turn away from the wall with an intensity fitting of the angle of approach and distance from the wall; when the robot has turned a sufficient amount away from the wall and the wall is no longer observed produce a slight veer towards the wall. The behavior of wall following may be achieved by repeatedly and quickly turning toward the wall and away from the wall. This oscillation, if fine tuned correctly, will result in motion in a fairly straight line parallel to the wall.

Figure 6-8 shows the addition of a Wall Following layer containing one single *Wall Following* module. This module sends signals out through wire (m) into the *Wander* module. This signal in turn suggests to the *Wander* module that a slight veer to one side or another should occur, depending on the side which last observed an obstacle.

In creating the Wall Following behavior the non-standard procedure of editing a module that exists in a lower behavior layer took place. The *Wander* module had to be adjusted to handle an input value that represents a desired veer direction. The thread that previously existed in the *Wander* module still remains, but a second thread was added to produce a bias that favored motion in one direction as opposed to the other.

Resulting Behavior

As opposed to the notably smooth wall following behavior produced by the neural net wall following implementation (see Section 5.5), the wall following produced by the addition of this new behavior layer was jumpy and somewhat erratic. The oscillation between the behaviors that move the robot away from the wall obstacle and back toward the wall through a slight veer was visually observable to a spectator.

The two interacting behaviors are performed quite differently. The veer towards the wall was usually a smooth and gentle curve in one direction, while the turn from

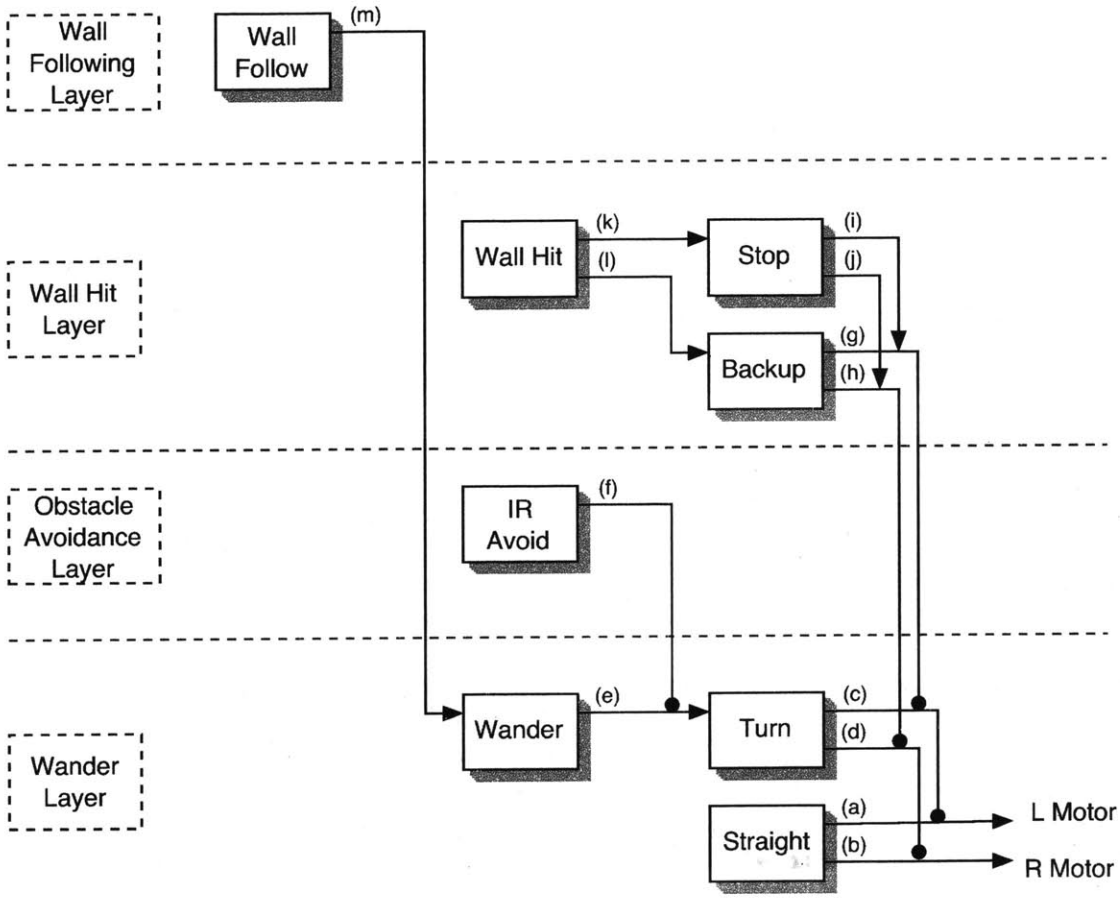


Figure 6-8: Modules within the subsumption layers of Wall Following, Wall Hit, Obstacle Avoidance and Wander. The Wall Following layer contains only the Wall Following module that sends a value to the input port of the Wander module. The value sent correlates to walls detected by the infrared sensors (monotored from within the Wall Follow module) and the direction in which the robot should veer to follow the wall.

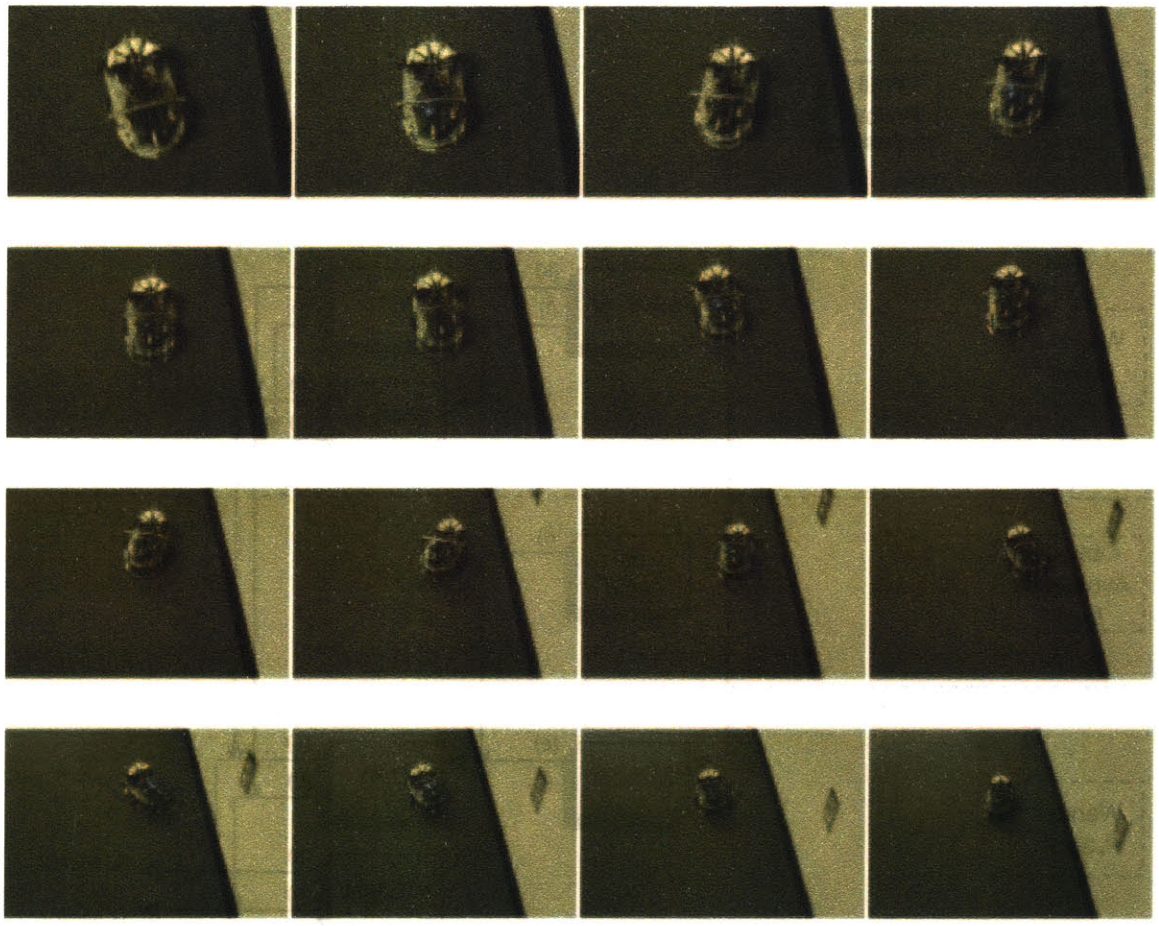


Figure 6-9: Still frames from a video demonstrating the wall following behavior implemented within subsumption. It can be seen that as the robot moves farther away the angle at which the robot is veering from left to right becomes more intense and noticeable. Nonetheless, the robot is successfully traveling along the wall at a fairly constant distance away from the wall.

Figure 6-10: The robot following walls with the Wall Follow, Wall Hit, Avoid and Wander modules

the wall tended to be intense and brief. Depending on the angle of approach the turn from the wall could be drastically different in nature from the gentle veer.

This is not to say that successful wall following never occurred, because in some narrow cases it worked quite well. If the robot began facing parallel to the wall the obstacle avoidance behavior a slight veer away from the wall would result. This, when paired with the slight veer toward the wall that occurs when the wall is out of range of the IR sensors, resulted in a fairly smooth wall following behavior.

When the angle of approach was not parallel, the obstacle avoidance behaviors from lower layers in the subsumption diagram result in a turn that is strong enough to angle the robot at roughly the same angle away from the wall as the angle of approach. When the robot resumes moving from the wall at this angle a veer towards the wall is performed, resulting in a arc or half circle of motion that will leave the robot at an even more perpendicular angle of approach to the wall.

Ideally an obstacle avoidance turn will leave the robot angled enough away from the wall that it may continue moving without being in danger of hitting the wall again. The oscillation of veering towards and away from the wall would ideally lead to a more smooth wall following behavior. In this case seen here, the lower obstacle avoidance layer produces a turn from the wall that is so intense that the robot is left facing quite away from the wall, making it difficult to integrate seamlessly with the veering.

Figure 6-10 shows still frames from a video capturing a successful wall following behavior.

6.3.5 The People Chasing Layer

The four pyroelectric sensors mounted on the front part of the robot are used in this layer to detect the motion of people and respond by moving in their direction. Pyroelectic sensors, as mentioned earlier, do not return values related to the distance of the robot to that person, but related to the speed at which the robot witnesses a person moving by. The default sensor value, in absence of any detectable motion, is near 128. When a detectable object moves past, the pyro is able to detect the speed

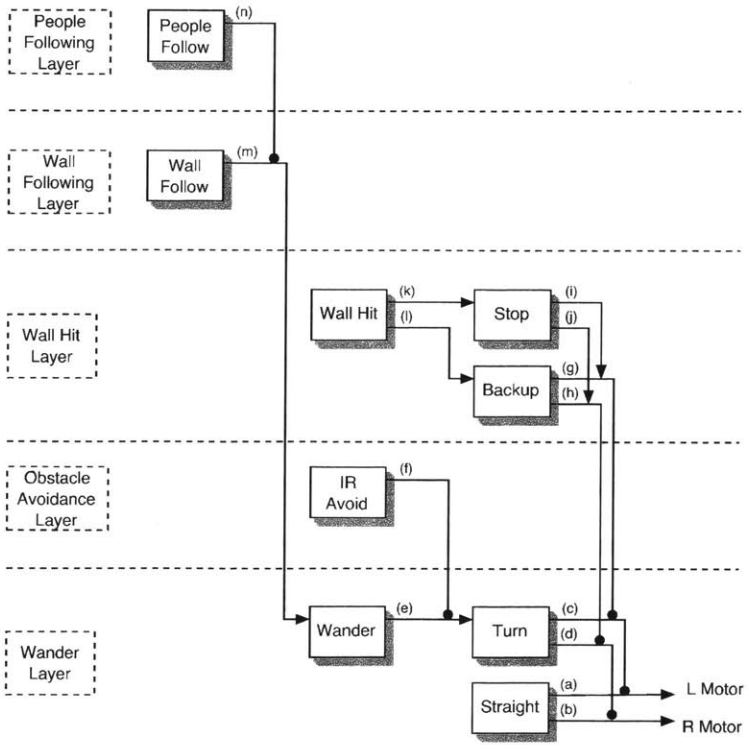


Figure 6-11: Modules within the subsumption layers of People Following, Wall Following, Collision Recovery, Avoid and Wander. The People Following behavior layer consists of one module, the People Follow module, that monitors the pyroelectric sensors. If a person is detected values are sent to the Wander module (subsuming the output of the Wall Follow module) so that the robot veers in the direction of the seen person.

at which that object passes perpendicularly to the sensor's horizon plane.

Structural Organization

When testing the pyros it was noticed that when a person walks past the sensor in one direction the value goes through the sequence of jumping from near 128 to a much higher value, then to a much lower value, followed again to a resting state middle value. A person passing in the other direction will result in the sensor first falling low then high before returning to the middle value.

To respond to this type of information a subsumption module was added which allowed its output directly influence the way in which the robot wandered, just as the *Wall Follow* module did. In Figure 6-11 we can see that wire (n) subsumes the

wire (m) that is the output of the Wall Following layer. This can be interpreted as the action of veering towards a recently seen person to take priority over the action of veering towards a recently seen wall.

In order to recognize the pattern of jumping high then low (or low then high) threads were created that kept constant watch over the values exhibited by the pyro-electric sensors. Once every fixed amount of time (0.2 seconds in this case,) a review would be gathered to calculate the maximum difference in sensor values that was reached during that time epoch. If the absolute difference in highest and lowest value received was more than a preset threshold then that sensor would have officially 'seen' a person. The direction in which a person walks past the robot regulates if a high value will be seen before a low value or vice versa.

The theory between led us to choose this design rationale is very similar in nature to the wall following technique of gently stradling the line of turning away from the wall when too close and veering towards it when too far away. When a person is seen passing the robot the resulting behavior is a veer in the direction of the person. If the robot were to veer too far then one of the sensors would detect that it had passed the person moving in the other direction and a veer in the opposite direction as before would result. If the person is always within a detectable distance from the robot then this oscillation between veering in opposite directions will result in the robot moving toward the person.

Resulting Behavior

To test the behavior of the robot one simply had to walk past it and see how it responded. If no response was displayed then that would indicate that the threshold parameter within the *Wall Follow* module was set too high. Multiple runs were taken in which this one parameter value was adjusted to result in a more effective behavior.

As seen in Figure 6-12, the robot performs the behavior of moving after its target quite well. The motion was smooth and fluid and integrated well into the types of motions resulting from the lower subsumption layers. In this figure it is possible to see that the direction the robot is facing and moving in changes slightly from frame

Figure 6-12: The robot chasing a person with the People Following, Wall Following, Wall Hit, Avoid and Wander modules. We notice that although the robot is veering from left to right it remains in the middle of the hallway and moves in the direction that the person is walking, staying a consistant distance behind the moving person.

to frame as the robot alternates between veering to the left and to the right.

Chapter 7

Future Work

Throughout the course of this research, ideas for future changes, design alternatives, and capability enhancements presented themselves. This is due, in large part, to the flexibility that the STACK and CREAL provide.

The robot itself serves as an easily manipulated testbed for various programming experiments. It would be interesting to use this platform to do a comparative study on the efficacy of different learning schemes such as Bayes Net Learning, and other more statistical approaches to function maximization. This presents one small problem in that the Rabbit does not support the mathematical functions required for these methods. Though switching processors would require, for instance, a new bios and a new STACK backend, the other parts of the STACK could remain intact. The work that would go into adapting the STACK to suit the architecture of a more mathematically powerful processor could result in work on more computationally and mathematically intense problems. And at the same time, much of the system would stand as an experimental control. On the other hand, the robot could also be changed to try different types of locomotion and completely different behaviors.

The form factor of the present robot lends itself to the possibility of working with networks of mobile robots. This kind of system would be an ideal platform for class projects and research groups. It is easily and inexpensively fabricated, and can be readily available.

Bibliography

[1] Rabbitcore rcm2300 user's manual. http://www.rabbitsemiconductor.com/.

[2] Arvin Agah and George A. Bekey. A genetic algorithm-based controller for decentralized multi-agent robotic systems. In International Conference on Evolutionary Computation, pages 431–436, 1996.

[3] Ronald C. Arkin and Douglas C. MacKenzie. Planning to behave: A hybrid deliberative/reactive robot control architecture for mobile manipulation. In ISRAM '94, Maui, Hawaii, 1994.

[4] Dave Baum. NQC Programmer's Guide (version 2.5).

[5] R. Brooks. Proposed creal communication protocol, July 2003.

[6] R. Brooks, L. Aryananda, A. Edsinger, P. Fitzpatrick, C. Kemp, U. O'Reilly, E. Torres-Jara, P. Varshavskaya, and J. Weber. Sensing and manipulating built-for-human environments. International Journal of Humanoid Robotics, November 2003.

[7] Rodney A. Brooks. A robot that walks: Emergent behaviors from a carefully evolved network. In Proceedings of the IEEE International Conference on Robotics and Automation, pages 692-94, May 1989.

[8] Rodney A. Brooks. The behavior language; user's guide. MIT Artificial Intelligence Lab Memo No. 1227, April 1990.

[9] Rodney A. Brooks. Intelligence without representation. Artificial Intelligence Journal, 47:139 159, 1991.

[10] Rodney A. Brooks. Creature Language. MIT Artificial Intelligence Laboratory, Cambridge, MA, June 2003.

[11] Rodney A. Brooks and Charles Rosenberg. L -a common lisp for embedded systems. In Association of Lisp Users Meeting and Workshop LUV'95, August 1995.

[12] Marco Colombetti and Marco Dorigo. Robot Shaping: Developing Situated Agents through Learning. Technical Report TR-92-040, Berkeley, CA, 1993.

[13] M. Doherty, M. Greene, D. Keaton, C. Och, M. Seidl, W. Waite, and B. Zorn. Programmable ubiquitous telerobotic devices. In Proceedings of SPIE Telemanipulator and Telepresence Technologies IV, volume 3206, pages 150 157, October 1997.

[14] Jerry Schumacher Don. Teaching introductory programming, problem solving and information technology with robots at west point.

[15] J. Donnett and T. Smithers. Lego vehicles: A technology for studying intelligent systems. In From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, pages 540 549, Cambridge, MA, 1991. MIT Press.

[16] Marco Dorigo and U. Schnepf. Genetics-Based Machine Learning and Behaviour Based Robotics: A New Synthesis. IEEE Transactions on Systems, Man and Cybernetics, 23(1):141 154, 1993.

[17] P. Gaudiano, E. Zalama, and J. Coronado. An unsupervised neural network for low-level control of a wheeled mobile robot: Noise resistance, stability, and hardware implementation.

[18] P. Gaussier, S. Moga, J. Banquet, and M. Quoy. From perceptionaction loops to imitation processes: A bottom-up approach of learning by imitatione. Technical Report FS-97-02, Socially Intelligent Agents, 1997.

[19] Living Machines Group. Alive stack user manual, February 2003.

[20] Ian Horswill. Polly, a vision-based artificial agent. In Proceedings of the AAAI-93, pages 824 829, Washington, DC, 1993.

[21] Susan P. Imberman. Teaching neural networks using lego-handyboard robots in an artificial intelligence course.

[22] iRobot Corporation. Roomba Intelligent FloorVac Owner's Manual, 2002.

[23] Joseph L. Jones and Anita M. Flynn. Mobile Robots: Inspiration to Implementation. A K Peters, Ltd., 1993.

[24] K-Team S.A., Ch. de Vuasset, CP 111, 1028 Preverenges, Switzerland. Khepera II Manual, Jan 2002.

[25] K-Team S.A. Khepera II Programming Manual, Jan 2002.

[26] et. al Laverde, editor. Programming Lego Mindstorms with Java. Syngress, 2002.

[27] Henrik Hautop Lund and Luigi Pagliarini. Robot soccer with LEGO mindstorms. Lecture Notes in Computer Science, 1604:141–??, 1999.

[28] P. Maes. How to do the right thing. Connection Science Journal, Special Issue on Hybrid Systems, 1, 1990.

[29] Fred G. Martin. The Handy Board Technical Reference, 2000.

[30] Fred G. Martin. About Your Handy Cricket, 2002.

[31] Fred G. Martin. Intro to Cricket Logo, 2002.

[32] Maja Mataric. Behavior-based control: Main properties and implications, 1992.

[33] Michael Meadhra and Peter J. Stouffer. Lego Mindstorms for Dummies. For Dummies, 2000.

[34] A. Nagchaudhuri, G. Singh, M. Kaur, and S. George. Lego robotics products boost student creativity in precollege programs at umes, 2002.

[35] Stefano Nolfi and Dario Floreano. Coevolving predator and prey robots: Do "arms races" arise in artificial evolution? Artificial Life, 4(4):311–335, 1998.

[36] Stefano Nolfi and Dario Floreano. Evolutionary Robotics: the Biology, Intelligence, and Technology of Self-Organizing Machines (Intelligent Robotics and Autonomous Agents). MIT Press, November 2000.

[37] Jonathan Rees and Bruce Donald. Program mobile robots in scheme. In Proceedings of the 1992 IEEE International Conference on Robotics and Automation, pages 2681–2688, Nice, France, 1992.

[38] J. K. Rosenblatt. DAMN: A distributed architecture for mobile navigation. In Proc. of the AAAI Spring Symp. on Lessons Learned from Implememted Software Architectures for Physical Agents, Stanford, CA, 1995.

[39] S. J. Rosenschein and L. P. Kaelbling. A situated view of representation and control. In P. E. Agre and S. J. Rosenschein, editors, Computational Theories of Interaction and Agency, pages 515–540. The MIT Press: Cambridge, MA, USA, 1996.

[40] T. Ziemke, J. Carlsson, and M Boden. An experimental comparison of weight evolution in neural control architectures for a garbage collecting khepera robot, 2000.