Autonomous Robotic Programming Framework

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Leslie Luyt

Grahamstown, South Africa
November 2009

Abstract

To date in the field of robotics, there has been no significant research into creating a generic framework that will integrate robots with artificial intelligence and learning techniques to create an autonomous robot. This research investigates the feasibility of developing a programming framework that allows autonomy to be added to a robot easily. This objective is achieved by creating a Python programming framework which includes modules for movement, sensor management, and artificial intelligence. The results obtained through testing the framework show that the amount of code required to add autonomy to robotics is on average, reduced by half. Therefore, it can be concluded that using a programming framework such as this one is both possible and beneficial to robotics developers.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2009) [16]:

D.3.3 [Language Constructs and Features]: Frameworks

I.2.6 [Learning]: Connectionism and neural nets

I.2.9 [Robotics]: Autonomous vehicles

General-Terms: Autonomous Robot, Framework

Acknowledgements

The production of this thesis has taken many months and it would be unfair not to acknowledge all those who have assisted me. I would therefore like to thank all those who have contributed and assisted, in any way, to the completion of this project. There are a number of people I would like to specially note.

Firstly, I would like to acknowledge the financial and technical support of Telkom SA, Business Connexion, Comverse SA, Verso Technologies, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Secondly, I would like to acknowledge my parents for all the support, guidance, funding and love they have given me over the past four years. I know I would not have been possible for me to have had this great opportunity had it not been for them.

Finally, I would like to thank and acknowledge my supervisor, Dr. Karen Bradshaw. Without her much valued input, support and guidance this thesis would not have been possible.

Contents

List of Figures

Chapter 1

Introduction

Although the field of robotics is not a new field in computer science, there does not appear to be any significant research on creating generic frameworks that allow the integration of robots with artificial intelligence and learning techniques. With artificial intelligence becoming more and more popular within the field of robotics, research into programming artificial intelligence and learning procedures has become very important. It is also becoming increasingly important to be able to create a robot that can recover from a mistake and not make the same mistake twice.

The ideal robot is one that can 'think independently' and solve basic problems without human interaction or intervention. However, independent thinking is still a way off in artificial intelligence research and so for now it will suffice to be able to create a robot that is as smart as possible and to push the its ability to the limits.

1.1 Problem Statement and Research Goals

This research project investigates the feasibility of a generic robotic programming framework to combine standard robotic operations with artificial intelligence techniques thereby creating an autonomous robot. To conduct this research, a prototype framework will be developed and tested with basic learning tasks to ascertain how flexible the framework is and how easily it can be adapted to the problem at hand.

The developed framework should be able to control and interact will all aspects of the robot, as well as be easily extensible with new features or modules as required. It should

also supply different methods to control the robot, so that the best method can be used for the problem at hand.

The following are the research objectives, based on the above problem statement:

Primary objectives:

- To create a programming framework that allows quick and easy addition of autonomy to a robot.
- To make the programming as easily extensible and adaptable as possible.

Secondary objectives or extensions:

- To adapt the programming framework for different programming languages.

1.2 Thesis Organisation

This remainder of this thesis is divided into five chapters. This section briefly describes the content of each of these chapters.

Chapter 2 introduces, describes and discusses some of the relevant work and research done in the fields of artificial intelligence, robotics, and programming frameworks. It also describes the robotic kit that has been used, as well as discussing compatible programming languages.

Chapter 3 introduces and describes the framework specification that has been designed.

Chapter 4 describes and discusses how the framework specification has been implemented, as well as highlighting relevant decisions made during the implementation phase.

Chapter 5 describes the tests conducted and compares the expected results with the actual results obtained. Any differences between the expected and obtained results are also discussed in the chapter.

Chapter 6 gives a final summary of the project and describes possible extensions to the project.

Chapter 2

Background

2.1 Introduction

This chapter introduces and defines the main concepts and issues related to creating a framework for programming a robot with artificial intelligence. It also includes background concepts on programming artificial intelligence in general, as well as a description and explanation of the robotics kit that has been used.

The process of creating the artificial intelligence programming framework consists firstly of implementing artificial intelligence methods. Secondly, the artificial intelligence needs to be reconfigured to take robotic movement input, or data from the sensors; and produce output that is sensible for robotic processing. Finally, it is required that all of the above is encapsulated inside of a programming framework. This framework should reduce the difficulty of implementing artificial intelligence when working with robotics, and should be versatile enough such that most tasks can be easily accomplished using it.

2.2 Artificial Intelligence

2.2.1 Neural Network Approach

2.2.1.1 What is a Neural Network?

An artificial neural network is a computational model defined by a directed graph; where each node, or vertex, in the graph has inputs and outputs. These inputs and outputs form

the edges of the graph and are associated with a weight function, or a value. This network provides a practical method for learning real, discrete, and vector valued functions from examples. After the learning phase has been completed the associated weights will provide the required output for all training data. However, these weights may be difficult to explain and interpret, and all that can be inferred is just a numerical relationship between two nodes. [3, 16]

2.2.1.2 How does it learn?

The artificial neural network learns by taking in a list of example inputs and the corresponding correct output for each input. For every input given, it uses the current network to compute an output. It then calculates the error between the given output and the current output, and adjusts the weights by backward propagation to minimise the error. After the weights have been adjusted it re-computes the input example. It follows this loop until the network correctly produces the required output for each input, or the learning process returns with an error stating that the current network configuration cannot correctly simulate the function that maps the given input to the given output. The system will then have to alter the structure of the network, by adding or deleting more nodes or layers as appropriate; and then re-attempt the learning process.

Once the learning process has been completed successfully, we have a correct neural network structure and weights for the function we wish to emulate. We can then proceed to feed in inputs for which we do not know the output and observe what the neural network produces. The final output produced by the neural network may not necessarily be correct, as there may be missing cases in the example input and output which lead to only a partial solution being formed. [3, 16]

2.2.1.3 Relevant Research

Yang and Meng[34] suggest that a neural network is sufficient for real-time collision-free path planning in a dynamic workspace, claiming that the model does not suffer from either the “too close” or “too far” problem. Na and Oh[20] proposed a hybrid control architecture based on reactive navigation, that works by using a neural network to classify sixteen topological maps roughly describing the environment.

2.2.2 Bayesian Network Approach

2.2.2.1 What is a Bayesian Network?

A Bayesian network is a probabilistic graphic model defined by a directed acyclic graph; where each node represents a random variable and the edges between nodes represent probabilistic dependencies between the random variables associated with the connected nodes. After the learning phase has been completed the probabilities associated with the edges will provide the required solution. These probabilities tend to be relatively easier to interpret than the neural network weights. [2, 3, 16]

2.2.2.2 How does it learn?

When the structure of a Bayesian network is unknown at the outset, it is necessary that we learn the structure from the given training data. This problem is known as the BN learning problem, which can be stated informally as follows: Given training data and prior information, estimate the network structure and the parameters of the joint probability distribution in the BN. Once this initial estimation has been done, the system falls into one of the four cases shown in Table 1.

Table 1: Four cases of BN learning problems

Case	BN structure	Observability	Proposed learning method
1	Known	Full	Maximum-likelihood estimation
2	Known	Partial	EM (or gradient ascent) MCMC
3	Unknown	Full	Search through model space
4	Unknown	Partial	EM + search through model space

For case one, the training can be performed by maximising the log-likelihood of each node independently.
For case two, the expectation maximisation algorithm can be used to find a locally optimal maximum-likelihood estimate of the parameters.
For case three, the goal is to find a directed acyclic graph that explains the data. An approach to solving this NP-Hard problem is to take the naïve approach and assume that each node is conditionally independent, thereby simplifying the problem.
For the fourth case, finding the directed acyclic graph is an intractable problem. Thus, an approach is to use asymptotic approximation to the posterior called Bayesian information criterion. This approach will marginalise out the hidden nodes and parameters. [2, 3, 16]

2.2.2.3 Relevant Research

Olivier Lebeltel et al.[12] proposed a new method of programming robots called BRP (Bayesian Robot Programming), which uses a Bayesian Network to control the decision making of the robot. The Bayesian network has many diverse uses, one of which is to emulate an expert system as shown by Wiegerinck et al.[33].

2.3 Programming Frameworks

2.3.1 What is a Programming Framework?

A programming framework is an abstraction of complicated, or simple tasks that reduce the time it takes for a programmer to perform these tasks. A programming framework should be completely reusable and encapsulate any required resources, such as: dynamic shared libraries; nib files; image files; localised strings; header files; and reference documentation in a single package. The programming framework should be relatively intuitive and fairly easy to understand, without the need for the developer to read the source code to get the system working. [1, 27, 35]

2.3.2 Framework Types

There are two main types of programming frameworks with the most common being the procedural libraries, such as dynamic link libraries. The more useful of the two is the object oriented framework which assists will all aspects of programming instead of with a simple task in the case of the procedural libraries.

2.3.2.1 Procedural Libraries

Procedural frameworks are more commonly know as procedural libraries or code libraries, as they contain methods and functions that solve specific problems or perform particular actions. These procedural frameworks are often found in the form of dynamic link libraries, where the methods or functions required to solve a particular problem can be imported from the standalone library and used in the calling program. However, this method does

not offer any significant advantage to the programmer, as the methods and functions simply perform tasks and do not manage objects or significantly abstract away the low level management tasks.

2.3.2.2 Object-oriented Frameworks

Frameworks designed using the Object-oriented approach promise higher productivity for the developer, by increasing the ability for code reuse and better design. As a result a shorter time, in comparison to non-framework based approaches, is taken from the beginning of production to when the product is finally marketed. [1, 27]

2.3.3 Components of a Programming Framework

Generally, a programming framework will model a solution to a specific problem or a specific domain. Thus, they are geared toward providing methods and abstractions that aid in working in the specified domain or solving the given problem.

A programming framework is generally composed of the following:

- Generic setup and instantiation methods
- Specific problem solving methods
- Groupings of methods that problem solve using a specific algorithm [27, 35]

2.4 Fishertechnik Robotic Kit

2.4.1 Fischertechnik Kit Contents

The Fischertechnik set used is the ROBO Mobile Kit with additional sensors as listed below [7].

The components available include:

- Lego-like pieces to construct a robot.

- The Fischertechnik ROBO Interface.
- 2x 9V DC Motors.
- Sensors included in the set are: 2x Light sensor; and 4x Touch sensors.
- Additional sensors: 2x Ultra-sonic Distance sensors; and 1x Colour sensor.
- ROBO Pro software for the ROBO Interface.
- A 9V DC/1A rechargeable battery.
- A 9V DC/1A power converter to recharge the rechargeable battery.
- An instruction booklet containing building instructions for 8 different constructible models.

2.4.2 Controller Specifications

The 16-bit micro-controller has both USB and serial interfaces, and comes with 128 kBytes of flash memory for the download of two separate programs. These programs are retained in the flash memory when the power supply is disconnected. The board also has: four 9V/250mA (max. 1A) motor outputs now with variable speed control; eight digital inputs; two analog inputs for resistances from 0-5kO; two analog inputs for voltages from 0-10V and two inputs for digital distance sensors.[7]

The micro-controller provides connections for the following add-on modules: one ROBO I/O Extension expansion module; one ROBO RF Data Link radio interface; and one interface for a infrared transmitter from the IR Control Set.

All the inputs and outputs run through a 26-pole male connector strip, for convenient connection of finished models through a single 26-pole connector [7].

2.4.3 Supported Languages

Apart from the ROBO Pro software provided by Fischertechnik for the kit; there exists a custom built DLL for the Fischertechnik sets, such that it is compatible with the following major programming languages:

- Delphi 4 (or up to Delphi 7)

- Java 2 (SDK 1.2.2 or higher)
- JScript
- MSWLogo (32bit, engl.)
- Microsoft CSharp 2005 .NET 2.0
- Microsoft Visual C++ 2005 .NET 2.0
- Microsoft Visual Basic .NET 2005
- Perl
- Python
- Visual Basic 6 (includes VBA, VBS)

This functionality can also be extended to any other programming language that can reference a C++ compiled DLL if you desire, but will require a wrapper class to be written so that the functions contained inside the DLL can be referenced [7, 17].

Programming can also be achieved in the Microsoft Visual Studio 2008 Development Suite, by simply updating all the calling methods to be compatible with the updated language and compilers. This can generally be done by simply changing and updating what is specified in the change-log between Microsoft Visual Studio 2005 and Microsoft Visual Studio 2008.

2.5 Programming Languages

2.5.1 Evaluation of ROBO Pro

2.5.1.1 Description of ROBO Pro

The ROBO Pro software is a flowchart programming environment provided by Fischertechnik for the ROBO Interface. Being a flowchart programming environment, it is easy for beginners to use as they can string blocks together without knowing the inner workings. In addition, program operation is easier to understand as all the details are abstracted away by the flowchart blocks in an object-oriented fashion.

The ROBO Pro software consists of various software modules, which can interchange data between themselves or even subroutines using not only variables but also graphical or flowchart connections. Subroutines are stored in a library and can be used without the need for understanding the internal workings of the subroutine.

The graphical programming language ROBO Pro provides all the key elements of a modern programming language, such as arrays, functions, recursion, objects, asynchronous events and quasi-parallel processing. All programs are translated directly into machine language for efficient execution on the ROBO Interface. This makes it a useful tool even for professional programmers.

When running the ROBO Pro software in on-line mode, it is possible to control multiple ROBO Pro interfaces in parallel for large-scale models; and to make custom control panels which include switches, controllers and display elements. [7]

2.5.1.2 ROBO Pro Limitations

Given the wide functionality and seemingly limitlessness of the ROBO Pro language, one would think this would be the ideal programming language for any development using the ROBO Interface. However, this is not the case as the ROBO Pro language lacks portability and only runs on Microsoft Windows operating systems. The ROBO Pro software is also not translatable, and has no translation tools to other programming languages. Since ROBO Pro software is not freeware or open source, if one wants to make use of anything developed using this software one must first purchase a licence before using the application. Due to it being untranslatable, it is not ideal for research; as the research becomes severely limited and only relevant to those with a ROBO Pro licence. However, the most important factor is that the ROBO Pro language is not framework compatible. There is no way to create a further level of abstraction upon the one in which it currently operates. [30]

2.5.2 Evaluation of C++

2.5.2.1 Description of C++

C++ is a standardised, procedural, cross-platform, dynamic, object-oriented, high-level programming language. The C++ language is, in essence, an extension of the C language to include the Object Orientated Design approach to programming. C++ has a large

collection of standard libraries that can be imported and used, and has high community support due to its extensive use. [5, 31]

2.5.2.2 Benefits

C++ has great portability making it possible to compile the same C++ code on almost any type of computer and operating system, without making any changes. Code written in C++ tends to be very short in comparison with other procedural languages, since the use of special characters are preferred to long key words. This preference of special characters decreases the amount of time necessary for the programmer to spend typing, and thus increases their productivity.

C++ is designed as a modular language, with an application's body often being made up of several source code files that are compiled separately before being linked together. This saves in compile time when modifying large applications, as it is not necessary to recompile the complete application but only the file that has been changed. In addition, this characteristic allows the linking of C++ code with code produced in other languages, such as Assembler or C.

C++ compiles to native byte-code for a specific architecture, and thus the resulting code is very efficient and can be run on the processor without any further interpretation or translation required. The efficiency is also improved by the fact that C++ functions as both a high-level and low-level language, and by the reduced size of the language itself; the produced byte-code is well optimised for the system is was compiled on. [31]

2.5.2.3 Limitations

Although the conciseness of C++ is an advantage for programmers, combined with the wealth of operators available to the programmer; it is also a disadvantage as it can make code difficult to follow. Given this and C++'s freedom of expression with regards to pointers, can make very obfuscated code that is impossible to follow and debug. [21]

2.5.3 Evaluation of Python

2.5.3.1 Description of Python

Python is an interpreted, interactive, cross-platform, dynamic object-oriented language, very-high-level programming language (VHLL) that provides strong integration and support when combined with other languages. Python is simpler, faster to process, and more regular than classic high-level languages. Python also comes with an extensive set of standard libraries, providing basic functionality in many areas by simply using and including the correct library. Python is stated as boasting a multitude of features, such as: exception-based error handling; high portability; full modularity; supporting hierarchical packages; intuitive object orientation; very clear, readable syntax; very high level dynamic data types. [14, 24, 29]

2.5.3.2 Benefits

Python has been designed to be modular, with a small kernel that is extended by importing extension modules, or packages. The standard Python distribution includes a diverse library of extensions for operations ranging from string manipulations and regular expression handling, to Graphical User Interface generators, operating system services, and debugging and profiling tools. New Python extension modules can be created to extend the language, and can be written in Python, C or C++. [14, 26, 29]

Python, as with most other scripting languages, supports numerous high-level constructs and data structures that enable you to write shorter programs than those, with similar functionality, written in C, C++, C#, or Java. Where Java requires a loop, Python may require only a single line statement to perform the same operation. This significantly increases the maintainability and readability of code, thereby producing a better end product than possible with more traditional languages. [11, 26, 29]

2.5.3.3 Limitations

Since Python is an interpreted language, some performance loss is to be expected. This is a direct result of hardware-independent byte-code being converted to hardware-dependant byte-code, by being run through an interpreter before execution can occur. However, Python code runs with reasonable performance, sufficient not to be a significant loss; and

in most cases will only run marginally slower than C code for the same tasks. The Python coding style used can also alleviate this problem by using extensions, such as the Numeric extension, which provides good performance when working with large arrays of numbers. [11, 14, 26, 29]

2.6 Summary

Artificial intelligence techniques were evaluated and it was found that both the neural networks approach and the Bayesian networks approach are very powerful techniques that can be very useful tools in learning applications for robotics. Both of these approaches have sufficient power to adapt to changing environments, and after the initial learning phase is completed will perform well in most situations.

The Fischertechnik kit was explored, and each of its components investigated to ascertain the full extent of their usability. The micro-controller included with the set was also sufficiently researched to clarify exactly what the kit is capable of.

The essence of creating a programming framework was also researched. This included: what to put into the framework; how to structure the framework; and how best to create a framework useful to the end-user.

Finally, both the Python and C++ programming languages were evaluated, and were found to be very powerful languages. The clean, simple syntax of Python, with its advanced built-in data types and the expansive standard libraries, make it an attractive programming language for beginners and experienced programmers alike. On the other hand, the powerful features and speed of C++, combined with the extensive libraries available, make it a good candidate for any advanced development. Both Python and C++ are highly extensible with a powerful modular system for easily adding extra functionality or libraries. All the above features make Python and C++ both highly suitable languages for use in creating an artificial intelligence framework extension.

Chapter 3

Framework Design

3.1 Introduction

As described in the project aims in *Chapter 1*, the main goal of the project is to produce a working prototype framework for the programming of autonomous robots. This chapter introduces and describes the framework model and design that has been used to achieve this primary objective.

This chapter starts by examining and discussing the overall framework design. We describe each class contained in the framework and the objective the class was designed to accomplish. We continue by describing the framework organisation. This includes describing where and how each class fits into the overall framework design. The chapter concludes with a section describing how the different classes in the framework were designed to interact, and thus how they produce a self-contained framework that accomplishes the primary goal of the project.

3.2 Design Considerations

The framework is object-oriented by design to take advantage of the benefits of class hierarchy and inheritance. It is the use of this object-oriented design that allows greater code re-use, code extensibility, and maintainability of the framework [27, 35]. All these features mean the object-oriented benefits far outweigh the procedural framework benefits and thus the object-oriented approach has been used. All of these features are key to the success and continuation of any framework model as a usable problem solving tool.

3.3 Framework Overview

Figure 3.1 shows the overall framework module design hierarchy, while *Figure 3.2* depicts the framework sensor design hierarchy. All the components of these designs are explained in *Section 3.4*.

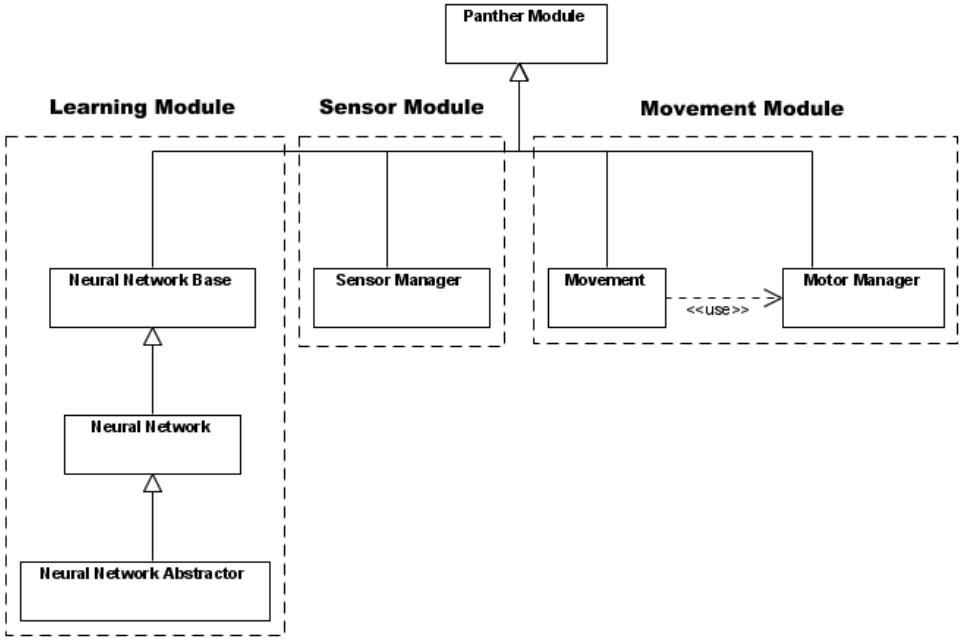


Figure 3.1: Framework Module Hierarchy

3.4 Classes and Interfaces

This section lists each class in the framework model and describes what each is designed to accomplish.

3.4.1 Panther Module

This module contains all the base interfaces for basic robot operations.

3.4.1.1 Panther Robot Interface

The Panther Robot interface, as shown in *Figure 3.3*, describes the basic robot which any hardware specific robot class must implement to make use of this framework. The

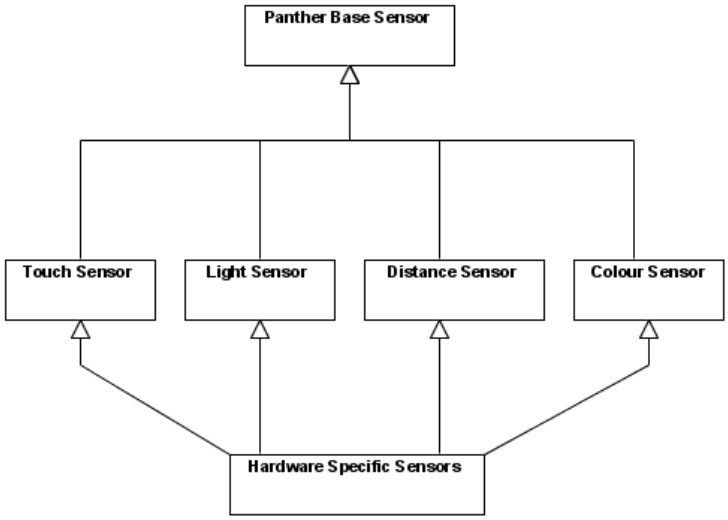


Figure 3.2: Sensor Class Hierarchy

subsidiary framework classes rely on the methods described in this interface, and thus must be implemented correctly for the framework to function. Only the most basic methods are included in the interface, as it is impossible to tell what kind of functionality the implemented robot will have. However, since the framework is designed for autonomous robots, we can assume it will have basic movement capabilities.

The abstract methods described in the interface are the following:

- `set motor` - set a motor to a specific state
- `set motors` - set all motors to a specific state
- `pause` - sleep for a specified amount of time

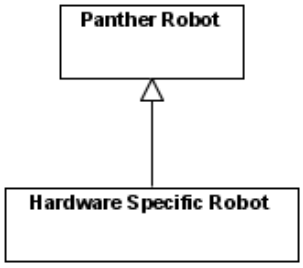


Figure 3.3: Panther Robot Interface Class Hierarchy

3.4.1.2 Panther Module

The Panther Module interface is the basic building block for enhancement modules in the framework. The Panther Module also keeps track of the Panther Robot instance it is attached to, thus allowing sub-classes to make calls to the Panther Robot instance.

3.4.1.3 Panther Base Sensor

The Panther Base Sensor interface is the base model for all sensors and must be implemented by the hardware specific class for each sensor type to allow the framework to function correctly.

3.4.1.4 Panther Base Motor

The Panther Base Motor interface is the base model for all motors and must be implemented by the hardware specific motor classes, as shown in *Figure 3.4*. It includes at the very least, methods to control the motor state.

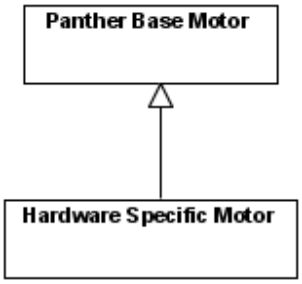


Figure 3.4: Panther Motor Class Hierarchy

3.4.2 Movement Module

This module contains all the classes that govern movement of the robot using motors.

3.4.2.1 Motor Manager

This class is designed to be the controller and manager of all the motors attached to a robot. The main aim of the class is to maintain a list of the motors attached to the robot

and their orientation. It should also mediate access to these motors and provide methods for controlling the functionality of all connected motors with the same orientation.

The methods provided by this class are the following:

- `add motor` - add a motor to the set of motors maintained by the Motor Manager
- `delete motor` - delete a specific motor from the managed set
- `get motor` - get a specific motor from the motor set maintained by the Motor Manager
- `turn motors` - set all motors of the specified orientation to a specific state

3.4.2.2 Movement

Performing basic movement can be challenging if each motor is controlled manually, and this often leads to incorrect motor selection or an incorrect motor state when complex movement is attempted. The Movement class aims to solve this problem by providing basic movement routines which can be used in conjunction with one another to enable the robot to perform complex movements. The Movement class uses the Motor Manager class to perform basic movement tasks.

The Movement class provides at the very least, basic methods for moving the robot.

3.4.3 Sensor Module

This module contains all the classes that govern and manage sensors on the robot.

3.4.3.1 Sensor Manager

The Sensor Module class is designed to be the controller and manager of all the sensors attached to a robot. By maintaining a list of sensors attached to the robot and their types, it mediates access to these sensors and provides methods for controlling the functionality of all sensors connected with the same orientation.

The following methods are provided by this class:

- `add sensor` - add a sensor to the set of sensors maintained by the Sensor Manager

- `delete sensor` - delete a specific sensor from the managed set
- `get sensor` - get a specific sensor from the sensor set maintained by the Sensor Manager

3.4.3.2 Specific Sensor Classes

Each specific sensor type has its own interface that provides methods to obtain the value of the sensor in a manner usable by the framework or framework user. Each of these interfaces inherits from the Panther Base Sensor interface.

3.4.4 Learning Module

This module contains all the classes and interfaces that govern learning within the framework.

3.4.4.1 Neural Network Base

This interface is the base for all neural network type learning classes within the framework. It provides methods and functions that are generic to working with all neural network learning, such as artificial neural networks and Bayesian neural networks. Functions are included to manage the nodes contained in the network, as well as edges between these nodes. In addition, the interface keeps track of and manages the data associated with the network, as well as providing methods to obtain the complete data set. The interface also provides a means to recalculate the node values of the network, and to obtain the value associated with any node.

The set of functions provided in the Neural Network Base interface are the following:

- `add node` - add a neural node to the network
- `delete node` - delete a neural node from the network
- `get nodeset` - return all nodes currently contained in the network
- `add edge` - add an edge to the network

- `delete edge` - delete an edge from the network
- `get edgeset` - return all edges contained in the network
- `add rule` - add a data rule to the data set of the network
- `delete rule` - delete a data rule from the data set of the network
- `get ruleset` - return all the current data rules contained in the network
- `recalculate` - recalculate the value of each data node
- `get node value` - obtain the value of the requested data node

3.4.4.2 Bayesian Network

Although the Neural Network Base provides all the functionality required, it is merely an interface and the specific functionality still needs to be implemented for a particular Neural Network. The Bayesian Network class is such an implementation of the Neural Network Base interface for Bayesian neural network structures. This class implements all the functionality described in the Neural Network Base interface for a Bayesian network, allowing the Bayesian network to be manipulated via these means and used for learning.

3.4.4.3 Neural Network Abstractor

Neural networks tend to be rather abstract concepts and difficult for those not educated in how they work. This class is designed to abstract away the use of neural networks so that simple decisions can be made easily.

3.5 Class Interoperability

3.5.1 History System Specification

One of the most common requirements for learning and artificial intelligence is data capture. Thus, there is the need for the framework to provide a mechanism to capture calls made to the framework, and the results obtained from these calls. However, the

architecture of this mechanism must be implementation specific in order to benefit from the implementation language.

The History System should be configurable in terms of whether the call and arguments or the results of the call are logged. The History System should also be able to log both the call and the results or neither depending on the required logging state. As a result of logging calls it may be possible in some languages to implement a backtracking system that will be able to invert the operation of a function using the parameters passed to the function. This is not strictly required by the History System, however it is a useful feature if implemented.

Chapter 4

Framework Implementation

4.1 Introduction

The goal of this chapter is to describe how the proposed model, described in *Chapter 3*, has been implemented. It describes and discusses the advantages and disadvantages of each implementation decision made, starting with the choice of implementation language. The chapter continues by giving a general overview of the implemented framework, showing how all the classes fit together. Following that, each class and feature of the framework model that has been implemented is described and, if applicable, why it was implemented in that particular way; together with the file structure of this implementation. Finally, the chapter concludes with an explanation of how extensions to the framework can be implemented.

4.2 Language Choice

This section discusses the available languages and the reasons for the final choice of language for implementation. It also describes the advantages and disadvantages of the chosen language.

4.2.1 Language Summary

As described in *Section 2.5*, the languages ROBO Pro, C++, and Python were evaluated for use in the implementation of the framework. All of the languages evaluated contain a

good combination of traits for the creation of a programming framework, with C++ being one of the most widely used languages for framework or module creation. ROBO Pro is the least used of the review languages, which is most likely due to its proprietary nature and GUI only interface that hinders widespread use.

4.2.1.1 Portability

ROBO Pro only has a Microsoft Windows® distribution, which means it is completely non-portable as any program developed using it can only be run on a computer using the Microsoft Windows® operating system[7]. On the other hand, C++ programming languages have compiler implementations for the Microsoft Windows®, Unix, and Linux operating systems, which makes them extremely versatile and portable. Python is also available on a wide range of operating systems, with Python code able to run on any operating system with an interpreter implementation of the mainstream Python implementation, CPython. Jython can be used to compile the Python code into Java byte-code which can then run on any Java virtual machine. This means that Python can be used on any operating system running the Java virtual machine [24].

4.2.1.2 Object Oriented

Flow chart programming languages cannot strictly be called object oriented as they do not have the concept of constructible objects. Some do allow for the construction of functions and procedures, however these are generally just abstracting away a large chunk of flow chart as a single block. Although C++ is an object oriented programming language it lacks some of the finer features of object oriented programming, such as interfaces. This however can be circumvented by using carefully coded `#defines` that effectively replace the key word `interface` with **`class`**. Python, similar to C++, also lacks an interface directive, however in Python's case this can be fixed by careful use of the Python language and the creation of factory classes that can check that an interface has been correctly implemented, as displayed in Zope[36].

4.2.1.3 Computation Speed

C++ used to be the forerunner in computing speed, however, current age interpreted languages, such as Python, tend to run just as fast if not faster in certain instances [13].

Python being an interpreted language however, runs marginally slower than hardware specific compiled languages [9]. This can be mitigated by the use of Python extensions such as py2exe[22], which allow the compilation of Python code to hardware specific byte-code comparable to C++ byte-code. Swig[32], or similar utilities, may be used to integrate C or C++ functions and code into Python allowing computationally heavy code segments to be run from optimised C or C++ code.

4.2.2 Python Selected

After careful consideration of all the factors listed above, Python was adopted as the language of implementation due to its superior high portability. Its support for parallelism and easy importation of C and C++ libraries also contribute highly to its eligibility. Although it does not have full object-oriented compatibility, these features can easily be emulated. As well, Python is a dynamically typed language which provides a significant productivity boost for development over statically typed languages[13]. The dynamic typing does introduce the risk of type errors only being detected at runtime, but this can be mitigated by making use of the PyChecker[23] extension for Python. Python also contains many features that are not implemented by many other main-stream languages, such as the `__getattribute__` method which gives the programmer full control of all access to the attributes of a class. [25]

4.3 Implementation Overview

This section gives a general overview of the implemented model, and describes how the implementation fulfils each part of the model specified in *Chapter 3*.

4.3.1 Model Implementation

The specified model has been implemented in Python, with the Python classes generally matching the required model class or interface name. For example, the Panther Robot specification, as described in *Section 3.4.1.1*, has been implemented as the `Panther_Robot` class which is described in *Section 4.4.2.1*.

Due to Python's lack of inherent support for interface directives, the interfaces required by the model have been implemented as classes with abstract methods. The lack of an

interface directive in Python means that there is no explicit checking whether a class has correctly implemented an interface at compile time, and it is up to the programmer to ensure all abstract methods have been overridden [25].

Explicit checking can be enforced by creating an Interface class and class factories, which check that all abstract methods are present and have been correctly implemented [36]. This however, significantly decreases the ease of learning, understanding and use of the framework and has thus been excluded. Instead, each time an object is passed which should be implementing a certain interface, an inheritance check is made to ensure that the required interface abstract class is included in the super classes.

4.3.2 Class Hierarchy

Since the described model, in *Chapter 3*, uses an object-oriented design and hierarchical structure, the implemented solution must also use these features. These features are required to keep the implementation consistent with the model as well as to gain the benefits of an object oriented approach described in *Section 2.3*.

Figure 4.1 shows the `Panther_Robot` class, which implements the functionality required by the Panther Robot specification in *Section 3.4.1.1*. This class provides abstract basic functionality for a robot, with the `FT_Robot` child class providing the actual implementation of the required features for the Fischertechnik robotic set.

The `Panther_Base_Motor` is the implementation of the Panther Base Motor as described in the model. Although the model specifies that the hardware specific classes should implement this class, this is not required as the `Panther_Base_Motor` passes requests for a motor state change to the `Panther_Robot` class. This means that unless a specific motor feature is required, the `Panther_Base_Motor` can act as a generic motor without a hardware specific implementation.

Since Python does not have inherent support for constant variables, the `Panther_Constants` class is used to provide the implementation with an object that can store constant variables. The `Panther_Constants` class is purely an implementation concept and hence does not appear in the described model. Although only a few of the framework classes rely on the `Panther_Constants` class, all of the hardware specific classes rely heavily on it to know exactly what state should be set or changed on the robot.

The Panther Module specification and derivative classes have all been implemented as specified by the model, with the History System specification being implemented in the

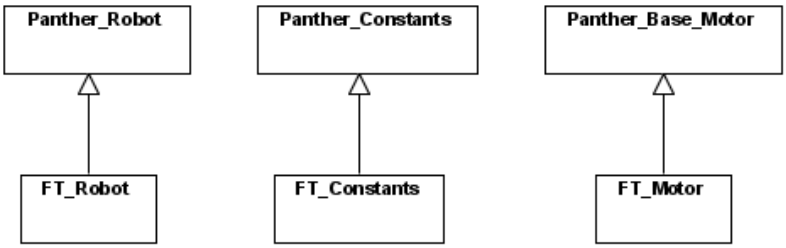


Figure 4.1: Implemented Framework Additional Class Hierarchies

`Panther_Module` class which means that all child classes contain the History System. The hierarchy shown in *Figure 4.2* **almost replicates that depicted in** the framework overview in *Section 3.3*. The Learning Module specifications have been implemented as follows: `Learning_Neural` implements the Neural Network Base specification; `Learning_BN` implements the Bayesian Network specification; and `Learning_Neural_Abstractor` implements the Neural Network Abstractor specification.

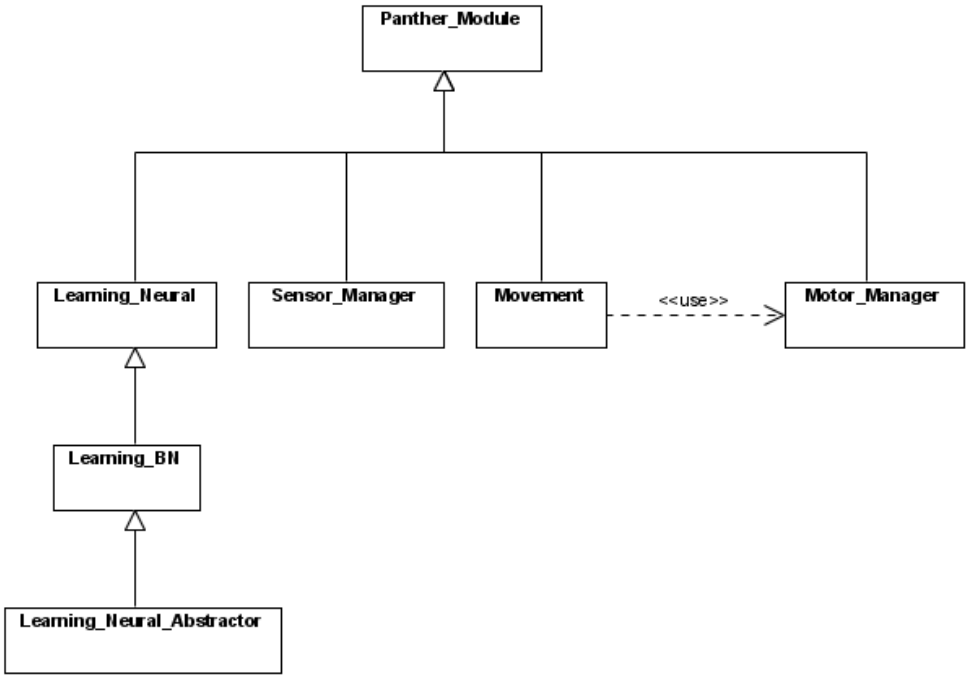


Figure 4.2: Implemented Framework Module Class Hierarchy

Figure 4.3 shows the `Panther_Base_Sensor` class which implements the Panther Base Sensor description from the model. As shown, the framework includes colour (`Color_Sensor`), touch (`Touch_Sensor`), light (`Light_Sensor`) and distance (`Distance_Sensor`) sensor interface classes. All of these sensors have been implemented by the hardware specific classes with the interface names prepended with `FT_`. These hardware specific classes pass their requests to the `Panther_Robot` interface instance. Although the `Panther_Robot`

interface does not contain the methods required by these sensors classes, these classes do in fact interact with a `FT_Robot` instance containing the corresponding method.

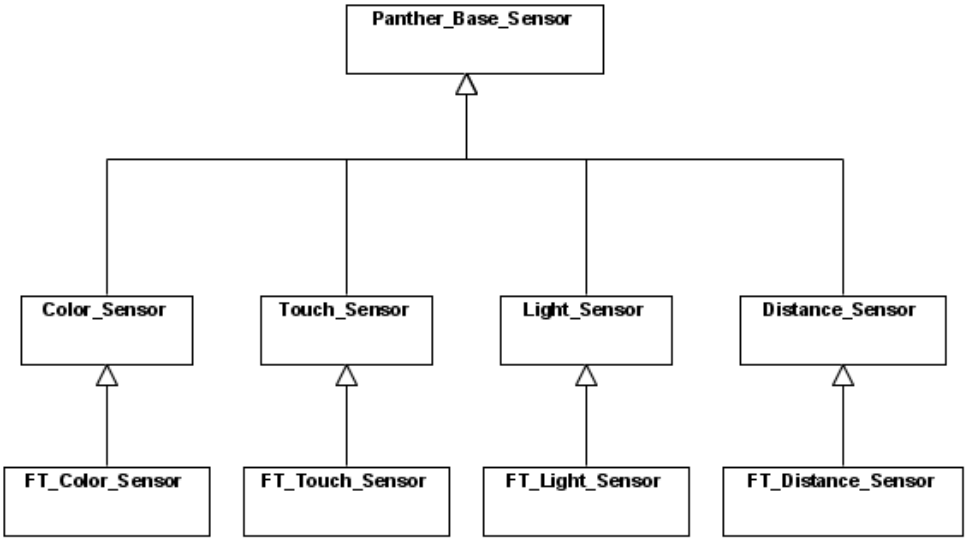


Figure 4.3: Implemented Framework Sensor Class Hierarchy

The implementation also contains error classes which help to debug and understand why the system may not be functioning correctly. These are shown in *Section 4.4.1* and are: NotPantherClassError, ConstTypeError, and InvalidDirectionError.

4.4 Class Implementation

This section lists each class in the framework model and describe the prototype implementation that fulfils the requirements laid out by the framework design. As stated in *Section 4.2.2*, Python is a dynamically typed and therefore all the static types shown in the method signatures below are for demonstrative and descriptive purposes only. Due to the amount of classes and methods in each class, only the important non-obvious method implementations are explained below. For example, a class constructor that only calls the super class' constructor is not included.

4.4.1 Error Classes

Below the errors that may be thrown by the framework are listed as well as a description of each.

4.4.1.1 NotPantherClassError

A `NotPantherClassError` error may be generated when a function is passed an instance of a class that does not descend from the required parent class. Normally a string error message will accompany this error which will indicate which Panther class instance was expected.

4.4.1.2 ConstTypeError

The `ConstTypeError` is generated when an attempt is made to modify a property of the class `Panther_Constants` or any child class of its.

4.4.1.3 EInvalidDirection

The `InvalidDirectionError` is generated when an invalid direction is passed to the `Movement` class described in *Section 4.4.3.2*. The `Movement` class requires that the direction passed is a valid direction and contained in the `Panther_Constants` hardware specific child class.

4.4.2 Framework Interfaces

The following subsections describe how the interface classes within the framework have been implemented.

4.4.2.1 Panther_Robot

Basic robotic functions tend to be completely specific to the robotic system used, hence this class is composed only of abstract methods. Thus, the Panther Robot class contains the following abstract methods which must be overridden by child classes:

`set_motor(motor_ident, motor_direction, speed)`
: This method will set the motor specified by the `motor_number` motor identifier to turn at a speed of `speed` in the direction `motor_direction`.

`set_motors(mode)`

This method should set the status of all motors to the mode specified by `mode`. A common use of this would be to set all the motors to an off or braking state.

`pause(int time)`

This method should pause for the specified number of milliseconds `time`.

`build_robot(Panther_Constants robot_config)`

The `build_robot` function uses the robot builder constants, as described in *Section 4.4.2.3*, defined on the `robot_config` object parameter to construct a robot configuration. A robot configuration is defined as all the object instances required to control the physical configuration of motors and sensors as well as any required learning class instances. The `robot_config` parameter should be specified as either a subclass or and instance of `Panther_Constants`.

This function creates all the required instances and returns them in a dictionary, containing at least the string keys: `Panther_Robot`, containing a reference to the `Panther_Robot` instance; `Panther_Constants`, containing a reference to the `Panther_Constants` instance. When the required instances are created, all parameters to be passed to the class constructors are checked. If any are subclasses of either `Panther_Robot` or `Panther_Constants`, the current instance of `Panther_Robot` is used instead of the specified `Panther_Robot` subclass and the `robot_config` parameter is used instead of the specified `Panther_Constants` subclass.

If a motor manager and motor set were specified in the robot configuration, then the returned dictionary will also contain a string key of `Motor_Manager` containing an instance of the specified motor manager class with all the motors created and added to it. Similarly, if a sensor manager and sensor set were specified the returned dictionary will contain a `Sensor_Manager` key. If a learning class was specified, then the `Learning` key will contain the required instance of the specified classes.

4.4.2.2 `Panther_Module`

The `Panther_Module` is the base for all modules within the framework, and implements the History System. Since the history system implemented in the base of the framework, the History System permeates the entire framework giving the framework a significant adaptability advantage. The History System is implemented as a list, or array, of records that are stored when a function is called or returns, depending on the history logging state.

The History System logging state is stored in `_history_state` and should contain one the following values:

- Zero - denotes the History System is turned off.
- One - denotes the function call and parameters will be logged.
- Two - denotes the function results will be logged.
- Three - denotes the function call, parameters, and results will be logged; as two separate records.

Each history record is composed of: the time the function was executed; the function that was executed; the arguments passed to the function; and the key-worded arguments passed to the function. The time is obtained by using the Python `time.time()` function, which returns the time as a floating point number expressed in seconds since the epoch. The time value is only guaranteed to be accurate to a second even though it is returned as a floating point value, as some systems do not provide functionality for further precision [25].

`__init__(Panther_Robot panther_robot, Panther_Constants panther_const)`

The `__init__` method checks that `panther_robot` and `panther_const` parameters are derived from the correct interface classes, `Panther_Robot` and `Panther_Constants` respectively. The references to these instances are stored as class variables so that they may be used by child classes or accessed so as to expose low level functionality. Although the `Panther_Constants` instance could be stored in the `Panther_Robot` class, having each module keep a reference to a constants class instance gives greater extensibility by allowing individual modules to use different constant instances.

This method also rebinds all functions currently attached to the running instance excluding those in the disallowed list. This rebinding of functions occurs at runtime and therefore includes all functions introduced by child classes. The disallowed list is a property of the `Panther_Module` class and contains a string list of the required function names. The rebinding of the functions is accomplished by creating a new function with logging code, using the `log` function described below, and replacing the original function attached to the class with this altered one.

```
#distance between first and last history
   records
tdist = history(end)-history(start)+1
t = ttofind-tstart
#percent t is of full distance
ratio = t / tdist
#use ratio to find approx record position
guess = ratio*length(history)
```

Figure 4.4: Naïve record position estimation

`void log(function)`

This function rebinds the function given by `function` and returns the same function with logging code before and after the original code in the function [25]. It also adds an extra property to the function `_logged` and assigns it to TRUE. This extra property identifies that the logging code has already been applied, and is checked before applying the logging code to the given function. This makes sure that the logging code is not applied to the same function twice.

`list get_history(int number = 0, int timecode = 0, int mode = 1)`

This function returns a list containing the logged calls to methods and their results, in a varying fashion depending on the parameters passed. If no parameters are different from the default values, then a copy of the entire history listing is returned.

If the `timecode` parameter is greater than zero, then the system will attempt to find the first record with the closest time value to the value specified by `timecode`. It does this by making a naïve estimate based on the time in the first and last records in the history using the algorithm shown in *Figure 4.4*.

This estimated record position is looked up in the history list and a linear search is started from the estimated position toward the required time. The linear search continues until it finds the required time or the closest record to the required time. Once the closest record is found, the function returns the closest record if `mode` is zero; and the list of records starting from the record containing a time value less than or equal to the required time value to the end of the history list if `mode` is one. A further case specifies that if `mode` is one and `number` is greater than zero, then the function will return a list starting from the record containing a time value less than or equal to the required time value and continuing toward the end of the history list containing `number` records.

Finally, if `number` is greater than zero with `timecode` specified as zero then the function will return the last `number` entries in the history listing.

`void set_history_state(int state)`

This method checks that `state` is a valid History System state and then sets the class variable `_history_state`. To set the default History System state in a child class, the child class may either redeclare `_history_state` and specify a new value or set the value at the end of the `__init__` function.

`void backtrace(list command_list, boolean log = false)`

Backtracing or backtracking is the act of retracing one's course. The `backtrace` function does exactly this, it reverses the effects of all reversible functions listed in the `command_list` variable. The `command_list` variable may be either a list or list iterator with each record being of the same form as a History System record. This backtracing is accomplished by running the function with `invert_` prepended to the function name and passing it the same arguments as passed to its sister function. If the function name starts with an `invert_` prepend, then prepend is removed and the function with the resultant name is executed.

The use of this function relies on the fact that for each method that changes the state of the robot, there is a sister method with `invert_` prepended to the function name that reverses the effects of the function's state changes to the robot. If this function does not exist, that state change cannot not be reversed and hence back-tracing will fail to be accurate from any previous point.

4.4.2.3 `Panther_Constants`

The `Panther_Constants` class is the configuration class for the framework. The constants defined herein set the parameters and values for use by the framework when interacting with the hardware specific classes. The `Panther_Constants` class contains only a few methods which help manage the constant variables. The recommended constant variables, divided into categories, are specified below together with a description for each category. Only the robot building constant variables have been given a demonstrative type as the values of the other constants are not explicitly used within the framework, but are merely passed to the hardware specific classes. Thus these constant values may be assigned arbitrary values, or values solely beneficial to the hardware specific classes to which they are passed.

To comply with the history system specification, some of the constants contained within the class need to be reversible, so that inverse functions can in turn invert the constants passed to them, before being passed to the non-inverse function. These inverse constants names, as with the inverse functions, should be prepended with `invert_`. The inverse constants should also not store the value of their inverse but the name of the constant that contains the inverse constant value as a string. At runtime, this name is converted to the value contained in the constant. Using this dynamic binding methodology means that even if a constant pointed to by an inverse constant is overridden in a child class, the inverse constant pointing to it will always contain the correct value.

`__init__()`

Upon creation of an instance of the `Panther_Constants` class or a child class, the `__init__` function will dynamically bind all the inverse constants to their correct values. It does this by looking up the value of all constants that start with `invert_`. For each constant value looked up, the value returned is in turn used as a lookup on the instance and the value returned is assigned to the constant. If the constant value looked up is not a string type, that constant is ignored and the `__init__` function continues to the next matching constant value.

`void __setattr__(String name, value)`

The `__setattr__` method is called by the Python interpreter when another class attempts to modify a property value of the class. This method throws a `ConstTypeError` whenever called and thereby stops any modification to the properties contained within the class. Since Python does not have a constant type, this in effect, creates a constant class containing a set of constant variables that operate in a similar fashion to constant variables in other languages. For the same reasons the `__delattr__` function, which stops attributes from being deleted from the class, has been included.

`lookup(String name)`

This method returns the constant value of the property called `name` contained within this class. This method is used by the `Movement` class in conjunction with the `get_orient_name` function to obtain the value of the orientation constants.

Relative Directional Constants:

Directional constants define the directions, relative to the current robot heading, that are available for the framework to work with. The framework defines the

following: `left`, `right`, `up`, and `down` as well the inverses associated with these. These constants map to string versions of themselves and aid in the passing of any direction constant to a function. This allows a function passed a direction to know which constants to lookup, and allows it to discern the inverse constant correctly if required. However, as stated by the heading, these constants should be relative directions to the robot. An absolute directional system is discussed in the extensions section, *Section 6.3*.

Orientation Constants:

The orientation constants dictate where a motor is orientated with respect to a top down view of the robot and can be named at the user's discretion. These constants are most commonly used by the `Movement` class and although the `Movement` class does not require any specific orientation constants to be present, left and right type constants are recommended for 2D plane movement. Since the constants can be named at the user's discretion, inverse constants are required, where appropriate, so that the framework can ascertain which orientations are opposite each other. These inverse constants should share the same name as their counterparts with `invert_` prepended to the name.

Motor Constants:

The motor constants allow a constant value to be associated with a motor turning direction. This constant value will be passed to the hardware specific class for it to implement the motor turn direction. The `Movement` class uses these constants to turn the motors in the required direction, and requires that at least `motor_forward` and `motor_backward` be defined.

Speed Constants:

The speed constants are the most abstract constants in the framework and although not strictly required by any part of the framework, these constants may be useful for developers who wish to manipulate the motor turn speeds for their program quickly. However, they must be interpretable by the hardware specific classes which implement the motor turn speeds. Some useful constants included in this category are: `speed_fast`, `speed_medium`, `speed_slow`, and `speed_stop`.

Robot Builder Constants:

The robot builder constants should all start with `robotbuild_`, and define the class instances to be created when using the `Panther_Robot.build_robot` function. The constants recognised by the `Panther_Robot.build_robot` function are the following: `motor_manager`, `motors`, `sensor_manager`, `sensors`, and `learning`.

The `motor_manager` and `sensor_manager` constants are defined as a tuple, with the first element being the class to instantiate and the following elements the parameters to pass the constructor. Conversely, the `motors`, `sensors`, and `learning` constants are lists of tuples, with each tuple containing, as the first element, the class to instantiate. The remaining elements in the tuple are passed as parameters to the constructor class constructors.

4.4.2.4 `Panther_Base_Sensor`

The `Panther_Base_Sensor` class does basic housekeeping for a sensor by keeping track of the sensor identifier in `_sensor_ident`. It also keeps a record of the `Panther_Robot` instance to which it is attached and its name in `panther_robot` and `_sensor_name` respectively. All other methods associated with a sensor are generally sensor specific, and therefore only implemented in the sensor specific and hardware specific classes.

4.4.2.5 `Panther_Base_Motor`

The `Panther_Base_Motor` class provides basic services for a motor. The class keeps a record of its motor name in `motor_name`; its motor identifier in `_motor_ident`; the `Panther_Robot` instance to which it is attached in `panther_robot` and the specified motor orientation in `_orientation`. The class also provides a `set_motor` function, which allows the programmer direct control of a motor if desired.

`set_motor(motor_direction, speed)`
: An alternate control method for a motor is to use the motor's `set_motor` function to control its state. This function calls the `set_motor` function of the `Panther_Robot` instance using the stored `_motor_number` to pass as the motor identifier.

4.4.3 Framework Classes

4.4.3.1 `Motor_Manager`

To ease the management and control of motors within the framework, the `Motor_Manager` class can be used to maintain a list of motors connected to the robot. Although the class

is a child class of `Panther_Module`, the default history state is to set to off as it is highly unlikely logging each motor call would be useful information. However, it does provide inverse methods consistent with the history system specification that allow use of the history system and `Panther_Module.backtrace` functionality if required.

`add_motor(motor_class, motor_name, motor_identifier, orientation)`
: This method adds a motor to the list maintained by the class, by adding the the key `motor_name` to the managed directory. The function is quite flexible in that the `motor_class` specified may be either an instance and child of the `Panther_Base_Motor` class, or it can be a string specifying the name of the child class of `Panther_Base_Motor` to be instantiated. If `motor_class` is specified as a string, the function will try to create an instance of the class with that name, and thus requires all parameters to be specified. However, if `motor_class` is an instance and child of `Panther_Base_Motor` then any specified parameters are set on the instance before adding it to the dictionary.

`get_motor(motor_name, motor_identifier)`
: The `get_motor` function attempts to find the motor specified by either `motor_name` or `motor_identifier` in the managed list. If the `motor_name` parameter is specified it will be used first. Thus the `motor_identifier` parameter will only be used if either the lookup in the managed dictionary on `motor_name` fails, or `motor_name` is not specified. This is due to the fact that a lookup using `motor_identifier` must iterate through all motors in its attempt to locate the correct motor, and is thus very resource intensive.

`del_motor(motor_name, motor_identifier)`
: This function attempts first to locate the motor specified by either `motor_name` or `motor_identifier` by calling the function `get_motor`. If a motor is returned, then it is removed from the managed dictionary.

`turn_motors(motor_orientation, motor_direction, speed)`
: The `turn_motors` function is designed to allow easy motor control for all motors with the same orientation. This allows the grouping of motors by orientation. Since the orientations can be completely defined by the user, this allows motor groups to be created that may be controlled independently.

`set_motors_state(state)`
: This function sets the state of the motors in the managed list. It does this by using the `Panther_Robot.set_motors` method, and passes the specified state. Another

way of implementing this could be to iterate through all the motors, setting the state of each. However the hardware may contain specialised functions to control all motors at once, which would be far more efficient than this method. Hence, the implementation of this function is deferred to the hardware specific class.

4.4.3.2 `Movement`

The `Movement` class provides basic movement methods for the framework, and requires that at least the orientation and motor constants be specified in the `Panther_Constants` instance it is passed on creation. It uses these orientation and motor constants to perform movement operations relative to the robot. The `Movement` class also has default fall-back on attribute access to the `Motor_Manager` class, and therefore all the `Motor_Manager` functions are accessible directly in the `Movement` class.

Functions are also provided for turning the robot in multiple ways, and these require a `direction` parameter which is relative to the robot. This `direction` parameter is looked up as an orientation to find the correct motor sets to move. All functions provided by the `Movement` class are invertible and pass the looked up orientation inverses to the non-inverted function in order to provide the inverse operation. Thus, by doing this inverse look up the inverted functions take exactly the same parameters as the non-inverted functions and produce the inverse movement. However these inverse functions rely on the looked up orientations being invertible and the existence of at least one motor on both the orientation and inverse orientation side to perform correctly. If the specified orientation is not invertible, the `Movement` class throws an `InvalidDirectionError` error.

`move(motor_direction, speed, time, *orientation)`
: The `move` function provides basic movement ability for any group of orientations. The `*orientation` parameter works in a similar way to the C++ ellipsis functionality by grouping all the remaining non-keyword specified parameters into an array and assigning this to the parameter name given. If no orientations are specified, the `Motor_Manager.turn_all_motors` function is called, thereby turning all orientations.

`move_groups(speed, time, *orientations)`
: The `move` function above only allows for turning a set of orientations in the same motor direction. This function adds the ability to turn multiple orientations each with its own motor direction. This additional functionality is implemented requiring

tuples of orientation and motor direction to be specified after the time and speed parameters, and again uses the * functionality of Python to group them into a list. This list is then processed and each orientation is set to the specified motor direction at the required speed before pausing for the specified time period. Allowing this grouping of orientation and motor direction makes concurrent movement with multiple orientations relatively simple, and can be easily used to allow a robot to move in any required direction.

`turn(direction, time, speed)`
A simple way of turning a robot is to turn a single motor forward with this motor being on the opposite side of the robot to the turning direction. This is accomplished by using the looked up orientation and inverting it and then turning the inverse orientation motor group forward.

`turnrev(direction, speed, time)`
Another simple way of turning a robot is to take the opposite approach and turn a single motor backward with this motor being on the same side of the robot to the turning direction. The `turnrev` function accomplishes this by using the looked up orientation and then turning that orientation motor group backward.

`turnspin(direction, speed, time)`
When a robot is manoeuvring in tight spaces, it is often useful to be able to turn without changing position. The `turnspin` function implements this idea and turns both the orientation and inverse orientation. The orientation motor group looked up from the `direction` parameter is turned backwards while the inverse orientation motor group is turned forward at the specified speed. This causes the robot to turn almost in place without significantly changing position.

4.4.3.3 `Sensor_Manager`

The `Sensor_Manager` class operates in a similar way to the `Motor_Manager` class, in that it is used to maintain and manage a list of all the sensors attached to the robot. Differences are associated with the fact that it initialises the history system to state two and it contains specific sensor functions instead of the motor control functions contained in the `Motor_Manager` class.

`add_sensor(sensor_class, sensor_name, sensor_identifier)`
This method adds a sensor to the managed dictionary with the key `sensor_name`.

The `sensor_class` parameter may be specified as either an instance and child of `Panther_Base_Sensor` class, or it may be a string specifying the name of the child class of `Panther_Base_Sensor` to be instantiated. If the `sensor_class` parameter is specified as a string, the function uses the data contained in the string to create an instance of that class, passing all parameters specified to the constructor. However, if `sensor_class` is a child instance of `Panther_Base_Sensor`, any specified parameters are set on the instance before adding it to the managed dictionary.

`get_sensor(sensor_name, sensor_identifier)`

The `get_sensor` function attempts to find the sensor specified by either `sensor_name` or `sensor_identifier` in the managed list. If the `sensor_name` parameter is specified it will be used first. Thus the `sensor_identifier` parameter will only be used if either the lookup in the managed dictionary on `sensor_name` fails, or `sensor_name` is not specified. This is due to the fact that a lookup using `sensor_identifier` must iterate through all sensors in its attempt to locate the correct sensor, and is thus very resource intensive.

`del_sensor(sensor_name, sensor_identifier)`

This function attempts first to locate the sensor specified by either `sensor_name` or `sensor_identifier` by calling the function `get_sensor`. If a sensor is returned, then it is removed from the managed dictionary.

`call_sensor(function_name, sensor_name, sensor_identifier, *arguments)`

Sensors within the managed dictionary may be accessed by finding the required sensor using the `get_sensor` function and then calling the required method upon the sensor instance to obtain the sensor's current value. However, this can be rather cumbersome when accessing multiple sensors and thus the `call_sensor` function may be used to obtain the current sensor value without first obtaining a reference to the required sensor instance.

This function accomplishes its task by first obtaining the sensor instance by passing the `sensor_name` and `sensor_identifier` parameters to the `get_sensor` function. Once the required sensor instance has been obtained the string parameter `function_name` is converted to a function using a lookup on the sensor instance. This function is then called and passed all parameters contained in the `arguments` parameter array. If the history system is still in state two, the function call and result obtained are logged.

4.4.3.4 Specific Sensor Classes

The specific framework sensor classes provided are listed in this section with each class a child class of `Panther_Base_Sensor`, as described in the framework model. However, these are interface classes and so only provide abstract methods the implementation of which must be completed by the hardware specific sensor classes. To understand these generic sensor type specific classes, each has been listed together with a small description of the class and the kind of output to be expected.

`Touch_Sensor`
: Touch sensors are generally push button sensors, and therefore will likely return either a one or a zero when queried.

`Distance_Sensor`
: The distance sensors currently employed in modern robotics are either laser or ultra-sonic distance sensors, both of which are likely to return the distance measured in millimetres or centimetres.

`Light_Sensor`
: A light sensor will most commonly detect light intensity and return a number indicating the intensity of light reaching the receiver. The number returned is often a floating point number between zero and one, however some sensors use integers and a comparable scaling system.

`Color_Sensor`
: Colour sensors are commonly composed of three light dependant resistors (LDR), each with a different light filter in front of them. These are three different filters, with each filter allowing only one colour through, one for red, green and blue. The colour sensor returns a value indicating the intensity of light reaching each of the LDRs. Colour sensors can also be used as light sensors by using only that part of the returned value that indicates light intensity. However, they are generally more expensive than light sensors so this is not common practise.

`Lamp_Sensor`
: A lamp sensor in this regard in not actually a sensor but rather a small light bulb that will produce light when a current is put across it.

4.4.3.5 `Learning_BN`

The `Learning_BN` class allows management of a Bayesian network making use of the OpenBayes Python library [8] to create and manipulate the Bayesian networks. The class manages a dictionary of neural nodes within the network, as well as a dictionary of the edges between these nodes. It also keeps a list of all the data rules for the constructed network, so that learning can be achieved easily after adding new rules.

The `Learning_BN` class managed dictionaries and lists can be obtained using the `get_nodeset`, `get_edgeset` and `get_ruleset` functions. Each of the managed dictionaries and lists has a specific format and stored data types that need to be understood when working with them, and are explained hereafter. Network nodes are stored in the managed node dictionary as OpenBayes BVertex [8] instances with the node name as the key. The managed edge dictionary keeps the connected edge names in a tuple connecting the first edge to the second edge. These edges are kept in the dictionary under the key of the first edge name and second edge name separated by an underscore. The managed data rule list keeps a list of every data rule entered in no particular order. Since data rules are not unique it is not uncommon for the list to contain multiple copies of the same piece of data. Although this could be considered data duplication, this duplication has significant meaning for the statistics involved with the Bayesian network. The fact that the same rule occurs more than once means this particular data rule is more prominent and thus the node probabilities should reflect this prominence. [19]

`add_node(node_name)`

When adding a node to the managed node dictionary, first a check is made that a node of name `node_name` does not already exist. If the node does not exist, an OpenBayes BVertex instance is created using the node information. This BVertex instance is then added to the managed node dictionary with the key `node_name`. If there is currently another data in the rule data set, the current rule data set is updated by assigning each rule with zero data to the new node.

`delete_node(node_name)`

Deleting a node is similar to adding a node, in the sense that after the node has been deleted from the managed list the rule data needs to be updated. This function accomplishes this by deleting all data pertaining to the deleted node from each rule in the rule data set. Due to the high level of potential data loss, deleting nodes without first creating a copy of the rule data set, using the `get_ruleset` function, is not advisable.

`add_edge(node1, node2)`

Adding an edge to the managed edge dictionary is a relatively simple task. The node dictionary is checked for entries of `node1` and `node2`, and if one is missing the request is ignored. Otherwise, the node names `node1` and `node2` are saved as a tuple and this is added to the managed edge dictionary with key `node1_node2`.

`delete_edge(node1, node2)`

The `delete_edge` function is far simpler than its counterpart `add_edge` function. To delete an edge, it checks for the existence of a key `node1_node2` in the edge dictionary and if found deletes the entry.

`add_rule(node_dict)`

Adding a rule to the rule data set requires that a dictionary be passed as the parameter `node_dict`. The `node_dict` dictionary should have the node names as its keys, with the associated values being the state of the node. If any nodes are not specified in the `node_dict` dictionary, they are set to zero before the rule is added to the rule data set. This is done to enforce data consistency, as malformed rules will cause the learning algorithms to fail.

`delete_rule(node_dict)`

In comparison with adding a rule to the rule data set, deleting a rule is relatively simple. As with the `add_rule` function, the `delete_rule` function requires that the `node_dict` dictionary should have node names as keys, with the associated values being the state of the node. Any nodes that are not specified in the `node_dict` dictionary are set to zero, and this final dictionary used for searching. Once the search finds a match, only this first search match is deleted as the rule data set may quite correctly have duplicate rules.

`rebuild_network(edges = true)`

The `rebuild_network` function is designed to rebuild the Bayesian network from the ground up using the managed node and edge dictionaries. Firstly, an OpenBayes BNet is created as this is the basic Bayesian network to which all the nodes and edges will be added. After this, all the BVertex instances that were created can be bound to the BNet instance by iterating through the managed node dictionary.

If the `edges` parameter is specified as true, we iterate through the managed edge dictionary converting the stored node names to BVertex instances. This extracts the node names into `node1` and `node2` from the stored tuple, which are in turn looked up in the managed node dictionary and the associated BVertex instance

references found. These BVertex instance references are then used in the creation of a tuple, containing the BVertex instance references for `node1` and `node2` in order. This effectively creates a directed network edge from `node1` to `node2`.

`calc_values()`

Once the rule data set has been populated, the `calc_values` function can be used to calculate the probabilities for each node. This is accomplished using an expectation-maximisation (EM) algorithm, which iterates over the nodes in the Bayesian network in an attempt to estimate node probabilities [6]. Inference is used to estimate for incomplete data, and iterations continue until either the node probabilities converge or the iteration limit is reached [8].

`calc_struct()`

In a system with unknown edges or structure, a stochastic expectation-maximisation (SEM) algorithm can be used instead of the standard expectation-maximisation algorithm to approximate the network structure. The SEM algorithm starts with an initial structure and then adds, deletes and reverses all possible edges keeping only the structure with the highest Bayesian information criterion (BIC) score [28]. For every structure change iteration, the EM algorithm is used to calculate the node probabilities. It is also recommended that the SEM algorithm be run multiple times with differing initial structures. All this heavy computation makes this algorithm very slow and not ideal for decision making, however it is more accurate than the approximation methods. [10, 19]

`calc_struct_approx()`

A faster way of computing structure than using the `calc_struct` function, is to use an approximation algorithm for calculating the node probabilities after each structure modification. An approximation can be made by estimating the node probabilities using only the parents of the nodes, without doing an inference on the Bayesian network. The EM algorithm will then only be used once, at the end of each iteration to correct the parameters for the next iteration. [8]

`get_node_value(node_name, mode = 0)`

The `get_node_value` function uses inference to obtain the exact probabilities associated with the node specified by `node_name`. This can be done either by using Join Tree [4], using mode zero, or Markov chain Monte Carlo (MCMC), using mode one, inference algorithms [15]. The resulting probabilities for the different edges are then returned to the calling function.

`add_evidence(evidence)`

The `add_evidence` function adds evidence to the Bayesian network so that the probabilities calculated are calculated with the given known node values.

4.4.3.6 `Learning_BN_Abstractor`

The `Learning_BN_Abstractor` class, inherits from the `Learning_BN` class, and abstracts away the inner details of a Bayesian network allowing simple decisions to be made quickly and easily. It uses each node in the Bayesian network as a boolean value and as data is added to the data set, the probabilities of the of the node will change proportionally.

`add_decision(node_name)`

The `add_decision` function will add a node to the Bayesian network using the `Learning_BN.add_node` function.

`add_data(node_name, data_value)`

The `add_data` function will create a new data set rule with the data value specified in `data_value` set for the node specified by `node_name`.

`get_value(node_name)`

The `get_value` function will return the probabilities associated with the node specified by `node_name`.

4.4.4 Fischertechnik Hardware Specific Classes

The obtain results, as shown in *Chapter 5*, from the framework, an implementation of hardware specific classes is required. This section makes use of the Fischertechnik robotics kit, as described in *Section 2.4*, and describes the hardware specific classes implemented and how they function. All the classes in this section make use of the Fischertechnik C++ libraries [7] and the Python-Ecke libraries [18] to interface with the Fischertechnik robot.

4.4.4.1 `FT_Robot`

The `FT_Robot` class is a child class of the `Panther_Robot` interface class and the `FishFace` Python-Ecke imported library class. This gives it the required compatibility with the

framework as well as all the functionality for interacting with the Fischertechnik robot provided by the `FishFace` class.

`__init__(conn_type)`
: The `__init__` function establishes a connection to the Fischertechnik robot, if possible, using the specified connection protocol in `conn_type`. The `conn_type` parameter should be of the form of a tuple containing first the name of the connection method to call, and second an array of parameters to pass to the function. The connection method name may be any valid `FishFace` connection method name.

`pause(time)`
: The `pause` method requires the `time` parameter to be specified in milliseconds. It uses the kernel method `GetTickCount` to obtain the current time in milliseconds, and then sleeps for the `PollInterval`, specified in the `FT_Constants` class, before checking again. Therefore this function is not exact and sleeps for at least `time` milliseconds meaning that it may exceed the required sleep time by just less than the specified `PollInterval`.

`__del__()`
: The `__del__` function is the class destructor and thus should never be called explicitly. The function disconnects the framework from the robot, thereby preventing any problems caused by unclosed connections.

4.4.4.2 `FT_Constants`

The `FT_Constants` class is a child class of the `Panther_Constants` and overrides all the required constants and identifiers from the `Panther_Constants` class. The overridden values are specific identifiers only useful to the Fischertechnik classes and will make little or no sense if used explicitly outside of these. The class also adds some constants not used by the framework at all. Some examples are the connection constants which are only used by the `FT_Robot.__init__` function to connect to the robot. Any other required constants can be added to the `FT_Constants` class or sub-class for any specific solution requirement.

4.4.4.3 `FT_Motor`

Since no special implementation is required for the Fischertechnik motors, the `FT_Motor` class is simply a renamed `Panther_Base_Motor` class with no extra implementation.

4.4.4.4 Fischertechnik Sensors

The following are the hardware specific classes for interaction with Fischertechnik sensors. Each class inherits from the specific Panther base class for that type of sensor, and overrides the corresponding function.

`FT_Distance_Sensor(Distance_Sensor)`

`FT_Touch_Sensor(Touch_Sensor)`

`FT_Light_Sensor(Light_Sensor)`

`FT_Color_Sensor(Color_Sensor)`

`FT_Lamp(Lamp_Sensor)`

4.5 File Structure

This section lists all the files contained in the implemented framework, as well as a list of the files upon which it is dependant. The classes contained within each file are also listed, with a brief description of why the particular classes are grouped together.

4.5.1 Panther.py

The Panther.py file contains all the Panther classes. These are all base classes for the framework, thus it should not be necessary to import from this file unless you are extending the framework as explained in *Section 4.6*. However, if this file is missing no part of the framework will function as all the other classes rely on the classes herein.

The classes contained in the Panther.py file are the following:

- `Panther_Robot`
- `Panther_Module`
- `Panther_Base_Sensor`
- `Panther_Base_Motor`

- `Panther_Constants`
- `NotPantherClassError`
- `ConstTypeError`

4.5.2 Movement.py

The Movement.py file contains all the framework classes that relate to movement of the robot. Hence, the classes included are the `Movement` and `Motor_Manager` classes. This file requires that only Panther.py be present to function correctly.

4.5.3 Sensors.py

The Sensors.py file contains all the framework classes that relate to using sensors connected to the robot. Thus, the classes contained are the `Sensor_Manager` class and all the framework sensor specific classes. This file requires only Panther.py to be present in order to function correctly.

4.5.4 Learning_Neural.py

This file contains the `Learning_Neural` base neural network interface class and is required to be present whenever a project requires neural network capabilities.

4.5.5 Learning_BN.py

The Learning_BN.py file contains only the `Learning_BN` class, and thus relies on only the Learning_Neural.py and Panther.py files containing the `Learning_Neural` and `Panther_Module` Classes, respectively. This file, along with its dependencies, can be used as a stand-alone module for creation and manipulation of a Bayesian network.

4.5.6 Fischertechnik.py

This file contains all the Fischertechnik hardware specific classes that directly interface and control the robot. These classes are the implementations of the interface classes contained in the Panther.py and Sensors.py files. Below is a list of the implemented hardware specific classes contained in this file.

- `FT_Robot`
- `FT_Constants`
- `FT_Motor`
- `FT_Distance_Sensor`
- `FT_Touch_Sensor`
- `FT_Light_Sensor`
- `FT_Color_Sensor`
- `FT_Lamp`

4.6 Extending The Framework

This section explains how to extend the implemented framework with additional modules and features as desired.

4.6.1 Additional Modules

In creating additional modules for the framework, the first step is to identify where the module fits into the framework and what features the module requires to function correctly. Generally additional modules can simply derive from the `Panther_Module` class, described in *Section 4.4.2.2*, as this will give the new module access to a history and backtracking systems. The `Panther_Module` class also provides basic management and tracking for the `Panther_Robot` and `Panther_Constants` instances to which it is attached.

If the extension deals with constants it is likely to be a child of the `Panther_Constants` class, described in *Section 4.4.2.3*. Given the multiple inheritance feature of Python[25], the additional module can derive from both the `Panther_Constants` and `Panther_Module` classes and integrate the functionality of these two classes. However, the disallowed list of the `Panther_Module` class would need to be modified to stop arbitrary calls from being captured.

4.6.2 Additional Features

An easy way to add additional features to the framework, is to integrate these into the existing framework code by modifying the files specified in *Section 4.5*. This has the benefit of giving all existing code these additional features, however modifying existing framework methods in such a way that their input or output is altered may cause existing programs to malfunction or produce unexpected results.

Rather than modifying existing code, it is suggested that additional methods be added that produce the required functionality. This means that all existing code will still function perfectly and any newer code written can use this additional functionality in addition to the original code. If a rewrite of a feature is required, it is suggested that a child class of the required class be created and this child class can then be used to access the rewritten feature as shown in *Figure 4.5 (a)*. If necessary, existing classes can be modified so that they derive from the new class although this is not recommended.

Preferably, if an existing class hierarchy needs to be changed, the feature rewritten class should be a sister class of the class with the feature to be rewritten. This feature rewritten class should only contain the code required for the rewritten feature, and should have the same parent as the class it is modifying as shown in *Figure 4.5 (b)*. Then this feature rewritten class can simply be added in front of the old class in the inheritance listing, thereby overwriting the old feature with the new one.

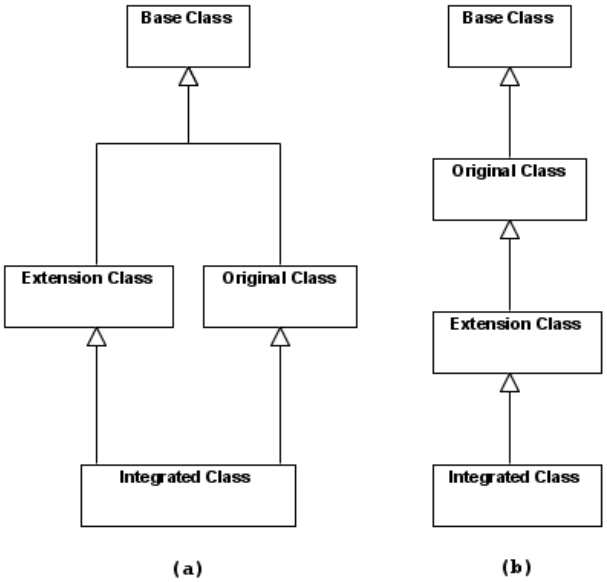


Figure 4.5: Framework Extensions Hierarchy Design

Chapter 5

Results From Using The Framework

5.1 Introduction

The goal of this chapter is to show how the implementation described in the previous chapter can be used to obtain results. The chapter starts by showing how to set up up a basic template to use the framework, and describing recommended practises when using the framework. The chapter continues by describing a simple test scenario for the framework, and how the framework was used to accomplish the required goal with a discussion of the expected and actual results. Thereafter, a more advanced test scenario for the framework is described. Once again we describe how the framework was used to accomplish the required goal and discuss the expected and actual results.

5.2 Template Setup

This section describes how to construct a basic template that makes use of the framework, as well as recommended practises for using the framework.

5.2.1 Basic Framework Template Setup

This basic template, as given in *Appendix A.1*, describes how to create a basic autonomous robotic project using the simple robot configuration containing left and right motors, a distance sensor and a blank Bayesian network. In addition to showing how to set up this

configuration correctly, the template uses more generalised concepts to make adaptation easy for cases where there are a number of sensors and motors.

5.2.1.1 Imports

The first step in creating a template is to import all the required modules, which is accomplished using the **`from`** notation. It is suggested that only that which is necessary is imported from any particular file using the from notation [25]. However, either import notations are acceptable with the **`import`** notation seemingly preferred in practise where a large number of imports are required from a single file.

To simplify working with the motors and sensors, as well as keeping the template generalised, the `Movement` and `Sensor_Manager` classes have been used to manage the motors and sensors attached to the robot. Thus these classes need to be imported from the Movement.py and Sensors.py files, respectively. As expected the majority of imports comes from the Fischertechnik.py file, where all the hardware specific classes reside. From this file, the `FT_Constants`, `FT_Robot` and `FT_Motor` imports will be required by almost any project, while the `FT_Distance_Sensor` import is required only for this template. Other hardware specific sensor class imports may also be required, given that the robot has the corresponding sensor attached. The last import required for this template is the `Learning_BN` class which resides in the Learning_BN.py file.

5.2.1.2 Initialisation

The initial step for setting up the template is to create an instance of the `FT_Constants` class, so this instance can be used to pass the required connection constants to the `FT_Robot` class constructor. The `FT_Robot` can be created using the required connection constant from the `FT_Constants` instance, and for this template the `ct_RF_distance` constant is used to connect to the robot using radio-frequency (RF) with distance sensor initialisation. Following this, the `Movement` and `Sensor_Manager` classes can be instantiated, and the two motors and distance sensor added. Finally the `Learning_BN` class can be instantiated to create the empty Bayesian network.

5.2.2 Robot Builder Template

The robot builder template, as shown in *Appendix A.2*, describes how to create a basic autonomous robotic project with the same configuration as the basic template, shown in *Section 5.2.1*. However instead of manually instantiating each class the `Panther_Robot .robot_build` function, as described in *Section 4.4.2.1*, in conjunction with the robot builder constants, as described in *Section 4.4.2.3*, have been used to create all the instances required by the configuration automatically. Only the `FT_Robot` class has been explicitly created, meaning that all the other class instances are created by the `robot_build` function. As shown by the template the dictionary returned by the `robot_build` function can be used as a control centre for the robot, as it contains all the classes required for controlling the robot.

Since the `Panther_Robot.robot_build` function merely requires a `Panther_Constants` subclass, a custom descendant of the `Panther_Constants` class can be passed to it, as shown in *Appendix A.2*. In the example, the class `Private_Constants`, a subclass of `Panther_Constants`, has been created and all the required constants are stored therein. This concept of storing all the constants required to build the configuration for a single robot hardware configuration can be used to create static robot hardware configuration classes. By creating these classes with a parent of the hardware specific constant class, either a custom solution specific constants class can be created or the robot hardware configuration class can simply be used to create all the instances required for the solution.

5.2.3 Recommended Practises

It is recommended to use the Robot Builder Template as well as hardware specific constant classes to create all the necessary instances to solve the problem at hand quickly and easily. Using this method, the framework setup requires only about two lines of code, excluding imports, thereby reducing the initialisation code required to start problem solving.

5.3 Simple Example

5.3.1 Scenario

A rather simple yet useful example of an autonomous robot is a contrast rover. A contrast rover is a robot that stays within a predefined area, with the area generally identified by

its colour. In this case the area used was a dark floor, while the area to avoid was white.

5.3.2 Setup

5.3.2.1 Robot Configuration

The components used in the robot configuration for this experiment were: two motors, one light sensor, one lamp, and a three wheel stationary turning robot design. The two motors were oriented on either side of the robot, thus giving the robot left and right motors. The light sensor and lamp were both mounted on the front of the robot, and pointed downward toward the ground and about one centimetre above it.

5.3.2.2 Algorithm

A fairly simplistic algorithm was used since this is intended merely as a basic example of giving the robot some autonomy. The algorithm dictates that while the light sensor detects the correct colour, i.e. the robot is in or on the edge of the defined area, the robot must move forward. As soon as the robot moves out of the defined area, the light sensor will pick up the incorrect colour and stop moving forward. Now the robot will do a simple search to locate the defined area by turning left, then right in incrementally larger time intervals until the correct area is once again located. Having located the area, the robot can once again begin moving forward. The time interval for the forward movement is kept relatively short so as to make sure the robot does not go far out of the defined region before detecting that it has left. Commented code for this algorithm is given in *Appendix B.1.*

5.3.3 Results

5.3.3.1 Expected Results

The expected motion of the robot is to stay within the defined area. If the robot leaves the area, it should relocate to the area and once again begin moving through the defined area.

5.3.3.2 Obtained Results

Since such a simplistic algorithm was used, after leaving the defined area the robot tended to hug a side of the area. This behaviour should in fact have been expected, since there is neither random movement, nor the ability to turn further into the area after the correct area was rediscovered. The worst case scenario for the robot, a sharp point in the area, was easily navigated. This is due to the robot remaining stationary while turning, meaning that the once the robot has detected it has left the area there is always some part of the area still underneath the robot.

5.3.4 Extensions

A relatively simple extension for the scenario is a line following robot. An additional light sensor is required in the robot configuration, and the two light sensors should be positioned slightly further apart than the line's width. The algorithm for this robot is extremely simple and can be summarised as: move forward until one of the light sensor inputs changes, then turn in the direction of that light sensor until both light sensors again register the same value.

5.4 Advanced Example

5.4.1 Scenario

Given the basic contrast rover example described in *Section 5.3*, this example is an extension thereof with added intelligence. The idea of this example is to extend the contrast rover to optimise its searching, so as to locate the designated area faster than is the case with the default algorithm.

5.4.2 Setup

5.4.2.1 Configuration

The same robot configuration is required by this example as that of the basic example discussed in *Section 5.3*.

5.4.2.2 Algorithm

The same basic algorithm is used in this example as in the basic example, with an enhanced searching algorithm to which artificial intelligence is more easily applied. The basic algorithm of search in one direction and if not found then search in the other direction has been kept, however this is a very biased approach as by always searching in one direction first that direction has an increased probability of being the direction turned to regain the designated area. To remove this bias an incremental search is used, i.e. a small search is made in one direction then the robot returns to its starting position and makes a small search in the opposite direction.

The optimisation of the robot's turning has been done by using a Bayesian network to make the simple decision between searching left or right first, based on observed previous results. This can be simply by using the Bayesian network as a expert system and every time the the area is lost and regained the direction turned to regain the designated area is stored in the Bayesian network dataset. Thus the robot will learn which way in general to turn to find the area fastest and turn in that direction with the Bayesian probability represented by the node. However, if we always search in the Bayesian network results direction then the search will become biased toward the direction in which the robot first rediscovered the designated area. Thus some element randomness is needed to balance this out, and in this case a ten percent probability of ignoring the Bayesian network decision has been introduced. Commented code for this example is shown in *Appendix B.2*.

5.4.3 Results

5.4.3.1 Expected Results

The expected outcome is that the robot will start turning one direction more often that the other depending on how it reaches the edge of the area. The direction chosen will be the direction it finds the designated area first when turning either left or right.

5.4.3.2 Obtained Results

Starting the robot within the area and pointing it out of the area at an angle, it was found that the robot was able to learn the optimal direction to turn from experience in the given area with the final probability being sixty percent chance of a left turn. In effect this intelligent autonomy only took less that twenty lines of code to produce.

5.4.4 Extensions

This example can easily be extended by adding nodes for the different time intervals of turning, such that the robot will search in the specified direction by turning for a specified number of milliseconds first. Furthermore, this could easily be extended to solving a maze where the maze passage ways are laid down as the designated area with a Bayesian network, or any other means, used to track the path decisions.

Chapter 6

Conclusion

6.1 Project Summary

The goal of this project is the creation of a programming framework for the easy addition of autonomy to robotics. This process involved first designing the extensible framework to be implemented. Thereafter an implementation of the designed framework was created in Python, and this implementation tested using examples with an increasing degree of functionality and intelligence.

6.2 Revisiting the Objectives

The primary objective of this project was investigating the feasibility of creating a programming framework for easily adding autonomy to robotics. As can be seen by the results in *Chapter 5*, this objective has been achieved. The simple examples presented show that it is possible to add autonomy to robotics with a basic understanding of the framework, and the inclusion of the minimum number of lines of code. Furthermore, the objective of making the programming framework easily adaptable and extensible has also been achieved. The simple extension to the contrast rover adding more intelligence is proof of this.

6.3 Extensions

This section describes possible extensions that can be made to the framework.

Additional Intelligence Algorithms

Additional artificial intelligence algorithms can be implemented and added into the Learning module. Some example algorithms are: Q-learning, Artificial Neural Networks, and Decision Tree Learning.

Additional Sensor Input Types

Additional input types can be added to the Sensors module. These sensors might include video cameras, pitch indicators, and GPS locators.

Extensions to current framework

There are some extensions to the framework which are not currently implemented, such as an invert_turn_motors.

Pre-built Robot Configurations

The provision of pre-built robot configurations to be used with the `Panther_Robot.build_robot` method will enable all the instances required to start programming the robot, to be created even more quickly.

Neural Network Action Manager

An addition to the neural network or Bayesian networks is to be able to associate an action with a node, and if that node has the highest probability when evaluated the action is executed. This will add far greater ability to the intelligence of the robot, by allowing learning to integrate directly with the robot's responses.

Absolute Directional System

Directional properties are in general the specific constants that dictate direction relative to a fixed geographical point. For example, a value of zero may denote the directional concept of forward, and a value of one the concept backward. A few examples of this type of directional constant are the following: `direction_forward`, `direction_backward`, `direction_left`, and `direction_right`.

Bibliography

[1] Apple Inc. Framework Programming Guide. Online, May 2009. Available from: http://developer.apple.com/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html.

[2] Ben-Gal, I. E. *Encyclopedia of Statistics in Quality and Reliability*. John Wiley & Sons, 2007.

[3] Bender, E. A. *Mathematical Methods in Artificial Intelligence*. IEEE Computer Society Press, 1996.

[4] Borsotto, M., Zhang, W., Kapanci, E., Pfeffer, A., and Crick, C. A Junction Tree Propagation Algorithm for Bayesian Networks with Second-Order Uncertainties. In *18th IEEE International Conference on Tools with Artificial Intelligence* (2006).

[5] Cprogramming.com. Cprogramming.com - Your Resource for C and C++ Programming. Online, June 2009. Available from: http://www.cprogramming.com/.

[6] Dellaert, F. The Expectation Maximization Algorithm. Tech. Rep. GIT-GVU-02-20, Georgia Institute of Technology, February 2002.

[7] fischertechnik. Fischertechnik Official Webpage. Online, May 2009. Available from: http://www.Fischertechnik.de/en/.

[8] Foundation, P. OpenBayes for Python. Online, October 2009. Available from: http://www.openbayes.org/.

[9] Howard, D. C++ vs Java vs Python vs Ruby. Online, October 2009. Available from: http://www.dmh2000.com/cjpr/.

[10] Jank, W. The EM Algorithm, Its Stochastic Implementation and Global Optimization: Some Challenges and Opportunities for OR. Tech. rep., University of Maryland, January 2006.

[11] Langtangen, H. P. *Python Scripting for Computational Science*. Springer, 2008.

[12] Lebeltel, O., Bessire, P., Diard, J., and Mazer, E. Bayesian Robot Programming. *Autonomous Robots 16* (2004), 49–79.

[13] Lewis, J. P., and Neumann, U. Performance of Java versus C++. Online, October 2009. Available from: `http://www.idiom.com/~zilla/Computer/javaCbenchmark.html`.

[14] Martelli, A. *Python in a Nutshell, (2nd Edition)*. O'Reilly, 2006.

[15] Milch, B., and Russell, S. General-Purpose MCMC Inference over Relational Structures. Tech. rep., University of California, 2004.

[16] Mitchell, T. M. *Machine Learning*. McGraw-Hill Book Company, 1997.

[17] Müller, U. ftComputing. Online, May 2009. Available from: `http://www.ftcomputing.de/index.htm`.

[18] Müller, U. Python-Ecke. Online, May 2009. Available from: `http://www.ulrich-mueller.de/pythonecke.htm`.

[19] Murphy, K. P. A Brief Introduction to Learning Bayesian Networks. Tech. rep., University of British Columbia, 2007. Available from: `http://people.cs.ubc.ca/~murphyk/Bayes/learn.html`.

[20] Na, Y., and Oh, S. Hybrid Control for Autonomous Mobile Robot Navigation Using Neural Network Based Behavior Modules and Environment Classification. *Autonomous Robots 15* (2003), 193–206.

[21] Prata, S. *C Primer Plus*, 5th ed. SAMS, 2005.

[22] py2exe. py2exe. Online, October 2009. Available from: `http://sourceforge.net/projects/py2exe/`.

[23] PyChecker. PyChecker: a python source code checking tool. Online, October 2009.

[24] Python.org. About Python. Online, May 2009. Available from: `http://www.python.org/`.

[25] Python.org. Python API Documentation. Online, September 2009. Available from: `http://docs.python.org/`.

[26] Python.org. Python FAQ. Online, May 2009. Available from: `http://www.python.org/doc/faq/`.

[27] Riehle, D. *Framework Design: A Role Modeling Approach*. PhD thesis, Swiss Federal Institute Of Technology Zurich, 2000.

[28] Rose F. Liu, R. S. Learning on Bayesian Networks. Tech. rep., MIT, December 2004.

[29] Sanner, M. F. Python: A Programming Language For Software Integration And Development. *J. Mol. Graphics Mod 17* (1999), 57–61.

[30] Schreuders, P. D. Engineering Modeling Using a Design Paradigm: A Graphical Programming-Based Example. *Journal of Technology Education 19*, 17 (Fall 2007), –.

[31] Soulie, J. C++ : A brief description. Online, May 2009. Available from: `http://www.cplusplus.com/info/description/`.

[32] swig.org. Simplified Wrapper and Interface Generator. Online, October 2009. Available from: `http://www.swig.org/`.

[33] Wiegerinck, W., Kappen, B., and Burgers, W. Bayesian Networks for Expert Systems, Theory and Practical Applications. *Interactive Collaborative Information Systems* (2009), 1–10.

[34] Yang, S. X., and Meng, M. Real-time Collision-free Path Planning of Robot Manipulators using Neural Network Approaches. *Autonomous Robots 9*, 1 (August 2000), 27–39.

[35] Zend Technologies Inc. Zend Framework Programmer's Reference Guide. Online, May 2009. Available From: `http://framework.zend.com/manual/en/`.

[36] Zope.org. Zope 3 API Documentation. Online, September 2009. Available From: `http://docs.zope.org/zope3/`.

Appendix A

Templates

A.1 Basic Template

```
from FischerTechnik import FT_Constants, FT_Robot, FT_Motor,
   FT_Distance_Sensor
from Movement import Movement
from Sensors import Sensor_Manager
from Learning_BN import Learning_BN

# create the Robot and Constants instances
FT_const = FT_Constants()
FT = FT_Robot(FT_Constants.ct_RF_distance)
# create and initialise the motor control
move = Movement(FT, FT_const)
move.add_motor(FT_Motor, 'left', 1, FT_const.orientation_left
   )
move.add_motor(FT_Motor, 'right', 2, FT_const.
   orientation_right)
# create and initialise the sensor control
FT_sensman = Sensor_Manager(FT, FT_const)
FT_sensman.add_sensor(FT_Distance_Sensor, "distance1", 1)
# create and initialise a learning class
FT_learn = Learning_BN(FT, FT_const)

# perform a basic test to check everything is working
```

```
FT_distance = FT_sensman.get_sensor("distance1")
print FT_distance.get_distance_value()
```

A.2 Robot Builder Template

```
from FischerTechnik import FT_Constants, FT_Robot, FT_Motor,
   FT_Distance_Sensor
from Movement import Movement
from Sensors import Sensor_Manager
from Learning_BN import Learning_BN

# create the custom solution class with prebuilt
   configurations
class Solution_Constants(FT_Constants):
  robotbuild_motor_manager = (Movement, FT_Robot,
     FT_Constants)
  robotbuild_motors = [(FT_Motor, "left", 1, FT_Constants.
     orientation_left), (FT_Motor, 'right', 2, FT_Constants.
     orientation_right)]
  robotbuild_sensor_manager = (Sensor_Manager, FT_Robot,
     FT_Constants)
  robotbuild_sensors = [(FT_Distance_Sensor, "distance1", 1)]
  robotbuild_learning = (Learning_BN, FT_Robot, FT_Constants)

# create and initialise a Robot instance
FT = FT_Robot(FT_Constants.ct_RF_distance)
# call the robot instance build_robot method to create all
   other instances required
ctrl = FT.build_robot(Solution_Constants)

# perform a basic test to check everything is working
print ctrl["Sensor_Manager"].get_sensor("distance1").
   get_distance_value()
ctrl["Motor_Manager"].move(ctrl["Constants"].motor_forward,
   ctrl["Constants"].speed_medium, 500)
```

Appendix B

Result Scripts

B.1 Contrast Rover

```
from FischerTechnik import FT_Constants, FT_Robot, FT_Motor,
   FT_Distance_Sensor, FT_Lamp, FT_Light_Sensor
from Movement import Movement
from Sensors import Sensor_Manager
from Learning_BN import Learning_BN

class Solution_Constants(FT_Constants):
  robotbuild_motor_manager = (Movement, FT_Robot,
     FT_Constants)
  robotbuild_motors = [(FT_Motor, "left", 1, FT_Constants.
     orientation_left), (FT_Motor, 'right', 2, FT_Constants.
     orientation_right)]
  robotbuild_sensor_manager = (Sensor_Manager, FT_Robot,
     FT_Constants)
  robotbuild_sensors = [(FT_Lamp, "lamp", 4), (
     FT_Light_Sensor, "light1", 1)]

FT = FT_Robot(FT_Constants.ct_RF_distance)
ctrl = FT.build_robot(Solution_Constants)
ctrl["Motor_Manager"].braking = 1
ctrl["Sensor_Manager"].get_sensor("lamp").set_on(3)
FT.pause(1000)
```

```
for a in range(0, 15):
  # while we are in the designated area, move forward
  while ctrl["Sensor_Manager"].get_sensor("light1").get_value
     () == 0:
    ctrl["Motor_Manager"].move(ctrl["Constants"].
       motor_forward, ctrl["Constants"].speed_slow, 100)
  t = 200
  # while we are off the designated area, search for it
  while ctrl["Sensor_Manager"].get_sensor("light1").get_value
     () != 0:
    # turn left first
    ctrl["Motor_Manager"].turnspin(ctrl["Constants"].left,
       ctrl["Constants"].speed_slow, t)
    # if the area is not found to the left, turn right
    if ctrl["Sensor_Manager"].get_sensor("light1").get_value
       () != 0:
      ctrl["Motor_Manager"].turnspin(ctrl["Constants"].right,
         ctrl["Constants"].speed_slow, t*2)
    # increase the search turn time inverval
    t += 300;
```

B.2 Intelligent Contrast Rover

```
from FischerTechnik import FT_Constants, FT_Robot, FT_Motor,
   FT_Distance_Sensor, FT_Lamp, FT_Light_Sensor
from Movement import Movement
from Sensors import Sensor_Manager
from Learning_BN import Learning_BN_Abstractor
import random
import sys

class Solution_Constants(FT_Constants):
  robotbuild_motor_manager = (Movement, FT_Robot,
     FT_Constants)
  robotbuild_motors = [(FT_Motor, "left", 1, FT_Constants.
```

```
    orientation_left), (FT_Motor, 'right', 2, FT_Constants.
    orientation_right)]
  robotbuild_sensor_manager = (Sensor_Manager, FT_Robot,
    FT_Constants)
  robotbuild_sensors = [(FT_Lamp, "lamp", 4), (
    FT_Light_Sensor, "light1", 1)]
  robotbuild_learning = (Learning_BN_Abstractor, FT_Robot,
    FT_Constants)

def turn(direction, tend, t, value):
  for i in range(0, tend):
    ctrl["Motor_Manager"].turnspin(direction, ctrl["Constants
      "].speed_slow, t)
    if ctrl["Sensor_Manager"].get_sensor("light1").get_value
      () == ctrl["Constants"].AREA:
      ctrl["Learning"].add_data("leftright", value)
      return 1
  ctrl["Motor_Manager"].turnspin(ctrl["Constants"].invert(
    direction), ctrl["Constants"].speed_slow, t*(tend-1))

FT = FT_Robot(FT_Constants.ct_RF_distance)
ctrl = FT.build_robot(Solution_Constants)
ctrl["Motor_Manager"].braking = 1
ctrl["Sensor_Manager"].get_sensor("lamp").set_on(3)
# add decision for left-right turning; 0 is left, 1 is right
ctrl["Learning"].add_decision("leftright")
FT.pause(1000)
ctrl["Constants"].AREA = 1

while 1:
  # while we are in the designated area, move forward
  while ctrl["Sensor_Manager"].get_sensor("light1").get_value
    () == ctrl["Constants"].AREA:
    ctrl["Motor_Manager"].move(ctrl["Constants"].
      motor_forward, ctrl["Constants"].speed_slow, 100)
  # while we are off the designated area, search for it
  t = 100 # turning time interval
```

```
tend = 3 # number of turns to make before trying the
   opposite direction

# To make sure that the robot will search first in the
   correct direction until the area is regained, we specify
    the first direction to turn before starting the loop
if random.random() > ctrl["Learning"].get_value("leftright"
   ) and random.random() > 0.1:
  turndir = ctrl["Constants"].left
  v = 0
else:
  turndir = ctrl["Constants"].right
  v = 1
while ctrl["Sensor_Manager"].get_sensor("light1").get_value
   () != ctrl["Constants"].AREA:
  # first turn in the direction chosen above
  # if the area is found, break
  if turn(turndir, tend, t, v): break
  # then turn in the opposite direction
  if turn(ctrl["Constants"].invert(turndir), tend, t, v):
     break

  tend += 2 # incase the robot does not turn far enough,
     increase the turning circle

  # once the robot is turn a half circle each way it is
     most likely it cannot re-locate the area
  if tend > 20:
    sys.exit()
```

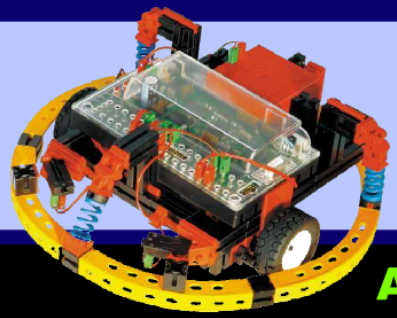
Appendix C

Project Poster

This appendix presents the project poster, which is one of the required project deliverables.

Autonomous Robotic Programming Framework

Reseacher: Leslie Luyt
Supervisor: Karen Bradshaw
Email: g06L0553@campus.ru.ac.za
Web: cs.ru.ac.za/research/g06L0553

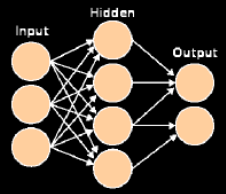


RHODES UNIVERSITY
Where leaders learn

Aims of Research

To create a self-contained programming framework to assist in the programming of autonomous robots with or without either generic or specific intelligence.

Bayesian Network



Robotics

fischertechnik

sponsored by
COMVERSE
STORTECH
For Your Information
amatole
Telkom
THRIP
TECHNOLOGY AND HUMAN RESOURCES FOR INDUSTRY PROGRAMME
Bright Ideas Projects 39
tellabs

Approach

Creation of an object-oriented framework that has modules allowing for interaction with the robot.

These modules will include functionaility such as: movement, sensor feedback, and intelligence.

The intelligence module will use bayesian and artificial neural networks to simulate learning, while the other modules will be specific to the fischertechnik robotic system.

Results

The intended result is a complete framework with completely independant modules that can be combined in various ways to create an intelligent robot.