Report Number: WUCSE-2005-17

2005-04-28

Programming Robots by Description and Advice

Andrew J. Martignomi III

Programming robots to perform tasks autonomously is difficult. The environment or even the task may change at any moment. The main drawback is that this programming requires a team of highly skilled roboticists to monitor the robot and change its programming to accomplish the task. The system presented here allows robot controllers to be constructed by a non-specialist, using the included restricted natural language parser. The controller can further be refined by a non-specialist using keywords which represent the changes that each parameter makes to the behavior. To show that the system is viable, controllers made by the system... Read complete abstract on page 2.

Programming Robots by Description and Advice

Andrew J. Martignomi III

Complete Abstract:

Programming robots to perform tasks autonomously is difficult. The environment or even the task may change at any moment. The main drawback is that this programming requires a team of highly skilled roboticists to monitor the robot and change its programming to accomplish the task. The system presented here allows robot controllers to be constructed by a non-specialist, using the included restricted natural language parser. The controller can further be refined by a non-specialist using keywords which represent the changes that each parameter makes to the behavior. To show that the system is viable, controllers made by the system are compared to direct programming on two common robot tasks. The assembled and tuned controllers are shown to perform similarly to direct programming in performance, even better in several cases, but without the large development/testing time that direct programming requires.

2005-17

Programming Robots by Description and Advice, Master's Thesis, May 2005

Authors: Martignoni III, Andrew J.

April 28, 2005

Abstract: Programming robots to perform tasks autonomously is difficult. The environment or even the task may change at any moment. The main drawback is that this programming requires a team of highly skilled roboticists to monitor the robot and change its programming to accomplish the task. The system presented here allows robot controllers to be constructed by a non-specialist, using the included restricted natural language parser. The controller can further be refined by a non-specialist using keywords which represent the changes that each parameter makes to the behavior. To show that the system is viable, controllers made by the system are compared to direct programming on two common robot tasks. The assembled and tuned controllers are shown to perform similarly to direct programming in performance, even better in several cases, but without the large development/testing time that direct programming requires.

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

PROGRAMMING ROBOTS BY DESCRIPTION AND ADVICE

by

Andrew J. Martignoni III, B.S. C.S., B.S. Phys.

Prepared under the direction of Professor William D. Smart

A project presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2005

Saint Louis, Missouri

Contents

Chapter 1

Introduction

Observers of mobile robots have an abundance of insight on how to perform tasks, especially when they see the robot fail. This insight could be useful advice for the robot. Most tasks for a mobile robot tend to be similar to tasks that humans themselves have done, so in that sense most humans can be good teachers. Mobile robots, for the most part, are controlled by programs written ahead of time to perform a set task in a set way.

1.1 Getting a Robot to Perform a Task

When we want to make a mobile robot perform a task, we program the computer in the robot at a low level to attempt to solve the task. As in any computer application, the program code must be debugged to remove any errors. In mobile robotics applications, the behavior of the robot must also be debugged, since it is not always clear how the sensors and motors of the robot are affected by or affect the world. This level of programming requires expertise and acquiring the level of expertise required costs money.

The program written for the robot is typically a fixed solution for the given task. It cannot be changed without going through the development process again, incurring more costs. In addition, a change in the environment can cause performance degradation and possibly failure of the system.

1.2 A Better Way

Ideally, we would like people with little or minimal training to be able to instruct the robot. We would like this instruction to be at a behavioral level, such as “Follow the truck, and stay away from trees.” We would expect to see good performance from the robot. Since the task specification might not be exactly right, the robot may fail at some part of the task and so needs to be capable of adapting to improve its behavior. This adaptation should be directed from a very high level, i.e. “Good robot” or “Move faster”.

After working with mobile robots using the low level programming method, we have observed some aspects of the larger problem. Two such observations are the inspiration for this work. First, many common mobile robot tasks have similar characteristics, such as keeping a sensor reading at a pre-defined value or balancing two sensor readings. As one would expect, since the tasks are similar, the programs to solve them have similar portions of program code. Also, a description of the desired task is a behavioral specification of that task, which leads to the second observation. There can be a direct mapping between words in the natural language specification and the portions of program code which achieves the task, based on the semantics of the specification. The work in this project attempts to use that information to create a system which allows robots to be trained as in § 1.2.

We have created a system to behaviorally specify a task in limited natural language. Each of the words recognized by the system is mapped to a specific portion of executable program code. The system contains a fixed set of these code modules. They are composed into a complete program based on the limited natural language task description. Once composed, the completed program can be run on the robot. The programs generated in this manner provide reasonable behavior, but still are programs which provide a fixed control policy. Because this is a coarse way to specify code, the resulting behavior is not well-optimized. The system provides a method of tuning the performance by changing the parameters on which the system depends. The behavioral advice, such as "Go faster", is mapped to a specific parameter change. Chapter 2 explains how the system composes the program from the limited natural language task description. Chapter 3 looks at the possible changes to the initial program. In Chapter 4, we discuss the limitations of the current system. Results are reported in Chapter 5.

Chapter 2

Constructing a Starting Point

Since, in essence, the goal is to communicate a solution to the robot, the first step is to pick a starting point, i.e. a strategy of controlling the robot. The starting point should allow the trainer and robot both "see" the same events[1], while keeping the robot moving through the environment. In this system, the controller that serves as the starting point is selected to ensure that the robot is safe from harm (i.e. hitting objects, falling down stairs, etc.) and reacts to the objects related to the task in *some* way. The system accomplishes this by combining bits of executable code based on a few key words from the trainer.

2.1 Overview

The first step to constructing a controller from a task description is the decoding process. The system uses a limited natural language for its input from the trainer. Using a simple grammar, each symbol of input is given a meaning. In this framework, each word has a single meaning and part of speech. For example, the word "Ball" is labeled as a noun and is semantically linked to the component which tracks a ball. Some words' meanings are not important, such as the articles "a", "an", and "the", which are used only to help the grammar parse the task description. See Figure 2.1 for an example of tagging words of a sentence with their part of speech.

The system uses two sentence structures:

[1]Not that the robot actually sees the situation like the human, but that the robot's sensors measure the same event that the human sees.

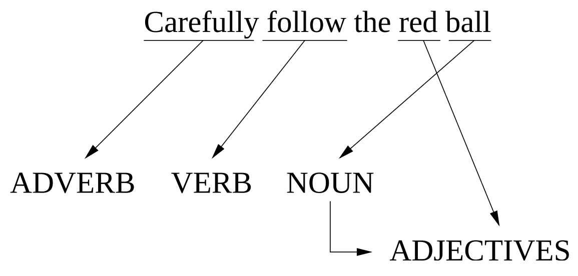

Figure 2.1: A diagram showing the parts of speech.

$$
\begin{array}{rcl}
\text{S} & \rightarrow & \text{Sentence} \\
\text{Sentence} & \rightarrow & \text{VERB Object} \\
 & | & \text{VERB Object ADVERB} \\
 & | & \text{ADVERB VERB Object} \\
\text{Object} & \rightarrow & \Lambda \\
 & | & \text{NOUN} \\
 & | & \text{Adjectives NOUN} \\
 & | & \text{ARTICLE NOUN} \\
 & | & \text{ARTICLE Adjectives NOUN} \\
\text{Adjectives} & \rightarrow & \text{Adjectives ADJECTIVE} \\
 & | & \text{ADJECTIVE}
\end{array}
$$

Figure 2.2: A sample grammar which parses task descriptions.

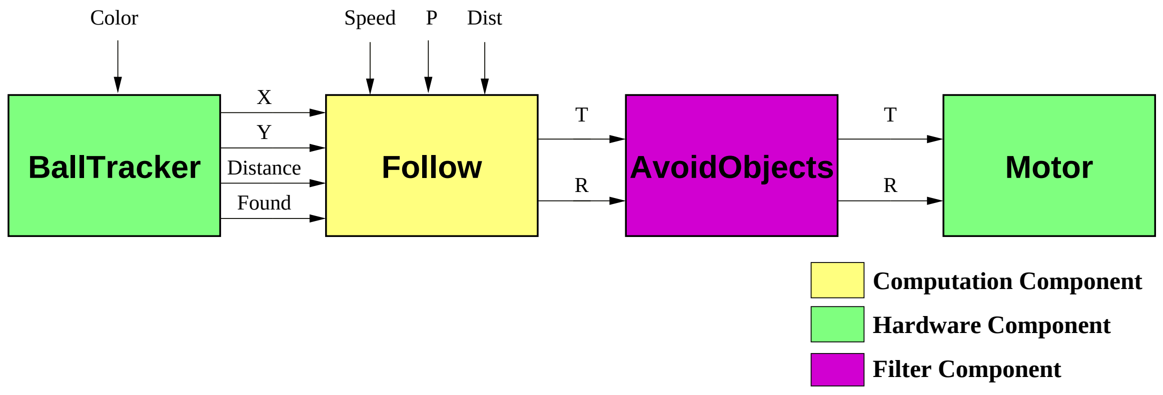

Figure 2.3: A complete controller for following a ball.

1. [ADVERB] VERB [NOUN[<ADJECTIVE(s)>]]

2. VALUE RELATION VALUE

For example, the task “Carefully follow the red ball.” is rearranged as in Figure 2.1. The sample grammar in Figure 2.2 is a subset of the grammar used to parse and output form 1. Nouns, verbs, and adverbs are replaced with executable code from the system. This portion of code is called a component of the system, discussed in more detail in § 2.6. Adjectives have a meaning that modifies a components’ interaction with the controller.

Once each of the components have been instantiated, the controller must be ‘wired’ together. The example has only three components, from “follow”, “ball”, and “Carefully”. The system instantiates each object and connects it based on the number of inputs and outputs. A noun’s outputs have standard meanings which specify a locatation in the environment. A verb takes zero or more nouns as inputs and has motor commands (actions) as outputs. Since adverbs modify verbs, they become filters on the verb’s output, taking motor commands in and outputting the modified motor commands. A sample controller for the example is shown in Figure 2.3.

Each of the component parameters shown across the top in Figure 2.3 can be tuned to enhance the controller's performance. Each component provides its own key words which are used to tune the parameter. For example, a proportional controller might use the words "More Aggressively" and "Less Aggressively" to allow the trainer to tune the gain. These words map directly to parameter value changes in the component. More complex setups could be useful, but are outside the scope of this work and are unnecessary for the controllers used here.

2.2 Parsing the Words

In this work, the natural language input comes from the trainer typing sentences on the keyboard. Using this method avoids the additional problems associated with speech recognition. The decoding process begins by determining each word's associated part of speech, such as noun, verb, etc. This is accomplished by a simple lookup, not through grammar. The list of words used by the system and their parts of speech are listed in Appendix B. The stream of these parts of speech is parsed using an LALR(1) parser with the grammar shown in Appendix C. After parsing and rearranging the words as discussed in § 2.1, "Carefully follow the red ball", or ADVERB VERB ARTICLE ADJECTIVE NOUN, becomes "Carefully" "Follow" "ball"<"red">, or ADVERB VERB NOUN<ADJECTIVE>, which fits the first form of task descriptions. The first component in the list will have its outputs connected to the `Motor` component, allowing the components to control the robot.

2.3 The Meaning of the Words

The meaning of the words is looked up from a table given to the system. The complete list of meanings is given in Appendix D. Each word is given exactly one meaning, just as each word is given one part of speech, limiting the complexity of natural language that can be processed. The meaning of a word falls into one of three categories:

1. **The name of a component.** Whenever this word appears in a sentence, the named component will be part of the final controller.

2. **A parameter value.** If this word appears linked (through the grammar) to a component, then the parameter's default value will be replaced with the value here.

3. **A number.** Some sentences may use literal numeric values, like "twenty one point seven" which must be translated to 21.7.

For example, the word "ball" is given the associated meaning of "BallTracker", which is the name of a component which uses a color blob tracker and returns information about the largest blob of a given color, determined by its one parameter. The word "red" has a meaning of "%Color=0" which means that the *Color* parameter of a component should be set to zero. When these words are used together, "red ball" means a `BallTracker` component with its parameter set to zero, which is the index into the blob tracker's color table representing the color red.

To construct the controller for the task "Carefully"# "follow" "ball"<"red">, each word in the description is replaced with its labeled meaning, giving the component list `AvoidObjects#`

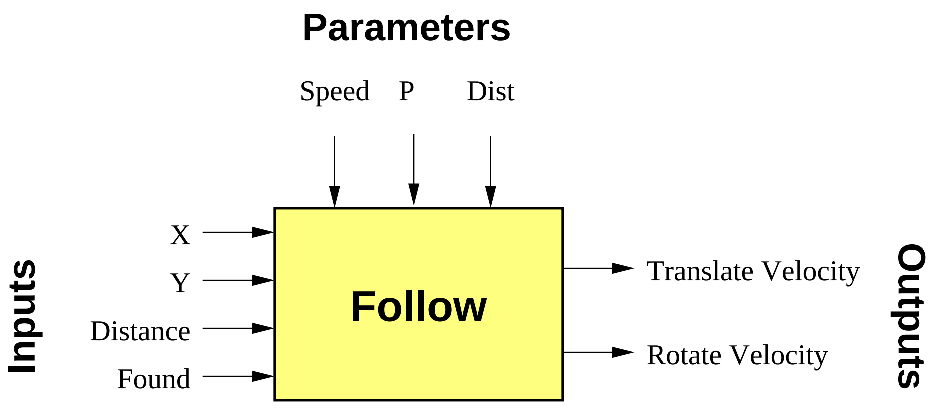

Figure 2.4: A component implementing the 'follow' behavior.

`Follow BallTracker<0>`. The parameters from adjectives are matched with their corresponding parameters in the nouns and written in short form as in `BallTracker<0>`. The word "Carefully" has a meaning of the object avoidance filter named `AvoidObjects`.

2.4 Connecting the Meanings

The key to constructing the controller based on a short description is the components of the system. They provide the knowledge of the system in executable form, which can be connected in many ways. The system described here does not allow cyclic connections, but components are allowed to have internal state and could easily store information for later use. More complex components can also be constructed out of other components, using a simple extensible language. Each component is dynamically created at run-time by specifying the component's name. One of the components of the system used in the above example is named `Follow`, shown in Figure 2.4. It has 4 inputs, 3 parameters, and 2 outputs. The inputs on the left of the module represent the direction and distance to some object that should be followed. The outputs on the right are assumed to be motor commands. The top lines are parameters, controlling values for rotational gain, following distance, and translation speed. A more complete description of the `Follow` component is in § D.1.

Once each of the words' meanings has been established, the controller must be 'wired' together. The example here has only three components, but other controllers may have more. The system instantiates each object and looks at the number of inputs and outputs.

Nouns in the system have no inputs and four outputs. The outputs have standard meanings which describe the position of the object in the environment. In the example, `BallTracker<0>` doesn't require any connections since it has no inputs[2], but the brackets at the end specify its parameter settings. `BallTracker` has one parameter which refers to the color of the ball being tracked. In this case, the parameter will be set to zero.

[2]Technically, `BallTracker` has the camera mounted on the robot as input, but this input is not represented in the system since the input is not the output of a component. Simillarly, the `Motor` component is said to have no outputs.

Verbs in the system take a multiple of four inputs. Verb components which have no inputs, such as `Spin` or `Stay`, do not require any nouns. Verbs with four or eight components require one or two nouns, respectively. In the example, `Follow` requires one noun, and hence has four inputs. Therefore, the system connects the four outputs from the `BallTracker<0>` component to the four inputs of the `Follow` component. Verbs also have a number of outputs which correspond to the actions of the mobile robot. Most verbs have two outputs, the first for translation and the second for rotation. Others may have three outputs, to control the pan–tilt unit and the zoom of the camera, or they may have five outputs, to control both devices. In this example, `Follow` has two outputs, and so should control the motors of the robot.

Since the task description has an adverb, its corresponding component, `AvoidObjects`, is connected as a filter on the motor outputs. The `AvoidObjects` component has two inputs and two outputs, which passes the `Follow` component's motor commands through unless an object is obstructing the path. Once all three of the components are wired together, the outputs of the verb or verb/adverb pair are connected to the components which control the robots motors. In this example, a `Motor` component is created and connected to the `AvoidObjects` component outputs.

The completed controller is shown in Figure 2.3. The *Color* parameter gets a value of zero, while the other parameters (from the `Follow` component) are not specified and so get the default value. For `Follow`, these values happen to be $Speed = .2$, $P = .01$, $Dist = .5$. Since all of the components are assumed to be in SI units, the speed of the robot when using this controller is $.2\,ms^{-1}$, the rotational gain is $.01\,s^{-1}$, and the following distance is $.5\,m$. Additional examples of complete controllers constructed from task descriptions are included in Appendix E.

2.5 Execution

Once the controller has been constructed, it can then be prepared for execution on the mobile robot. The components have all been instantiated and connected to one another. To ensure that all inputs have been computed before they are needed, the system topologically sorts the components. Since the controller will always be acyclic, the sort will succeed. Each component can then be executed in turn and all inputs will be current. Since each component may need some initialization, the system gives each component the following life cycle:

1. Init Hardware (optional)
2. Initialize
3. Execute
4. Shutdown
5. Release Hardware (optional)

The optional parts of the life cycle are for those components which use part of the hardware which makes up the robot. It is necessary to distinguish between hardware components such as `BallTracker` or `Motor` and computation components such as `Follow` and `Multiply`. Hardware components must share the hardware devices which make up the robot, and hence need additional

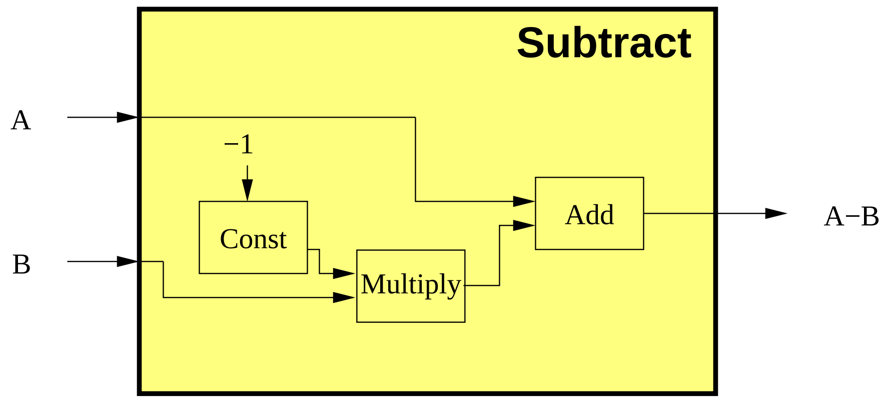

Figure 2.5: A computation component made from other components.

initialization and shutdown. These additional steps are also the reason that only computation components can be used to construct other components.

2.6 Components

The control programs used as a starting point in this work are composed of components. They are designed to be flexible. Each built-in component has a certain number of inputs, outputs, and parameters. For example, a component to perform an addition operation would have two inputs and one output. Parameters are numbers which are constant during any given run of the controller, and can be tuned between runs by advice given from the user. The inputs can be connected to any output in the controller, and outputs may have any number of inputs connected to them. Appendix A contains a list of the working components used for the system presented here.

Each component is a collection of executable code along with all the semantic knowledge about the word or words it represents. For added generality, the parameters can be tuned, and are semantically linked to keywords built-in to the component. Since the controller is meant to be run on a mobile robot, some of the components interface with the hardware devices on the robot. Components which use the robot's hardware are called *hardware components*. All other components are called *computation components*. These computation components compute a value which only depends on their inputs and parameters, not any sensor available to the robot. A special type of component is the *filter component*. The filter component takes motor commands as inputs and outputs the modified commands. It is important to note that filter components may also be hardware components or computation components, depending on its use of hardware.

Computation components can be nested inside one another to make more complicated computation components. For example, the component `Subtract` might be constructed out of a `Multiply` component and a constant, -1, represented by a `Const<-1>` component[3], with an `Add` component to add the negated input with the unnegated input, as shown in Figure 2.5.

[3]The `Const` component takes one parameter and copies it directly to its only output, thus making a constant.

2.7 Related Work

Since one of the goals of the work here is to provide a natural way to express a control program, it is convenient to have a system which understands some natural language commands. In the work here, only a simple LALR(1) parser is used. More advanced language processing is outside the scope of the work here due to numerous pitfalls in language processing [4] which would distract from the overall goal.

The components here are similar to elements of Visual Programming. Burnett, Goldberg and Lewis's book [1] is a good reference on programming such visual languages in an object–oriented way. Visual Programming is a method of writing programs from graphical or picture–based representations of the programming language's grammar. Some visual programming languages use data flow diagrams, with "wires" that connect the components and show the component's relationships. The work here involves components which directly relate to these techniques. Each of the components has a number of inputs, outputs, and parameters which can easily be represented by graphical icons and "wires" linking them together, as shown in Figures 2.3(controller) and 2.4.

This chapter handled connecting the inputs and outputs of the components. The parameters can get values from the task description, but most parameters get a default value. What if that value is not correct? The next chapter discusses how to use the system to tune the behavior of the robot after giving it a task description.

Chapter 3

Tuning

Each of the component parameters can be tuned in the controller to enhance performance. Each component provides its own key words which are used to tune the parameter. For example, the proportional controller implemented by the `P` component uses the words “More Aggressively” and “Less Aggressively” to allow the trainer to tune the gain. The tuning words are separated into categories which correspond to the robots actions, such as “Turn” for rotation. The category is chosen based on the connections in the controller, so a P-controller connected to the rotational input of the `Motor` component is in the “Turn” category. Other components may create their own category to add tuning words.

In the example, the robot is instructed to follow a red ball. If the ball is recognized by the vision system, it will follow the ball by turning towards it and driving forward. If the ball is particularly fast in moving side to side and the camera loses sight of the ball frequently, the trainer can use the tuning words “Turn More Aggressively” to make the robot keep up with the ball. Likewise if the ball is moving too quickly away from the robot, the tuning words “More Speed” would be helpful.

3.1 Semantic vs. Literal

One might argue that the tuning keywords should be more literal, since a natural language description might be misleading. It is possible to use the words “Double the gain on the P-controller connected to the rotational motor output” which is the correct description of what “Turn More Aggressively” does. The main issue is that the focus for the work presented here is to allow people who may not know what a P-controller is to use the system. For them, the choice of describing the behavioral change to the system, i.e. “Turn More Aggressively”, is more appropriate.

3.2 Adjustment Functions

These tuning words provide a behavioral description of the changes a parameter change produces. This makes the system easier to use for trainers that do not know the lower level details of the robot’s controller. One problem is that a specific set of words is not usually as expressive as typing in a new “magic number”. A pair of words cannot easily navigate the space of real numbers. This is an open problem, and one which needs to be addressed. To that end, a formal description of the problem is presented here.

Given a real-valued variable x, and an unknown optimal value x^*, define a finite family of functions $f_i : R \rightarrow R$, such that a sequence of function applications from this family map the initial value x_0 to the optimal value x^*. For example, if $x_0 = 1$ and $x^* = 5$ when the available functions are $f_1(x) = 2x$ and $f_2(x) = x + 1$, then one solution would be $\{f_1, f_1, f_2\}$ or

$$f_2(f_1(f_1(1))) = 2(2(1)) + 1 = 5 = x^*.$$

If x^* had not been an integer of the same sign as x_0, these functions would not be able to reach the optimal value. A small set of functions which are describable in natural language, such as *a little more*, that provide a way to approach the optimal value in as few function applications as possible would be the best solution for this type of system. The functions should have a reachable set which mostly covers the set of real numbers, but pays particular attention to the region around zero, and the values with just a few decimal places.

The system here uses the functions $f_1(x) = 1.1x$ and $f_2(x) = 0.9x$ to provide tuning. Another method might use a binary search approach, using the minimum and maximum of previous values which were too high or too low. The functions would be

$$f_H(x) = \frac{\max L + \min H \cup \{x\}}{2}$$

and

$$f_L(x) = \frac{\max L \cup \{x\} + \min H}{2}$$

where H and L are sets of previous responses which were designated too high or too low, respectively.

The difficulty in deriving good families of functions is that it is not enough to have the set of real numbers in the reachable set of all combinations of compositions of the f_i’s, but we want to be ϵ-close in n steps, where n is a small number. For example, for binary search, given ϵ and the range of possible values $[0, h]$, we can compute n as

$$n = \log_2 \frac{h}{\epsilon}.$$

In addition, it may not be necessary to cover the set of real numbers, but a reasonable subset which is meaningful for a given parameter. The limits on the parameter might be based on hardware limitations such as motor speed or torque limits.

3.3 Related Work

The tuning presented here is really a form of advice. While the robot is moving, a person telling it “Turn More Aggressively” is giving it additional information to try to increase its performance. The idea of agents incorporating advice has been around for a long time [8]. McCarthy suggests the use of advice for systems which use logical inference and theorem proving. McCarthy's advice consists of sentences in a language the system understands, in his case, predicate logic. Some researchers attempt to use advice directly by modifying the structure of an Artificial Neural Network (ANN) [9]. Some work modifies the ANN by setting the initial weights prior to learning to large positive values instead of small random ones [10]. This method does improve learning rates, but requires specific knowledge about the environment and the function of the ANN being trained. These techniques seem more like seeding initial knowledge rather than giving advice in McCarthy's sense. Later methods [6, 7] translate the advice into nodes of the ANN to allow the network to use the advice if it helps achieve the goal, or ignore it otherwise. The advice is input as a boolean program, and hence requires a user to know the syntax of the language and the symbolic entities which the agent understands. The advice can thereafter be refined or become completely unrecognizable after continued learning.

Without having direct knowledge of how to incorporate the advice received into the final controller, advice can be difficult to use effectively [5]. The advice may be decoded incorrectly, leading to the wrong behavior. Also, the advice may be correctly interpreted, but the desired behavior may not work as intended due to randomness in the environment.

Riley, Veloso, and Kaminka [12] advise their soccer–playing robots with a special advice language. The limitations of the language seem to keep the system tied to very specific states, rather than allowing for generalization. This limitation seems to contribute to the "coaching problem" the authors describe.

Clouse and Utgoff [2, 3] use automated agents which give the advice to the learning agent. The training agent seems to be little more than a hand–coded control program which is tuned to be able to reach the goal state from any random state. The system also has a parameter to select the amount of advice to use, which equates to performing some random actions to explore the space. This seems to be a predecessor to Smart's work [14, 15] which takes input from a hand–coded procedure or a human with a joystick. Smart's JAQL framework bootstraps learning using this external control which shows "interesting" parts of the state space [13]. Since a large part of the learning time in Q–Learning is in exploring the space to find the sparse rewards [16], this method allows the agent to see rewards almost immediately.

The parameters of the controllers here can be adjusted from natural language advice words built into the system. Since the system takes "advice" at run–time and can be used as an "automated" training agent, this work is similar to a combination of two of the above methods.

Given the tuning methods from this chapter, and assuming that the optimal value can be found for each parameter, it may still be possible that the robot does not perform the task as it was intended to. The next chapter discusses the possible remedies for the shortcomings of the system presented so far.

Chapter 4

Discussion

The system so far leverages previous knowledge about the tasks the robot will be required to do, and uses that to construct sequences of executable code that attempt to accomplish the task. There are, however, limitations to the system as it stands.

4.1 Closed World

The system puts the robot under the closed world assumption[11]. The system has a fixed set of objects and behaviors that describe how the world works. Since the robot has a limited skill set, the closed world assumption is acceptable for the types of tasks this system can perform.

4.2 Limited Objects

The number of words in use by the system (Appendix B) is fairly small. The main problem is that there are only a few types of objects that the system can recognize. If a robot using this system were to be deployed in an area with arrow signs telling it which way to go, the sentence "Follow the arrows" doesn't mean anything to the robot since the word "arrows" doesn't have a meaning.

The solution is to add a component to the system, say `ArrowTracker`, and assign the words "arrow" and "arrows" the meaning of "ArrowTracker". Making the `ArrowTracker` component would take a programmer, since it interfaces with the robot's hardware, seemingly defeating the purpose of the system presented here. The goal of the system, however, is to reduce the need for robot programmers *after* the robot has been deployed.

Recognizing objects from a camera view or laser scan is difficult. Writing robust recognizer code generally takes a specialist programmer anyway, so until a universal recognizer exists, robot deployments will benefit from special purpose code for recognizing objects. This system simply decouples the effort from that of the behavior of the robot, so that each can be reused and combined in different ways.

Thus, the answer to the limited objects problem is simply to add more built-in program code, which is outside the scope of this project. The set of recognizers in the system were chosen as a simple set which could be used to demonstrate the capabilities of the system without spending a

```
# Subtracts the second input from the first
Input: A,B
Output: Difference

Negate (B)    => nB
Add    (A,nB) => Difference
```

Figure 4.1: The text needed to create the `Sub` component.

disproportionate amount of time on the recognizer code. The focus of the work is, after all, on the behavior of the system.

4.3 Limited Behaviors

The problem remains of how to expand the system to incorporate new behaviors. First, new behaviors may always be programmed into the system directly. Interestingly enough, there are no behaviors in the system presented here which were entered by direct programming. The second way to add behaviors is by using the simple extensible language which is built into the system. The final way is to *transfer* the knowledge from the controller into a learning agent, and then let it learn the proper behavior.

Programming behaviors for mobile robots often involves a lot of the same techniques over and over again. These techniques, such as proportional controllers, have been programmed into the system as components, such as `P`, which implements a P-controller. Other primitives, such as addition, subtraction, if-then-else, and range checking allow many behaviors to be built without the need for further direct programming.

The language built into the system relies on the names of the components. Since each built-in component can be created by its name, a new component can be made from a text file which is mainly a list of the components needed and the connections between them. An example file to create a `Sub` component is shown in Figure 4.1. The header lines, except for the comment, define the inputs and outputs of the component. Parameters can be defined using the `Parameters:` header line. Another header line, `Defaults:`, specifies the default parameter values for each parameter of the component. After the header lines, each line starts with the name of a component, followed by a list of symbols in parentheses. Each symbol represents a connection. Note that the component `Negate` takes one input, `B`, which is also one of the inputs to the `Sub` component. The symbols, like `B`, act as wires connecting inputs to outputs. After the input list a `=>` appears, followed by a list of symbols which define its outputs, like `nB`. The symbol `nB` can now be used as a symbol representing the output of the `Negate` component. The last line uses the input symbol `A` and the new `nB` symbol as inputs to `Add`, and connects its output to the output for the `Sub` component. This language is extensible, because each new component created can be used to make others. For example, the `Negate` used by the `Sub` was actually loaded from a different file, whose text is listed in Figure 4.2. Note also that parameters can be given values by placing them in angle brackets next to the component name. For more complex examples and examples of behaviors, see § D.1.

```
#Negation Component
#
# Takes one input x and returns -x
#
Input: X
Output: Minus_X

P<-1>(X) => Minus_X
```

Figure 4.2: The text needed to create the `Negate` component.

Learning can be used as an extension to this system. Since learning has been shown to greatly benefit from provided experience [13, 16], agents which started from the knowledge in the generated controllers made by this system will significantly outperform standard learning techniques over the same state space. The next chapter shows how the performance of this system compares to direct programming.

Chapter 5

Experimental Results

To relate the performance of the system presented here to other established work, two sample tasks were constructed. These trial tasks represent some common robot applications involving sensor/motor interaction.

5.1 The Two Trials

The two tasks used in this comparison are object tracking and corridor following. Both tasks involve obstacle avoidance. Each task is described in detail in the following sections. Two task-specific evaluation metrics are used to rank the competitors.

5.1.1 Keeping it Seen (KIS)

This task measures a robot's ability to keep track of a specific object and ensure it does not get away. Since the area has obstacles, it is necessary to move the robot in order to view the object. This task uses the laser range finder and blobfinder to drive the motors. Those are the only devices available (e.g., the full camera image is not available).

Environment. A round blue object, initially visible to the robot moves to the opposite corner of the environment at a constant speed. The object stops near the corner of the environment.

Goal. The goal is to keep the blue object visible to the robot as much as possible, and stop at a predetermined distance from the blue object.

Evaluation Metric. This task is scored in two parts. First, every 0.1 seconds the robot receives:

- -0.1 points if the object is not visible to the robot's camera
- -1 point if the robot is in contact with an obstacle or wall

Second, after the ball stops and the robot has stopped moving for 5 seconds, the robot receives the following:

- +50 points if the robot is facing the blue object within 3 degrees
- +25 points if the robot is facing the object between 3 and 10 degrees
- -1 point for each centimeter above or below one meter from the object in the direction of the robot

Variations. To examine the robustness of the candidates, the task is scored on three variations. The difference is in the speed of the ball's movement. The trials are run using $0.1m/s$, $0.2m/s$, and $0.3m/s$, in that order. The test environment uses the slowest speed and the obstacles are not in their official test positions. The obstacles are placed in one corner so the user can move them around to test their controller.

5.1.2 Corridor Traversal (CT)

Many jobs require getting from one place to another. In this task the robot is to traverse a mostly uncluttered corridor and reach a point near the center end of the corridor. This task uses only the laser range finder to drive the motors.

Environment. The robot starts off the center line of the corridor, facing in the general direction of the end to be reached, but not exactly. The angle is no more than 60 degrees from the center line.

Goal. There are two goals for this task. The first is to reach the area near end of the corridor. The second is to be as close as possible to the center line of the corridor while driving to the location.

Evaluation Metric. The scoring for this task is also in two parts. First, every 0.1 seconds the robot receives:

- -1 point per meter from the center line of the corridor
- -1 point if the robot is in contact with an obstacle or wall

Second, after the robot has stopped moving for 5 seconds, the robot receives the following:

- +50 points if the robot is within 10 cm of the center line of the corridor
- +25 points if the robot is between 10 cm and 30 cm from the center line
- -1 point for each centimeter above or below one meter from the end of the corridor along the center line

Variations. The candidates in this task must show that they can navigate successively longer corridors. The corridors are 5, 10, and 15 meters in length. Each has several rectangular obstacles placed along the wall. The robots starting position is also varied so the angle and distance from the center are not constant for each corridor. The test environment uses the shortest corridor and the obstacles are not in their official test positions. The obstacles are placed in the middle so the user can move them around to test their controller. Also, since there are more opportunities for negative points to accumulate in longer corridors, the bonus points at the end of the corridor are multiplied by the trial number (i.e., the large bonus is +100 for the second corridor, +150 for the third).

5.2 The Competitors

This section briefly describes how each of the controllers compared here were derived. Once each was constructed based on the test environment, the same controller was tested on each of the three variations of the two trial tasks.

Handwritten Code. The direct comparison of this projectis to handwritten code. Once the rules and metrics were chosen, the author wrote a simple program for each task and, using a test environment, debugged and tuned the behavior. The time to accomplish this development cycle was recorded. The time spent on the Keeping it Seen task was 72 minutes, while Corridor Traversal took 108 minutes.

Assembled. The system presented here was used to construct a controller for the robot using a single English sentence. For the Keeping it Seen task, the sentence was “Carefully follow the blue ball.” The Corridor Traversal task used “Carefully follow the corridor.” All values affecting the operation of the controller were left at their default values.

Assembled and Tuned. Using the test environments, the controllers were tuned using commands like “Turn More Aggressively” and “Drive Faster” to change the parameter values of the controllers. Once the author was satisfied that the controller was performing well, the controller was saved and used for testing.

5.3 Results

The bottom line for this project is its comparison to direct coding. Based on the limited study the system seems to provide a similar level of performance compared to the more time–intensive programmed controllers, although there were some cases were the programmed solution achieved slightly better results. The system–assembled controllers, however, seem more robust to changing conditions even without re–tuning, which would produce further gains in performance.

Keeping it Seen. The graph in Figure 5.1 displays the average score of the three controllers over 10 runs. The three separate variations of this trial are labeled KIS 1, KIS 2, and KIS 3. The controllers were programmed or tuned on the first trials speed. The second and third trials used the same controllers, to indicate the robustness of the solutions. The negative score for the programmed variation was a result of 6 of the 10 runs failing to maneuver around an obstacle, which was scored at -100, the minimum score. The obstacle avoidance code in the programmed solution was not as well polished as that built into the assembled controller. Aside from that, the performance of the assembled and tuned controller is on par with that of programming, although with the system presented here it took much less time to create the solution.

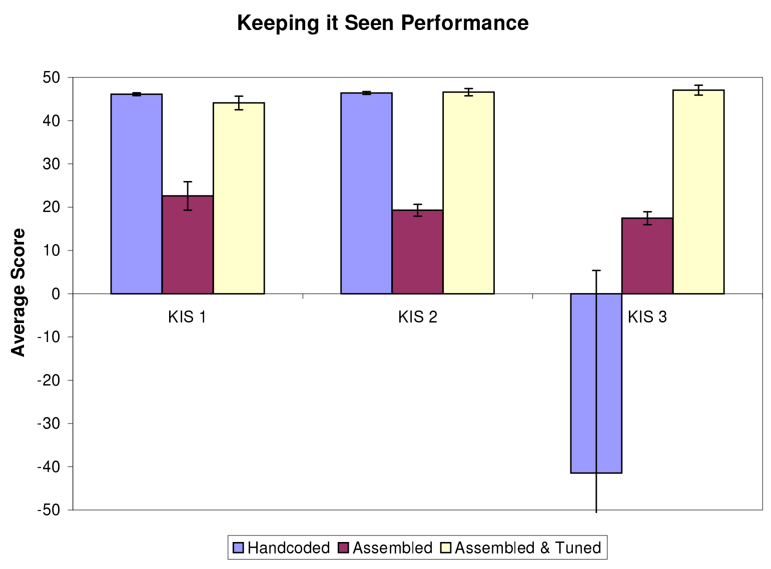

Figure 5.1: Performance on the Keeping it Seen task.

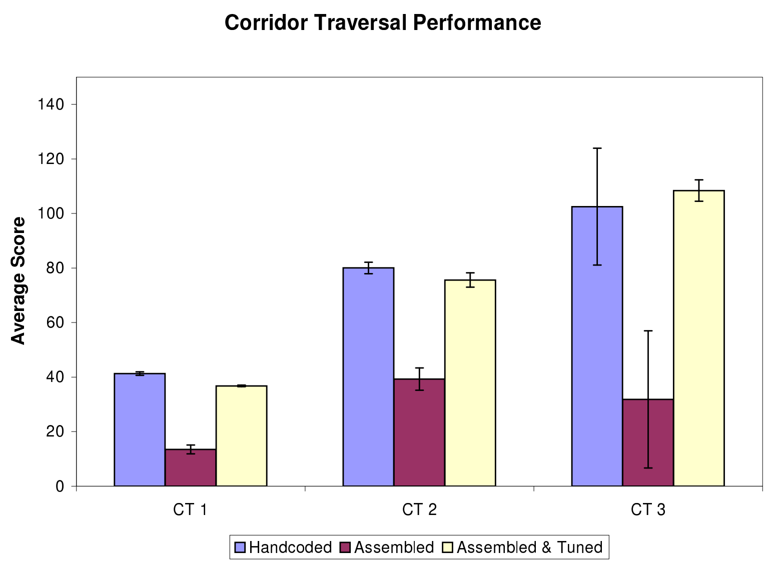

Figure 5.2: Performance on the Corridor Traversal task.

Corridor Traversal. Figure 5.2 illustrates that the programmed controller achieved significantly better results in the first two variations of the trial. This was because the programmed controller drove the robot backwards if it was facing far enough away from the center line, while the assembled controller only drove forward. This kept the programmed controller from losing some points in the beginning, when the robot starts far off the center line. The programmed controller did have problems with the longer corridors, since the obstacles on the wall tended to obscure the features it was looking for in the laser range data.

The assembled and tuned controller had mixed reviews compared to direct programming. It performed better in some cases and worse in others. The places with lower performance, however, are more than outweighed by the increased robustness that comes from modular, polished code, and a reduced time-to-solution for the tasks. Moreover, if an assembled solution were not performing at the desired level of performance, and a programmer must be called in, the programming time could be significantly reduced by providing the data from the assembled controller's trials. An additional benefit may be obtained from parts of the source code that makes up the controller, since the programmer may be able to determine which specific part of the controller failed and replace just that component.

Appendix A

List of Working Components

The components below are all working and usable in the current system. The left column contains all the built-in modules which are used to construct the added modules in the second column. The rightmost column contains modules which directly use hardware connected to the robot, and hence are less generic and not allowed to be used as components of other modules.

Built-In Modules

```
Add
Alias
Constant
P
I
D
FlipFlop
Multiply
If
InRange
Smooth
Smoother
SpeedFactor
Thresh
```

Added Verb Modules

```
Avoid
Follow
Spin
Stay
```

Other Added Modules

```
Average
Dist2Error
KeepEqual
Negate
PD
PID
Select22
Sub
```

Hardware Modules

```
AvoidObjects
BallTracker
CorridorTracker
HereTracker
LaserRegions
Motor
PersonTracker
SafeMotor
```

Appendix B

List of Words in Dictionary

Adjectives

RED
GREEN
BLUE
LONG
SHORT
MY
YOUR
GREATER
MORE
LESS
SMALLER
ALIKE
SAME
EQUAL
DIFFERENT
X
Y
DISTANCE
FOUND
LEFT
RIGHT

Adverbs

QUICKLY
SLOWLY
DOWN
UP
SMOOTHLY
CAREFULLY

Verbs

COME
GET
GO
DRIVE
KEEP
KEEPING
TRY
TRYING
DRIVING
MOVING
TURNING
LOOKING
PANNING
TILTING
ZOOMING
AVOIDING
FOLLOW
AVOID
SPIN
STAY

Articles

A
AN
THE

Nouns

BALL
ME
I
CORRIDOR
HERE
INPUT
OBJECTS
WORD
WALL

Prepositions

WITH
IN
FOR
FROM
AT
TO
ON
NEAR
BY
ABOVE
BELOW
THAN
OF

Conjunctions

AND
BUT
OR
WHILE
BEFORE

Number Words

ZERO
OH
ONE
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
NINE
TEN
ELEVEN
TWELEVE
THIRTEEN
FOURTEEN
FIFTEEN
SIXTEEN
SEVENTEEN
EIGHTEEN
NINTEEN
TWENTY
THIRTY
FORTY
FIFTY
SIXTY
SEVENTY
EIGHTY
NINETY
BILLION
MILLION
THOUSAND
HUNDRED
TENTH
TENTHS
HUNDREDTH
HUNDREDTHS
THOUSANDTH
THOUSANDTHS
MILLIONTH
MILLIONTHS
MINUS
NEGATIVE
POINT
DOT

Appendix C

Grammar of Natural Language Input

Terminal Symbols:

VERB, NOUN, ADVERB	Word of the specified part of speech
ADJECTIVE, ARTICLE	
CONJUNCTION, PREPOSITION	
END_OF_SENTENCE	End of sentence marker: . ? ! etc.
UNKNOWN	A word not recognized by the dictionary
NUM_WORD	A number word, like 'two' or 'hundred'
NUMBER	A literal number: 345
KEEP, TRY, INPUT, WORD	Special words which represent themselves
Λ	Null string (lambda)

```
        root  →  Λ
              |  root any_sentence END_OF_SENTENCE
              |  root error END_OF_SENTENCE
              |  root error UNKNOWN error END_OF_SENTENCE

any_sentence  →  Λ
              |  sentence CONJUNCTION sentence
              |  sentence
              |  special_command
              |  special_command CONJUNCTION special_command
              |  sentence CONJUNCTION special_command
              |  definition

    sentence  →  predicate
              |  subject predicate

     subject  →  descr_noun

   predicate  →  descr_verb
```

```
     descr_verb  →  VERB
                  | VERB object
                  | VERB ADVERB
                  | ADVERB VERB
                  | VERB ADVERB object
                  | ADVERB VERB object
                  | VERB object ADVERB

         object  →  descr_noun
                  | descr_noun prep
                  | prep

     descr_noun  →  NOUN
                  | adjectives NOUN
                  | ARTICLE NOUN
                  | ARTICLE adjectives NOUN

     adjectives  →  ADJECTIVE
                  | adjectives ADJECTIVE

           prep  →  PREPOSITION descr_noun

special_command  →  KEEP keep_clause
                  | TRY try_clause

    keep_clause  →  sensor relate_value by_clause
                  | sensor CONJUNCTION sensor comp_clause by_clause
                  | VERB relate_value

   relate_value  →  PREPOSITION value
                  | ADJECTIVE PREPOSITION value

      by_clause  →  Λ
                  | PREPOSITION VERB

    comp_clause  →  ARTICLE ADJECTIVE
                  | ADJECTIVE

         sensor  →  descr_noun ADJECTIVE INPUT

          value  →  NUMBER
                  | num_word
                  | ARTICLE NUM_WORD

       num_word  →  NUM_WORD
                  | num_word NUM_WORD

     try_clause  →  Λ

     definition  →  ARTICLE WORD UNKNOWN
```

Appendix D

Word Meanings in Dictionary

Adjectives

RED	%Color=0
GREEN	%Color=1
BLUE	%Color=2
LONG	%Length=1.5
SHORT	%Length=.5
MY	
YOUR	
GREATER	KeepGreater
MORE	KeepGreater
LESS	KeepLess
SMALLER	KeepLess
ALIKE	KeepEqual
SAME	KeepEqual
EQUAL	KeepEqual
DIFFERENT	KeepNotEqual
X	(0)
Y	(1)
Distance	(2)
Found	(3)
Left	(2)
Right	(1)

Articles

A
AN
THE

Nouns

BALL	BallTracker
ME	PersonTracker
I	PersonTracker
CORRIDOR	CorridorTracker
HERE	HereTracker
INPUT	:268
OBJECTS	
WORD	:271
WALL	LaserRegions

Verbs

COME	Follow<,,0.0>
GET	Follow<,,0.2>
GO	Follow<,,0.3>
DRIVE	Follow<,,0.0>
KEEP	:266
KEEPING	:266
TRY	:267
TRYING	:267
DRIVING	Motor(0)
MOVING	Motor(0)
TURNING	Motor(1)
LOOKING	PTZ(0)
PANNING	PTZ(0)
TILTING	PTZ(1)
ZOOMING	PTZ(2)
AVOIDING	AvoidObjects
FOLLOW	*Loaded from file*
AVOID	*Loaded from file*
SPIN	*Loaded from file*
STAY	*Loaded from file*

Adverbs

QUICKLY	SpeedFactor<1.5>#
SLOWLY	SpeedFactor<.5>#
DOWN	
UP	
SMOOTHLY	Smoother<.9>#
CAREFULLY	AvoidObjects#

Prepositions

WITH	
IN	
FOR	
FROM	
AT	KeepEqual
TO	KeepEqual
ON	KeepEqual
NEAR	KeepEqual
BY	KeepEqual
ABOVE	KeepGreater
BELOW	KeepLess
THAN	
OF	

Conjunctions

AND
BUT
OR
WHILE
BEFORE

Number Words

ZERO	0
OH	0
ONE	1
TWO	2
THREE	3
FOUR	4
FIVE	5
SIX	6
SEVEN	7
EIGHT	8
NINE	9
TEN	10
ELEVEN	11
TWELVE	12
THIRTEEN	13
FOURTEEN	14
FIFTEEN	15
SIXTEEN	16
SEVENTEEN	17
EIGHTEEN	18
NINETEEN	19
TWENTY	20
THIRTY	30
FORTY	40
FIFTY	50
SIXTY	60
SEVENTY	70
EIGHTY	80
NINETY	90
BILLION	*1000000000
MILLION	*1000000
THOUSAND	*1000
HUNDRED	*100
TENTH	/10
TENTHS	/10
HUNDREDTH	/100
HUNDREDTHS	/100
THOUSANDTH	/1000
THOUSANDTHS	/1000
MILLIONTH	/1000000
MILLIONTHS	/1000000
MINUS	-
NEGATIVE	-
POINT	.
DOT	.

D.1 Verb components

Definition of `Follow`

```
# Follow verb
Input: X,Y,Distance,Found
Output: T,R
Param: %Speed,%Pr,%Dist
Defaults: .2,.01,.5

# Make the Distance input an error signal
Dist2Error<%Dist> (Distance,Found) => Z

InRange<1e-5,1e30>(Z) => CanGo
Const<0>() => Stop
Const<%Speed>() => Go

If (Go,Stop,CanGo) => T
P<%Pr>(X) => R
```

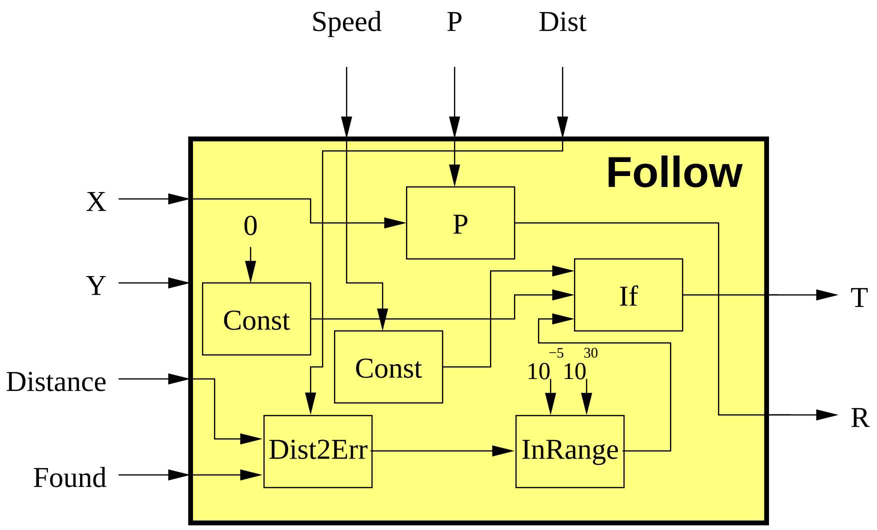

Figure D.1: A diagram of the inner workings of the `Follow` component.

Definition of `Avoid`

```
# Avoid action, mostly "don’t look at"
Input: X,Y,Distance,Found
Output: T,R
Param: %P,%Dist
Default: .01,1

Dist2Error<%Dist>(Distance,Found) => Z
Const<0>() => Zero
Const<-.1>() => Backup

Alias (X)     => C
InRange<-40,40>(C) => InCenter
InRange<-1e30,-1e-5>(Z) => TooClose

Multiply (InCenter,TooClose) => ShouldBackup
If (Backup,Zero,ShouldBackup) => T
Negate (C) => AvoidX
P<%P>(AvoidX) => R
```

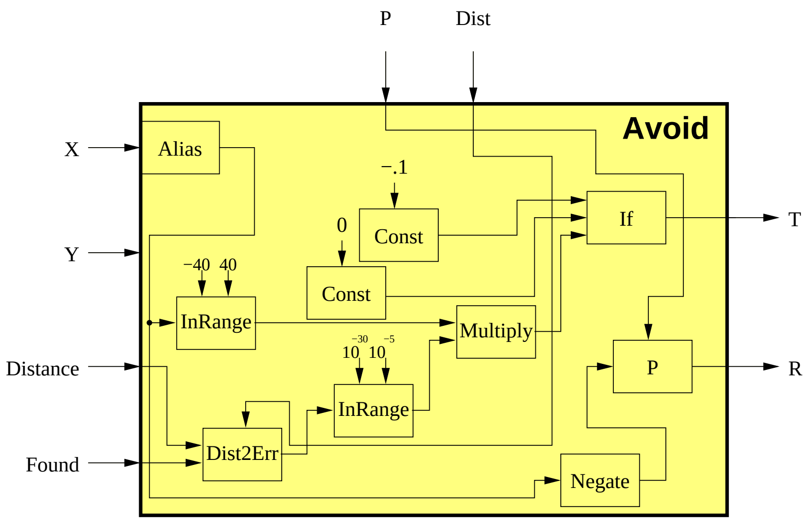

Figure D.2: A diagram of the inner workings of the `Avoid` component.

Definition of `Spin`

```
# Control Signals to spin the robot around
Output: T,R
Param: %Speed
Default: .5

Const<0>      () => T
Const<%Speed> () => R
```

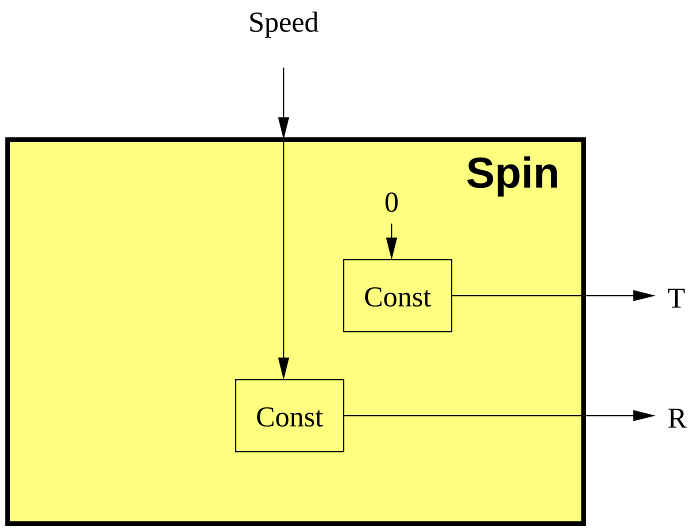

Figure D.3: A diagram of the inner workings of the `Spin` component.

Definition of `Stay`

```
# Robot is completely stopped
Output: T,R

Const<0> () => T
Const<0> () => R
```

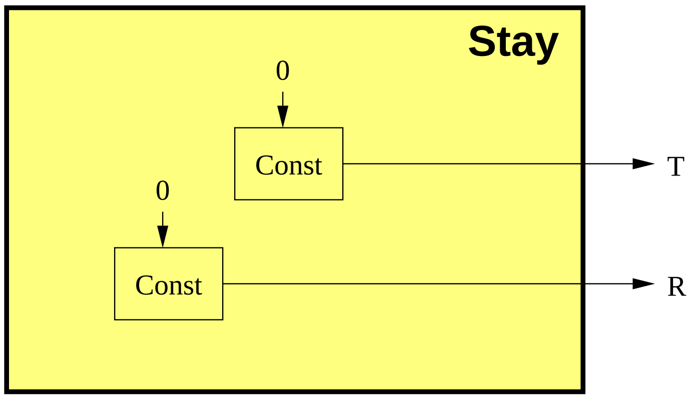

Figure D.4: A diagram of the inner workings of the `Stay` component.

Appendix E

Example Controllers

Five demonstration tasks were chosen and executed using the framework on a real mobile robot. It is difficult to give performance metrics on paper, so each task has a qualitative description in the *Behavior* section.

E.1 "Carefully follow the blue ball"

Setup. A round blue object was moved within view of the robot's camera. The object is slowly moved away from the robot to lead the robot down a corridor. The path of the object is not straight, but rather moves from one side of the corridor to the other randomly to ensure that the robot is actually seeking the blue object. The controller constructed for this task used a P-controller to control the robot's rotation and keep it centered on the blue object.

Behavior. The robot turned toward the blue object continuously and drove toward it. If the blue object happened to go outside of the camera's view, the robot stopped moving. It drove forward at a constant velocity whenever the object was at least a certain distance away. If the object was too far away, however, the robot would stop since it no longer was visible to the `BallTracker`.

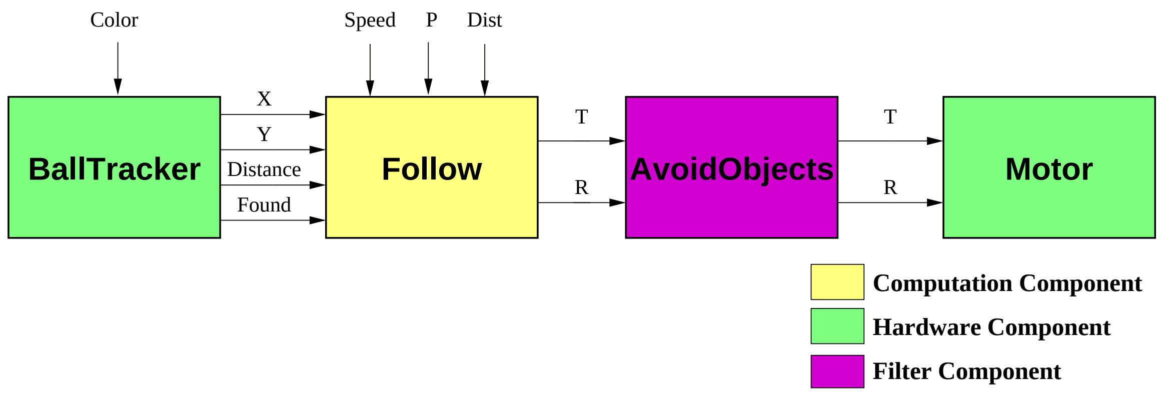

Figure E.1: The controller for the task "Carefully follow the blue ball."

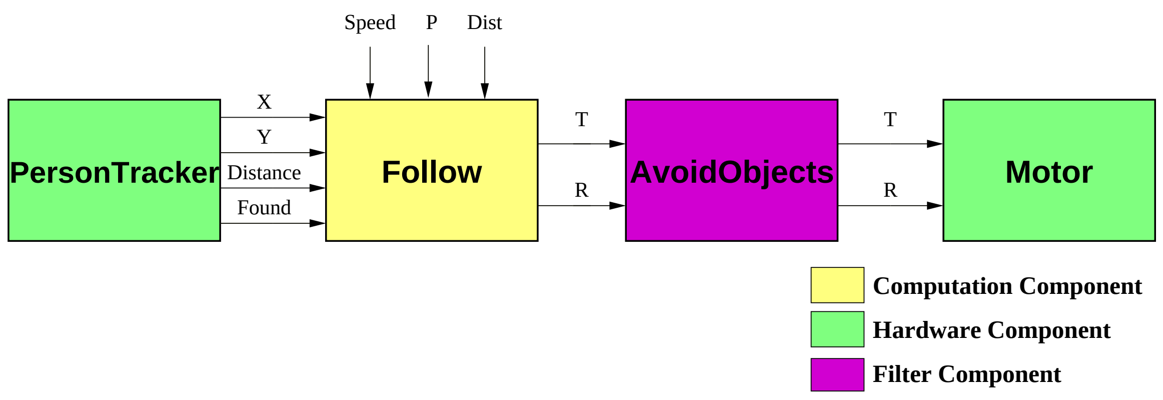

Figure E.2: The controller for the task “Follow me carefully.”

E.2 “Follow me carefully”

Setup. This task involves trajectory following. The word “me” refers to the path of the person walking in front of the robot. The laser range finder is used to watch for movement and record the position of movement relative to the robot. Once enough movement is seen to establish a trajectory, the robot emits a sound telling the person to wait while it catches up. The robot then drives to each point seen by the laser to approximate the trajectory it witnessed. A different sound is emitted to let the user know to continue walking. For evaluation purposes, a line of masking tape was placed on the carpet for the person to walk along. The controller is the same as in the task in section E.1, with a different tracker component. The tracker for this task uses the laser to locate points of movement instead of a colored object in the camera view.

Behavior. The robot imitated the path of the person that walked along the masking tape. The system currently cannot respond to movement from more than one source. It assumes that all movement is from the same source and follows it in the order it was received. This can also include intermittent reflections from a shiny metal chair leg as movement.

E.3 “Drive to here while avoiding objects”

Setup. The “here” referred to in the task description is actually a blue sign of a certain height. The height of the sign is important because it is used to determine the distance to the sign from the camera view. The camera is used exactly once to determine the location of “here” and then the robot attempts to drive to that location. The “here” sign is removed after the robot begins moving, and the location is marked with masking tape for evaluation purposes. The movement is controlled by a follow mechanism as in the task of section E.1, but tries to get closer to the target because the word “Drive” was used instead of “Follow” as in previous trials. Appendix D shows the meaning of the word “Drive” is similar to “Follow” but changes the distance parameter.

Behavior. The robot drove to the marked location and stopped when it got within a certain distance. A few obstacles were placed in the robot’s path, and it still reached the goal. When

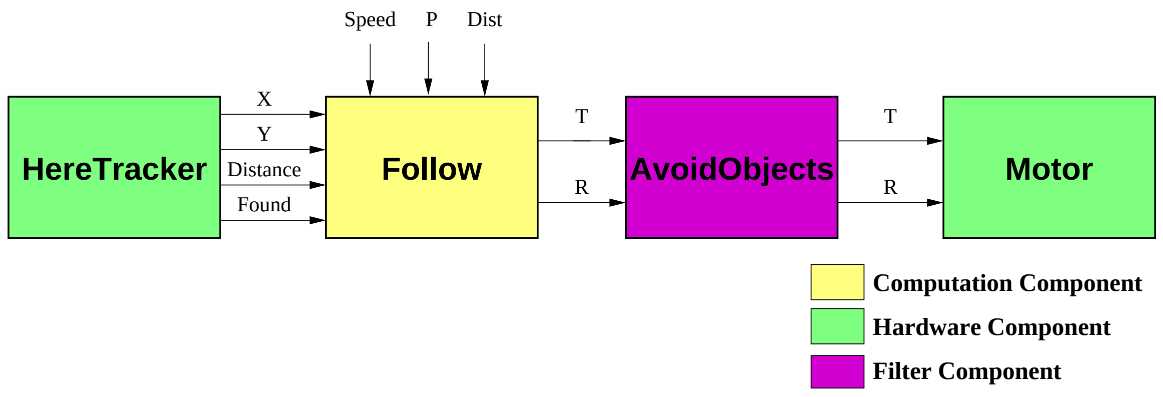

Figure E.3: The controller for the task "Drive to here while avoiding objects."

objects completely blocked the path to the goal, the robot wandered back and forth along the obstacles. If an opening appeared, it would then proceed to the goal. Certain types of obstacles would prevent the robot from reaching the goal, such as corners where each side of the corner was longer than a meter or two, and the corner was pointed roughly in the direction of the goal. This type of corner would cause the robot to turn around, leave the corner, and turn back toward the goal, which placed it back in the corner again.

E.4 "Avoid the blue ball and come here carefully"

Setup. This task involves the round blue object again, but this time the robot is told to avoid the object. Its secondary goal is to get to the location "here", as specified in section E.3. The controller consists of two parts, one of which is the "drive to here" controller, and the other is the `Avoid` component which has a negated P-controller to look away from the blue object, and a test to see if the object is centered which drives the robot backward slowly.

Behavior. The robot attempted to drive to "here", but whenever the blue object was visible, it stopped and looked away from the object. If the object persisted in the center region of the camera, then the robot backed up slowly. The avoid behavior was active whenever the blue object was visible, otherwise the "drive to here" behavior is active. If the blue object was held at certain points between the robot and the "here" location, the robot never was able to reach the goal. By using the tuning words "Drive Smoothly", the behaviors blended together and the robot was able to continue moving forward long enough to drive around the blue object.

E.5 "Keep the wall left input at one point three and keep driving at point one"

Setup. The task description of this task uses the second form (see section 2.1), to specify a relationship to be enforced between values. This low–level method of specifying tasks can be useful for constructing other behaviors. The given task allows the robot to follow a wall at a given distance.

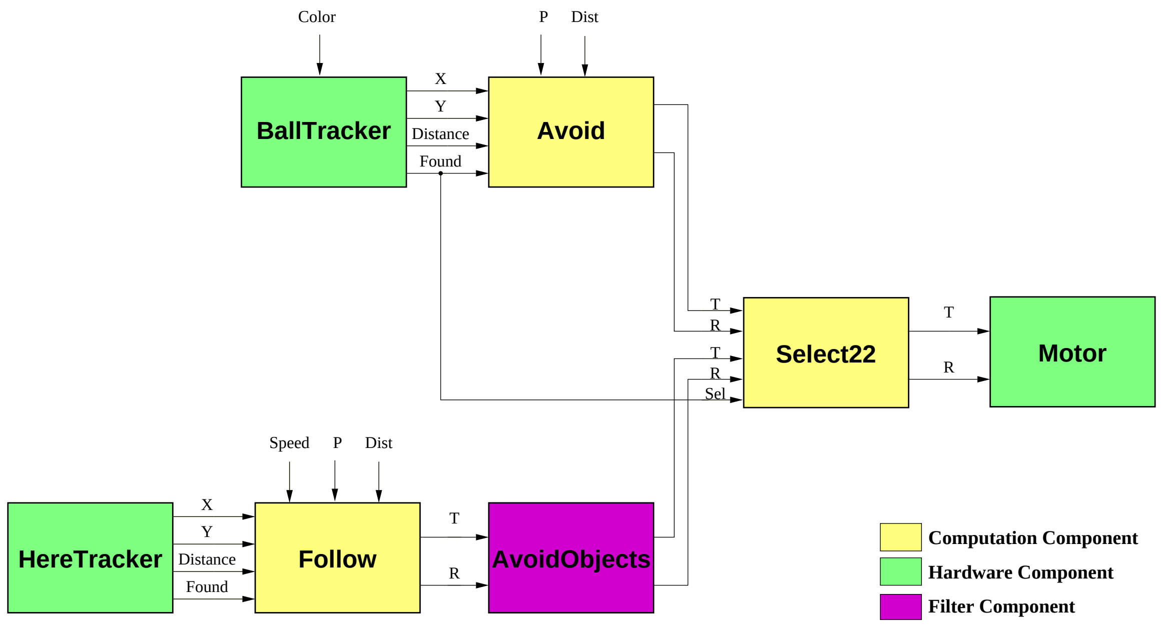

Figure E.4: The controller for the task "Avoid the blue ball and come here carefully."

Masking tape was placed on the floor parallel to the wall at the intended distance for evaluation purposes.

Behavior. The robot drove along the wall approximately where the masking tape was placed. When a corner was encountered, the robot curved around the corner, overshot the tape line and then returned to it and continued down the wall. This behavior was expected because of the P-controller used to keep the values equal. The robot continuously drove at a velocity of $.1\,ms^{-1}$, as specified in the task description. The task description does not include any obstacle avoidance, so any objects not touching the wall were hit. Also the gain of the P-controller used was very important in deciding how quickly the wall could change and still have the robot follow it.

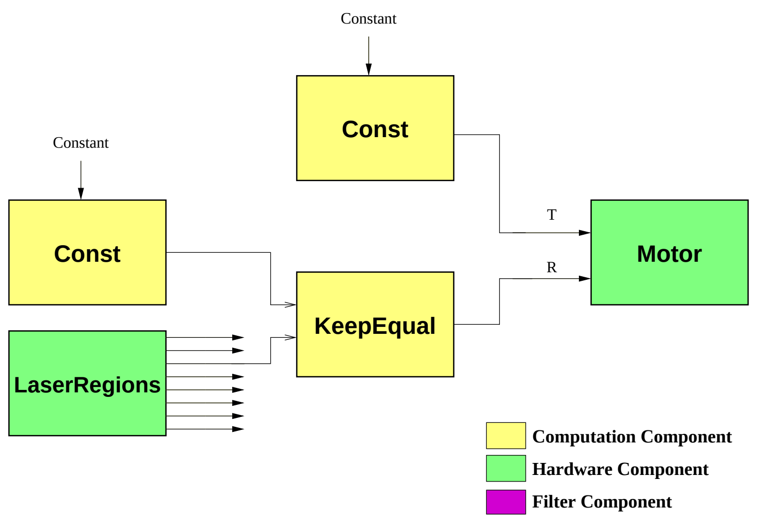

Figure E.5: The controller for the task "Keep the wall left input at one point three and keep driving at point one."

References

[1] Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors. *Visual Object-Oriented Programming: Concepts and Environments.* Prentice-Hall/Manning, 1995.

[2] J. A. Clouse and P. E. Utgoff. A teaching method for reinforcement learning. In *Proceedings of The International Conference on Machine Learning*, pages 92–101, San Mateo, CA, 1992. Morgan Kaufmann.

[3] Jeffrey Clouse. Learning from an automated training agent. In Diana Gordon, editor, *Working Notes of the ICML '95 Workshop on Agents that Learn from Other Agents*, Tahoe City, CA, 1995.

[4] Colleen E. Crangle and Patrick Suppes. *Language and Learning for Robots.* Cambridge University Press, Stanford, CA: Center for the Study of Language and Information, 1994.

[5] F. Hayes-Roth, P. Klahr, and D. J. Mostow. Advicetaking and knowledge refinement: An iterative view of skill acquisition. Technical report, Rand Corporation, 1980.

[6] Richard Maclin and Jude W. Shavlik. Incorporating advice into agents that learn from reinforcements. In *National Conference on Artificial Intelligence*, pages 694–699, 1994.

[7] Richard Maclin and Jude W. Shavlik. Creating advice-taking reinforcement learners. *Machine Learning*, 22(1-3):251–281, 1996.

[8] John McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationary Office.

[9] Y. Abu Mostafa. Learning from hints in neural networks. *Journal of Complexity*, 6:192–198, 1990.

[10] C. W. Omlin and C. Lee Giles. Training second-order recurrent neural networks using hints. In D. Sleeman and P. Edwards, editors, *Proceedings of the Ninth International Conference on Machine Learning*, pages 363–368, San Mateo, CA, 1992. Morgan Kaufmann Publishers.

[11] R. Reiter. *Logic and Data Bases*, chapter On Closed–World Data Bases, pages 55–76. Plenum Press, 1978.

[12] P. Riley, M. Veloso, and G. Kaminka. An empirical study of coaching. In *Distributed Autonomous Robotic Systems 6*, pages 215–224. Springer-Verlag, 2002.

[13] William D. Smart. *Making Reinforcement Learning Work on Real Robots*. PhD thesis, Department of Computer Science, Brown University, May 2002.

[14] William D. Smart and Leslie Pack Kaelbling. Reinforcement learning for robot control. In *Mobile Robots XVI (Proc. SPIE 4573)*, 2001.

[15] William D. Smart and Leslie Pack Kaelbling. Effective reinforcement learning for mobile robots. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation (ICRA 2002)*, 2002.

[16] S. B. Thrun. Efficient exploration in reinforcement learning. Technical Report CMU-CS-92-102, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.