The Development of a Control System
for an Autonomous Mobile Robot

A thesis

submitted in partial fulfilment

of the requirements for the Degree

of

Master of Science in

Physics and Electronic Engineering

at the

University of Waikato

by

Christopher Peter Lee-Johnson

THE UNIVERSITY OF

WAIKATO

Te Whare Wānanga o Waikato

2004

Abstract

The Mobile Autonomous Robotic Vehicle for Indoor Navigation (MARVIN) is the subject of a number of graduate research projects in the University of Waikato's Mechatronics Group. This thesis details the development of a control system for MARVIN's two-wheeled differential drive system that autonomously regulates its position, heading and velocity based on feedback from odometers, infrared rangefinders and tactile sensors. The completed control system operates in conjunction with a navigation system that was developed concurrently with this project. Together, they provide the most sophisticated autonomous behaviour currently implemented on a robot at the Mechatronics Group.

Acknowledgements

I wish to express my gratitude to the following people for their help during the course of this project.

Special thanks go to my supervisor, Dale Carnegie, who offered me this project, and who provided invaluable advice and motivated me to succeed, particularly during the final crunch. Thanks also to Scotty Forbes and Bruce Rhodes for their technical help.

My fellow inmates … er, ahem … graduate students … deserve no less appreciation for their advice, support and friendship over the past five years. Thank you (in no particular order) Andrew Payne, Lucas Sikking, Ashil Prakash, Craig Jensen and Adam Somerville.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 MARVIN

MARVIN is the flagship of an expanding fleet of large-scale autonomous guided vehicles developed by the Mechatronics Group of the Department of Physics and Electronic Engineering at the University of Waikato. MARVIN was named after the paranoid android from Douglas Adams's "Hitchhikers Guide to the Galaxy" novels (Figure 1.1), and in the grand academic tradition, a suitably descriptive acronym was chosen to match the name (Mobile Autonomous Robotic Vehicle for Indoor Navigation).

Figure 1.1: Marvin the Paranoid Android

The original long-term objective of the MARVIN project was to develop an autonomous mobile security device that would patrol the corridors of the university's science block, detecting and recording the activities of intruders, and recharging itself when necessary. However, this goal has since been revised, and it is now less specific about the intended application. Although security applications have not been ruled out, MARVIN could just as easily be used for tasks such as internal mail delivery. In recent years the project's focus has also shifted towards public relations and human-machine interaction.

Daniel Loughnane, a former graduate mechatronics student, began development on MARVIN in 1999. By early 2001, the basic mechanical structure and electronics had been implemented. The robot possessed a rudimentary control system, implemented on a Phillips 87C552 microcontroller, that regulated wheel speeds using odometers, and utilised infrared rangefinders for object avoidance [Loughnane, 2001].

Former graduate student Shaun Hurd concurrently developed a custom laser rangefinding system for MARVIN. This system utilised a rotating laser and CCD camera to detect objects at a range of up to 10 m over a 360° field of view. Images were acquired using a PC image capture card, and an image-processing algorithm was developed to extrapolate distances from the recorded images. The laser rangefinder was mounted to MARVIN's upper chassis, as shown in Figure 1.2. Although successful, this device was never integrated into MARVIN's control system [Hurd, 2001].

Figure 1.2: MARVIN and the Laser Rangefinder

Development slowed during 2001, because no full-time graduate students were assigned to the project at the time. Nevertheless, a number of revisions were implemented. The microcontroller was replaced with a PC and a data acquisition (DAQ) card, but the control software was not ported over to the new platform. Instead, a LabVIEW program was developed to allow remote driving capabilities using a wireless LAN connection. This was successfully demonstrated in the Osborne Lectures, a tour of secondary schools throughout New Zealand. Unfortunately the motor drivers, rangefinders and one of the odometers were subsequently removed for use in other projects, so that MARVIN was no longer operational at the onset of this thesis project.

1.2 Projects

Four separate thesis projects relating to MARVIN, including this one, are being conducted in parallel. They interrelate in the loosely defined hierarchy shown in Figure 1.3.

```
┌─────────────────────────────────┐
│      Robot Humanisation          │
└─────────────────────────────────┘
                ↕
┌─────────────────────────────────┐
│   Task Planning and Navigation   │
└─────────────────────────────────┘
                ↕
┌─────────────────────────────────┐
│      Localisation and Control    │
└─────────────────────────────────┘
                ↕
┌─────────────────────────────────┐
│      Generic Motor Drivers       │
└─────────────────────────────────┘
```
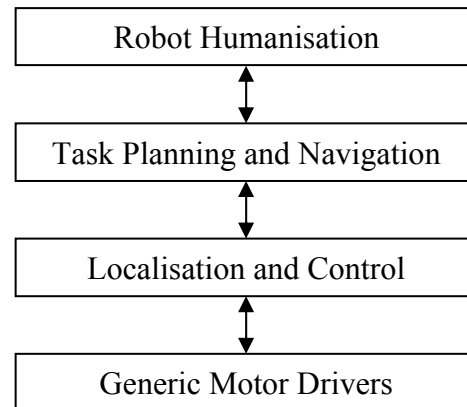
Figure 1.3: Project Hierarchy

1.2.1 Robot Humanisation

The original design for MARVIN's outer chassis bore an unfortunate resemblance to the Daleks from "Dr Who". A modern robot should not look like tacky 1960's science fiction, so graduate mechatronics student Ashil Prakash is working with Robotechnology Ltd. [http://www.robotechnology.co.nz] to design a chassis that is more in line with 21st century expectations (Figure 1.4). In addition to its aesthetic improvements, the new chassis will allow MARVIN to exhibit humanlike behaviour through motion of the head and torso [Prakash & Carnegie, 2003].

The second aspect of the robot humanisation project is a human-machine interface. This is primarily a voice interface that will utilise commercial speech recognition software to interpret spoken commands. Due the complexities of identifying unique voice patterns, this system cannot recognise individuals – an essential requirement for security applications. Instead, individual access will be provided using a Cardax swipe-card system [http://www.cardax.com].

Daleks   **Original Chassis**   **New Chassis**
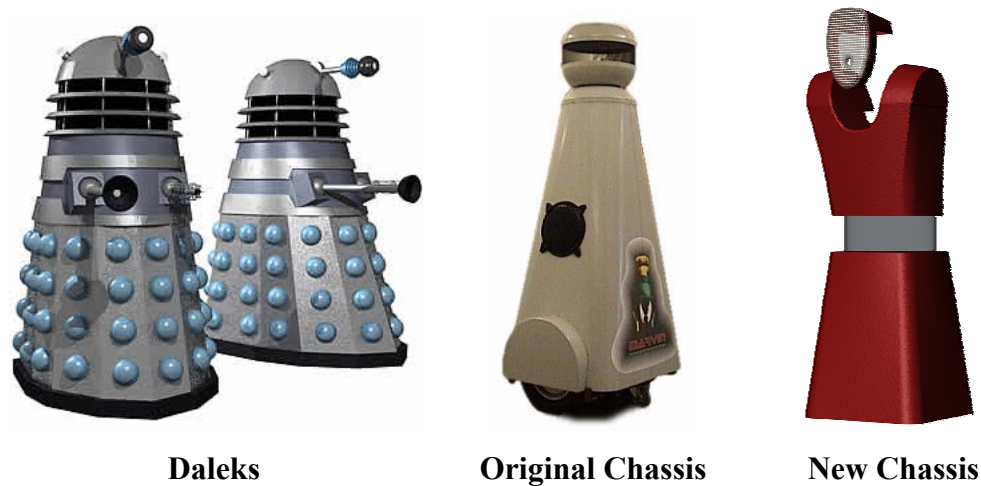
Figure 1.4: MARVIN vs. the Daleks

1.2.2 Task Planning and Navigation

The task planning and navigation system being designed by graduate student Lucas Sikking is the Mechatronics Group's first attempt at achieving autonomy in a large-scale mobile robot. Voice commands indicating target destinations must be translated into coordinates on an internal map. The navigation system will plot a course for MARVIN to travel in order to reach the target coordinates, and resolve it into a sequence of instructions that are passed to the control system (Section 1.2.3). If an unmapped obstacle is encountered, the map will be updated and a new course will be plotted around the obstacle. Infrared beacons are to be placed at strategic locations around the operating environment, providing active landmarks that the navigation system can use to correct any cumulative odometry errors [Sikking & Carnegie, 2003].

1.2.3 Localisation and Control

Although a simple control system was developed for MARVIN's embedded controller, it is not implemented on the new PC hardware. The limitations of the original control system, coupled with the fact that significant alterations will be made to the sensors and motor drivers, mean that converting it to the new platform is not a

viable option. This thesis describes the development of a new control system for MARVIN that will execute motion instructions delivered by the navigation system, and return sensor data and localisation information.

Instructions are translated into velocity profiles and a target trajectory. The control system must ensure that MARVIN's wheels follow the intended velocity profiles as closely as possible. Data from multiple sensors must be combined appropriately in order to produce an accurate representation of MARVIN's position and orientation, which the control system will use to track MARVIN's motion along the target trajectory. If it drifts off course, the wheel velocities must be adjusted accordingly [Lee-Johnson & Carnegie, 2003].

1.2.4 Generic Motor Drivers

One of the main difficulties encountered by the Mechatronics Group has been the unreliability of motor driver circuits, particularly when large loads are involved. Craig Jensen, a mechatronics graduate student, will design generic H-bridge motor driver circuits that can be applied to any robot in the Mechatronics Group's fleet, including MARVIN. This will greatly accelerate the initial stages of development, allowing students to focus on high-level design.

Since all the large-scale robots include a PC, the motor drivers will utilise a standard DB9 serial interface (which can also be adapted to USB, if necessary). Software will be developed that transmits power levels to the motor drivers, and receives odometer data. Eventually the software will be utilised by MARVIN's control system, replacing the system-specific hardware interface developed in this thesis [Jensen & Carnegie, 2003].

1.3 Operating Environment

MARVIN is intended to operate primarily within the first floor corridors of C block at the University of Waikato (overhead view given in Figure 1.5, photographs given in Figure 1.6). This is of greatest importance to the navigation system, which requires a predefined static map of the environment.
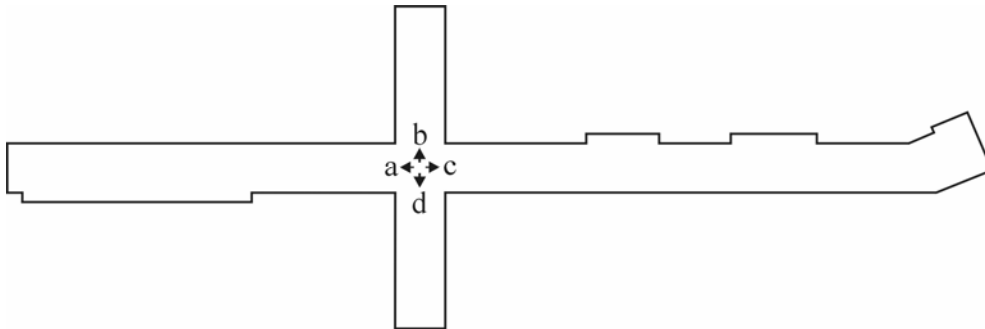
Figure 1.5: Overhead View of C Block Corridor

(a)

(b)

(c)

(d)

Figure 1.6: C Block Corridor Viewed from Intersection

Although it will receive information from the navigation system, which will operate only in pre-mapped environments, the control system itself will be less location-specific, and it should function at full capacity inside any environment with parallel walls that are within range of the rangefinders. In an environment where the walls are too distant or obscured by obstacles, the control system will still operate, but it must rely exclusively on dead reckoning for localisation, and consequently it will become more susceptible to cumulative error [Borensten & Feng, 1996].

1.4 Thesis Objectives

The primary objectives of this thesis project are as follows:

- Purchase or design new optical encoders (odometers), infrared rangefinders, tactile sensors, beacon receivers and motor drivers.
- Mount sensors and actuators to the chassis and interface them to the DAQ card.
- Develop LabVIEW software that uses the DAQ card to communicate with the sensors and actuators.
- Establish a means to control LabVIEW from MATLAB.
- Acquire sensor data in MATLAB.
- Obtain velocity, position and orientation information from odometer data.
- Obtain obstacle distances, position and orientation information from rangefinder data.
- Combine localisation information from odometers and rangefinders to produce a single representation of MARVIN's position and orientation.
- Obtain velocity profiles and an overall trajectory of motion for a given instruction
- Utilise localisation information to maintain MARVIN's trajectory.
- Develop a PID control system to maintain wheel velocities at the appropriate levels.
- Incorporate tactile sensor data and obstacle distances in a collision avoidance system so that MARVIN can react to a dynamic environment.

1.5 Project Planning

Figure 1.7 outlines the development schedule for this project.

Establish communication protocol between control system and navigation system.

↓

Simulate sensors and actuators.

↓

Obtain localisation information from simulated odometers and rangefinders.

↓

Obtain wheel velocity profiles that implement given motion instructions.

↓

Control MARVIN's simulated motion along intended trajectory.

↓

Install and interface real sensors and actuators.

↓

Refine control system for operation in the real world.

↓

Test control system in conjunction with navigation system.

↓

Improve system performance.

Figure 1.7: Flowchart of Project Development Process

1.6 Chapter Summary

This thesis is divided into the following chapters:

Chapter 2 – MARVIN's hardware is discussed, and the alterations and additions implemented during this project are described.

Chapter 3 – The various software interfaces utilised by the control software are explained.

Chapter 4 – The process of obtaining localisation information from sensor data and combining data from multiple sensors in a useful manner is discussed.

Chapter 5 – An algorithm that controls MARVIN's speed and trajectory is described.

Chapter 6 – Test results of MARVIN's motion under various conditions are presented.

Chapter 7 – Conclusions are given along with a performance evaluation and suggestions for future development.

2 Hardware

2.1 Overview

Although MARVIN had been operational in the past, it was in a non-working state at the onset of this project. The motor driver PCBs had been salvaged for use in other devices. One of the odometers was missing, and neither had been interfaced to the PC. Many of the components, including the motherboard, hard disk, DAQ connector module, power supply and Uninterruptible Power Supply (UPS), were mounted in a temporary manner that was insufficiently robust to withstand the forces that would be encountered during real-world operation. Figure 2.1 gives an outline of the unfavourable aspects of MARVIN's original hardware.
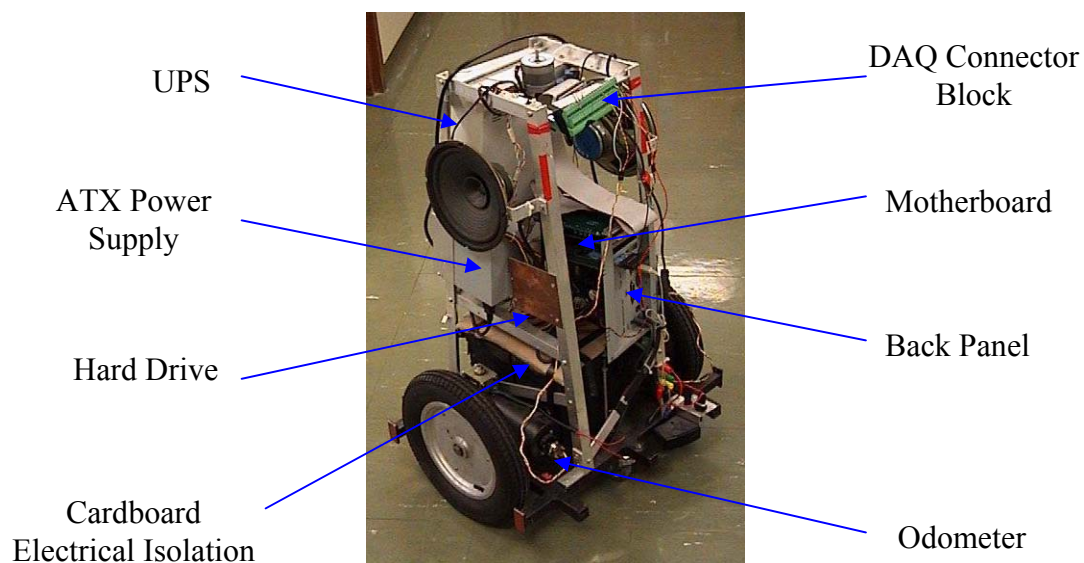
Figure 2.1: MARVIN 2000

Given MARVIN's initial state, the decision was made to do a significant hardware overhaul. The wheels, chassis, motors and batteries remain largely untouched from the original design, but the power supply, PC, DAQ card, wireless LAN module, odometers, rangefinders, tactile sensors, beacon receivers, PCB platforms, switches, cabling, motor drivers and microcontroller have been replaced or added during the course of this project. MARVIN's new hardware is shown in Figure 2.2.
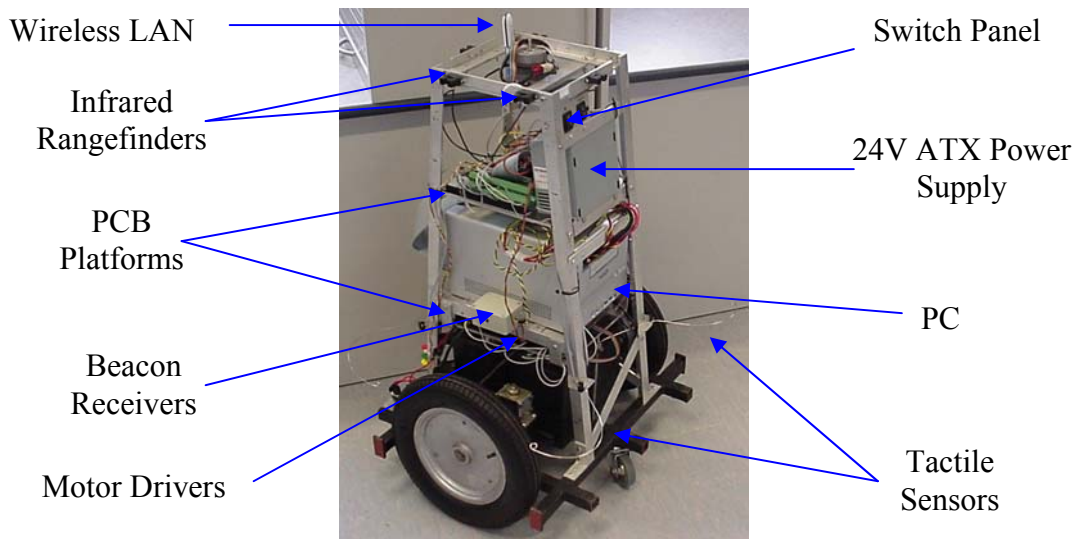
Figure 2.2: MARVIN XP

2.2 Mechanical Details

MARVIN's frame is 0.777 m high, 0.583 m wide and 0.515 m long. The base of the frame is constructed from 25 mm × 25 mm steel tubing, welded together for strength. The upper section consists of aluminium struts that are riveted or screwed together for easy modifiability.

In order to lower the centre of gravity and improve stability, the heaviest components, the motors and batteries, are mounted at MARVIN's base. The PC is mounted near the centre for accessibility reasons. The ATX power supply cannot fit inside the PC, so it is attached to two aluminium beams on MARVIN's side.

Two Perspex platforms are attached above and below the PC, providing non-conductive surfaces on which to mount the various PCBs. These circuits are attached to the platforms using PCB guides, with at least one end left open so that they can be removed for repair if necessary. The lower platform houses the motor drivers and microcontroller, which ideally should be as close as possible to the motors. DAQ connector blocks and sensor-related PCBs are mounted on the upper platform.

Locomotion is provided by two wheels in the standard wheelchair configuration, supported by castors at the front and rear. MARVIN's linear velocity and heading are controlled by varying the angular velocity of each wheel. This yields tighter turning circles and reduced wheel slippage in comparison to other possible arrangements such as tricycle or quad wheel configurations [Loughnane, 2001]. The wheels have a radius of 0.165 m, and the distance between the centre of each wheel is 0.508 m. The tyre pressure of each wheel is 303 kPa (44 lb/in^2).

2.3 Power Source

MARVIN is powered by two 12 V flooded lead-acid batteries in series (resulting in a total of 24 V). These batteries have a Reserve Capacity (RC) of 55 minutes, and a Cold Cranking Amperage (CCA) of 310 A. This corresponds to approximately 23 Amp Hours of useful operation. However, since they are not deep cycle batteries, damage will result if they are repeatedly allowed to run flat.

The motor drivers are powered directly from the 24 V battery terminals. The PC, microcontroller and sensors require a range of voltages (generally 5 V or 12 V) which are provided by an ATX power supply. The power supply was originally connected to a 240 V AC mains-equivalent signal produced by a UPS. This approach was relatively inefficient, as it involved a conversion from a low DC voltage to a high AC voltage, and back again. Another limitation to consider was the significant size and weight of the UPS.

Consequently, the standard ATX power supply was replaced with a 24 V ATX power supply, the ACE-828C from ICP Electronics (shown in Figure 2.3). This supply's output characteristics are virtually identical to those of a standard ATX supply, but it is powered from 18 V to 32 V DC, rather than mains, so it can be connected directly to the battery terminals, eliminating the need for a UPS. The ACE-828C is rated up to 250 W, which is more than adequate for the PC and the limited number of peripherals that are driven from it.
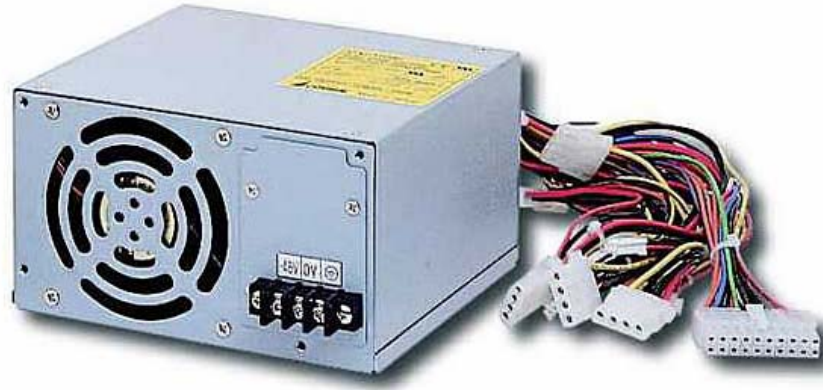
Figure 2.3: ACE-828C 24 V ATX Power Supply

The ACE-828C, like most modern ATX power supplies, goes into a "soft" shutdown mode when the PC is turned off. Due to the power supply's imperfect efficiency, it does draw a small current in this state, which can drain the batteries over time. It is therefore necessary to physically disconnect the power supply from the batteries after the PC is shut down. This is accomplished using a Double Pole Single Throw (DPST) switch mounted on an aluminium panel in an accessible location near the top of MARVIN, as shown in Figure 2.4.

A separate DPST switch is used for the motor drivers (mounted in the same location for aesthetic and accessibility reasons), since it is often necessary to disable the motors while the PC and sensors are running. An LED is mounted to the panel that indicates whether the microcontroller (Section 2.6.2) is receiving valid instructions from the PC.

Figure 2.4: Power Switch Panel

2.4 PC Hardware

MARVIN executes its high-level software on a standard PC platform rather than the embedded controllers favoured by most of the Mechatronics Group's smaller mobile robots (although it does still use a microcontroller to drive the motors, as described in Section 2.6.2). PCs have numerous advantages over embedded controllers, including:

- Processing speed increase of several orders of magnitude.
- Improved code portability.
- Wider selection of development tools.
- Large variety of alternative hardware interfaces.
- Cheap wireless communication options.
- Less hardware design necessary.

2.4.1 PC

The original PC used for MARVIN was a 466 MHz Celeron with 128 MB RAM, and a 6 GB hard disk. A standard PC case is too large to fit inside MARVIN's chassis, so most of the case had been trimmed away, leaving just the motherboard and back panel, which were mounted to MARVIN's chassis on a single corner.

This PC was barely adequate for the complex calculations that would be necessary for this and other projects, so it was upgraded. The Shuttle xPC, a Small Form-Factor (SSF) computer, was selected rather than a standard PC. This has the advantage that no case modifications are necessary for it to fit inside MARVIN's chassis, and it can remain fully enclosed, providing better protection and isolation. Unlike other small-sized computers such as notebooks and PDAs, SFF computers remain competitive with standard PCs in terms of price and performance.

The only significant drawback is that the Shuttle xPC has just one PCI slot (currently occupied by the data acquisition card), limiting the potential for future expansion. This is of particular significance to Shaun Hurd's custom-designed laser rangefinding

system (Section 1.1), which will be added to MARVIN at a later date. The device currently requires a PCI video capture card to operate, so it will be necessary to find an alternative means of video capture, or replace the PC.

The specifications for the Shuttle xPC (shown in Figure 2.5) are as follows:

CPU:	Athlon XP 2000+ (1.67 GHz)
RAM:	512 MB PC2700 DDR (333 MHz)
Motherboard chipset:	nVidia nForce 2 / MCP-T
Expansion slots:	1 × PCI
	1 × AGP
General purpose I/O Ports:	4 × USB 2.0
	2 × IEEE 1394
	1 × Serial DB9
Hard Disk:	10 GB, 5400 rpm
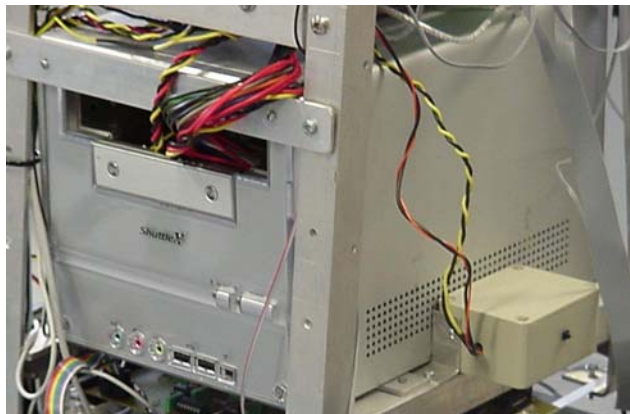
Figure 2.5: Shuttle xPC

2.4.2 Data Acquisition (DAQ) Card

The software originally communicated with the sensors and actuators using a National Instruments Lab-PC+ DAQ card (Figure 2.6) plugged into the PC's ISA slot. Featuring eight analogue inputs, two analogue outputs, 24 digital I/O lines and three configurable timers/counters, the Lab-PC+ is adequate for the purposes of this project.

Unfortunately, modern motherboards are no longer manufactured with ISA slots, so this card is unsuitable for use in the new PC.

Figure 2.6: Lab-PC+ Data Acquisition Card

The replacement card is the 6025E (Figure 2.7), also from National Instruments. Chosen for its similarity to the Lab-PC+, the 6025E has the following specifications:

Analogue Inputs

Number of channels:	16 single-ended or 8 differential
Resolution:	12 bits
Maximum sampling rate:	200 kS/s
ADC Type:	Successive approximation

Analogue Outputs

Number of channels:	2
Resolution:	12 bits

Digital I/O

Number of channels:	32

Counters/Timers

Number of channels:	2
Resolution:	24 bits
Maximum source frequency:	20 MHz
Type:	Up/down counters

Figure 2.7: 6025E Data Acquisition Card

A block diagram representing the hardware functions of the 6025E is given in Figure 2.8. Unlike the Lab-PC+, the 6025E is a jumperless card, and all of the configuration settings are software-selectable.
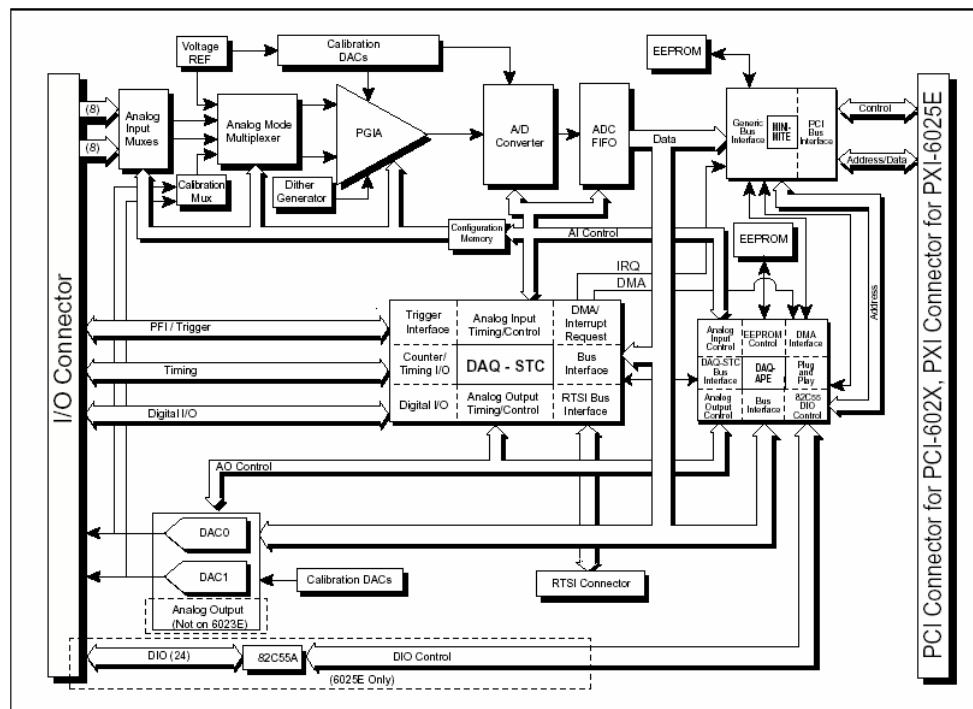
Figure 2.8: 6025E Block Diagram

The analogue inputs in the 6025E DAQ card utilise a Programmable Gain Instrumentation Amplifier (PGIA), which can be configured in Referenced Single-Ended (RSE), Non-Referenced Single-Ended (NRSE), or Differential (DIFF) modes of operation. While in RSE mode, the DAQ card ties the PGIA's negative inputs to the Analogue Input Ground (AIGND) terminal. In NRSE mode, the PGIA's negative inputs are tied to Analogue Input Sense (AISENSE). DIFF mode configures the DAQ

card so that adjacent analogue inputs are attached to the positive and negative inputs of the PGIA.

Most new National Instruments DAQ cards only provide 8 digital I/O lines. However, in order to maintain compatibility with legacy cards, the 6025E uses an 82C55A Programmable Peripheral Interface (PPI) to provide 24 additional digital I/O lines, divided into three 8-bit ports, which can be configured to perform the same functions as the Lab-PC+ digital I/Os. Four modes of operation are available: Basic I/O, Strobed Input, Strobed Output and Bi-directional. These modes configure various automatic handshaking signals on Port C.

The various digital and analogue lines are accessible via two 50-pin I/O connector blocks attached to the back of the card with a ribbon cable.

2.4.3 Wireless LAN

Until Ashil Prakash's voice recognition interface is implemented on MARVIN, the software is operated remotely from a notebook PC, which communicates with MARVIN's PC over a wireless LAN connection. MARVIN's PC is accessed using the standard Remote Desktop Connection (or Windows Terminal Service) tool that ships with Windows XP. This essentially allows the notebook to become MARVIN's keyboard, mouse, monitor and speakers.

There are insufficient PCI slots available for an internal wireless LAN card, so a USB module – the ZyAIR B-220 (Figure 2.9) – is used instead. The B-220 has the following specifications:

Media Access Protocol:	IEEE 802.11b	
Data Rate:	11 Mbps / 5.5 Mbps / 2 Mbps / 1 Mbps	
Coverage Area:	Indoor:	50 m @ 11 Mbps
		80 m @ 5.5 Mbps or lower
	Outdoor:	150 m @ 11 Mbps
		300 m @ 5.5 Mbps or lower
Frequency:	2.4 ~ 2.835 GHz (Industrial Scientific Medical Band)	
Output Power:	17 dBm (typical)	
Receiver Sensitivity:	-82 dBm @ 11 Mbps	
Bit Error Rate:	10^{-5} @ -82 dBm	
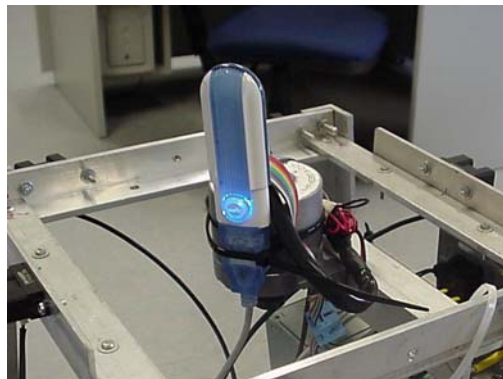
Figure 2.9: ZyAIR B-220 Wireless LAN Module

2.5 Sensors

MARVIN is equipped with a range of sensors – odometers, rangefinders, tactile sensors and beacon receivers – that the control and navigation software uses for tasks such as localisation, velocity control and obstacle avoidance. A detailed analysis of their usage in software is given in Chapter 4.

2.5.1 Odometers

MARVIN utilises the HEDS-5500 optical encoder module (Figure 2.10) for wheel position and velocity measurements. In an optical encoder, electrical pulses are generated from light that passes through holes on the perimeter of a circular disk (the code wheel) onto an optical receiver. The change in wheel position is measured by counting these pulses. The HEDS-5500 module includes the HEDS-9100 encoder and HEDS-5120 code wheel – the same components that were used on MARVIN prior to this project – but they are enclosed in a single package, simplifying the design and providing additional protection.

Figure 2.10: HEDS-5500 Optical Encoder Module

The HEDS-5120 code wheel produces 500 pulses per revolution. Since the odometers are mounted on the motors, the gearing ratio results in a resolution of 25780 pulses per wheel revolution. Given the wheel circumference of 1.0367 m, this corresponds to 24867 pulses per metre.

Each encoder includes two output channels that are 90° out of phase, providing a means to determine direction. MARVIN only utilises a single channel, because directional changes are slow enough that they can be more easily detected using software techniques (Section 4.1.1). The encoder outputs are connected to the counter source pins (GPCTR0_SOURCE and GPCTR1_SOURCE) on the DAQ card.

2.5.2 Rangefinders

MARVIN utilises the Sharp GP2Y0A02YK infrared distance-measuring sensor (Figure 2.11) to detect nearby objects. This device has a measurement range of 0.2 m to 1.5 m, which is adequate for location sensing and object detection in the intended corridor and laboratory environment.
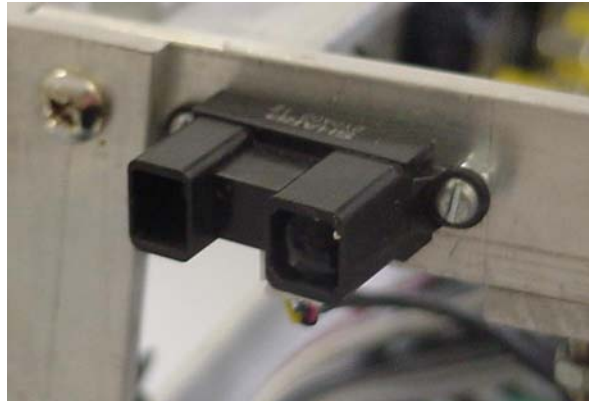
Figure 2.11: GP2Y0A02YK Infrared Distance-Measuring Sensor

The GP2Y0A02YK calculates distances using a scheme based on triangulation (Figure 2.12). Light from an infrared emitter is reflected off an object onto a Charge Coupled Device (CCD) detector. An analogue voltage is generated from the position of the detected light along the CCD array. Since the detector is positioned at a known location and orientation with respect the emitter, the range of an object can be calculated directly from this voltage. The primary advantage that this method has over other schemes such as intensity measurement is that the colour and reflectivity of an object has little effect on the measured distance.
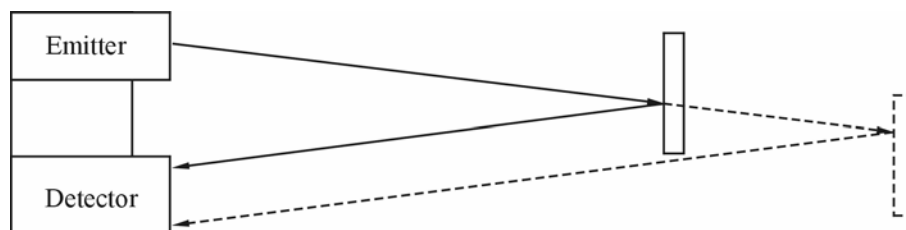
Figure 2.12: Triangulation with the GP2Y0A02YK

The voltage-distance relationship (Figure 2.13) reveals that distances less than 0.15 m will result in misleading measurements, so the rangefinders are mounted far enough from MARVIN's edge that this "dead region" is never encountered. Six rangefinders are mounted at the top of the chassis – one on the front, one on the back, and two on each side. The side rangefinders face about 15° away from each other, reducing optical crosstalk.
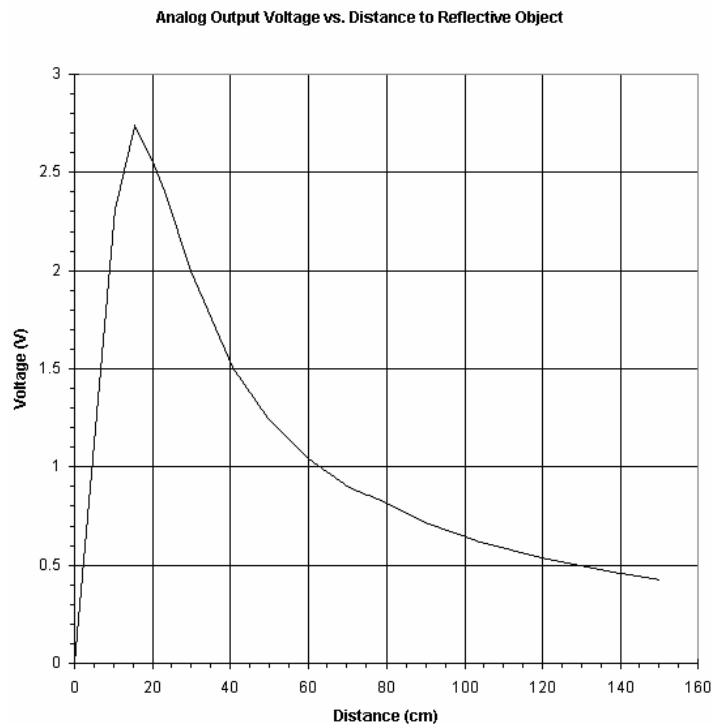
Figure 2.13: GP2Y0A02YK Voltage-Distance Relationship

In order to reduce noise on their supply rails, the rangefinders are driven from a separate supply rail produced by a 7805 linear regulator. The regulator provides a stable 5 V from a 12 V input. The rangefinder outputs are connected to DAQ card's the analogue inputs (ACH0-ACH5).

2.5.3 Tactile Sensors

Four tactile sensors are mounted near MARVIN's base – one at each of the four corners. Some tactile sensors are position-dependant, but MARVIN currently uses simple binary sensors. These are a temporary safety measure that will be replaced with a more robust design once Ashil Prakash's outer chassis is added.

Each sensor consists of a wire whisker that presses a SPDT switch when it flexes due to contact with an object, as shown in Figure 2.14. The switches are connected to four digital I/O ports on the DAQ card (DIO0-DIO3). When a switch is pushed (i.e. a collision with an object has occurred), the corresponding port is pulled high through an internal pull-up resistor on the DAQ card. Otherwise it is pulled low. The software can use this signal to implement an emergency stop procedure that executes in the event of a collision.
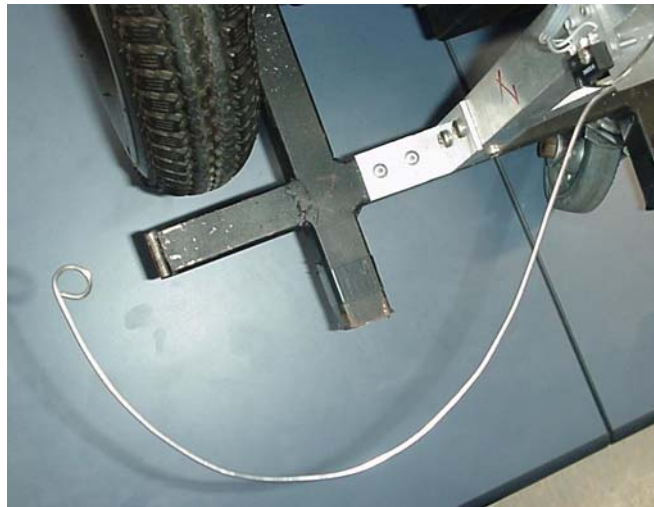
Figure 2.14: Tactile Sensor

2.5.4 Beacon Receivers

Two Kemo B062E infrared receivers are utilised in conjunction with modified versions of the B062S emitter (Figure 2.15). These emitters function as beacons (artificial landmarks) for Lucas Sikking's navigation algorithm, positioned at known locations on the corridor walls [Sikking & Carnegie, 2003]. The receivers are

mounted on MARVIN's sides, far enough below the rangefinders to minimise interference. They provide signals indicating when MARVIN passes an emitter, so that the navigation system can correct any cumulative odometry errors that have arisen.
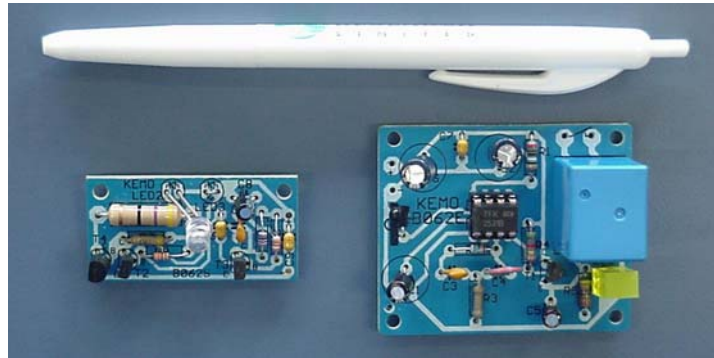
Figure 2.15: B062E Infrared Receivers and B062S Emitters

The receiver circuits consist of a photodiode and filtering IC that switches a relay when it receives a 14 kHz modulated infrared signal from the emitter. The relay is connected to the digital I/O port on the DAQ card in the same manner as the contact switches, so that the digital line is pulled high when MARVIN is in the path of a beacon. Receiver and emitter schematics are given in Appendix A.1 and Appendix A.2 respectively.

2.6 Actuators

MARVIN's wheels are driven independently by two 24 V DC permanent magnet brush motors (salvaged from an electric wheelchair), which are controlled using H-bridge motor driver PCBs designed by graduate Mechatronics student Andrew Payne. Pulse Width Modulated (PWM) inputs for the motor drivers are in turn provided by a separate 8051 microcontroller PCB, also designed by Andrew Payne.

The motor drivers were originally developed for Itchy and Scratchy; a pair of identical robots designed to perform cooperative tasks [Payne & Carnegie, 2003]. Since Itchy and Scratchy possess similar motor characteristics to MARVIN, the same designs can

be utilised for this project with no significant modifications. Craig Jensen's generic motor drivers may replace them at a later date, depending on comparative test results.

2.6.1 Motor Drivers

The motor driver PCBs (Figure 2.16, schematic shown in Appendix A.3) control the current supplied to the motors (and therefore the speed of rotation) by varying the duty cycle of a PWM signal. As long as the PWM signal's switching speed is much greater than the motor's time constant, the motor responds in approximately the same manner that it would react to a DC voltage that is proportional to the PWM signal's duty cycle.
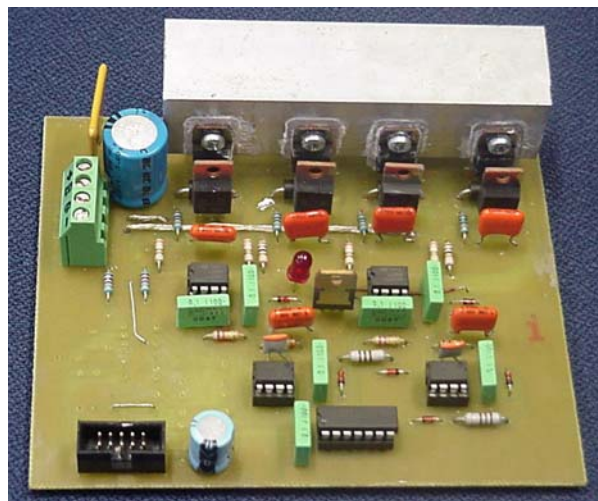
Figure 2.16: Motor Driver PCB

In order to provide the high current signals necessary to drive large motors, the motor drivers utilise the H-bridge circuit shown in Figure 2.17. Two diagonal MOSFETs are switched from the PWM signal, while the other two remain in the off state. Motor direction is controlled by selecting which transistor pair to switch [Payne & Carnegie, 2003].
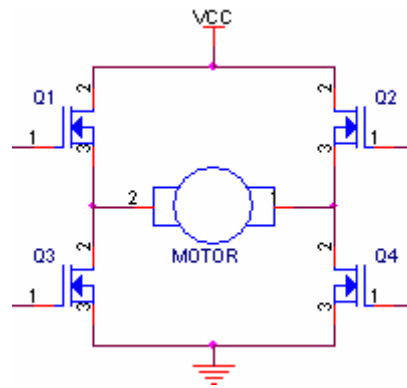
Figure 2.17: H-Bridge Circuit

2.6.2 Microcontroller

The 6025E DAQ card can generate PWM signals directly from its two onboard counters. Unfortunately, the counters are also used to count odometer pulses, and there are too few available for both tasks. Instead, an 8051-family microcontroller, the Phillips P89C51RC2HBP, is used as a PWM generator (Figure 2.18, schematic shown in Appendix A.4). It communicates with the PC's DAQ card using a 12-bit parallel interface, consisting of an 8-bit data connection (PA0-PA7 on the DAQ card), and a 4-bit handshaking connection (PC4-PC7).
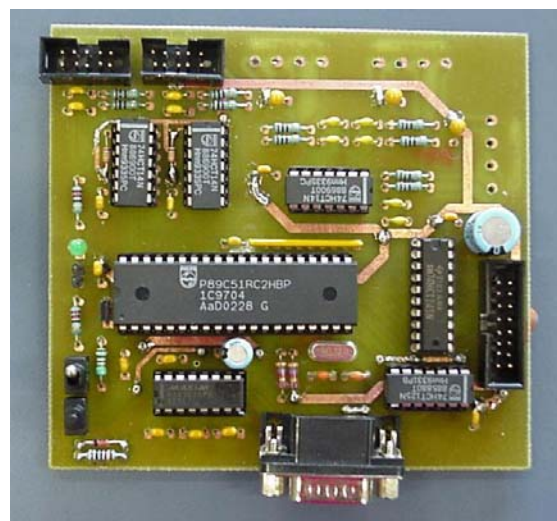
Figure 2.18: Microcontroller PCB

3 Software Interfaces

MARVIN's software utilises a number of different applications (HI-TECH C, LabVIEW, MATLAB and Microsoft Word) on two different hardware platforms (PC and 8051) that each require a means to communicate with each other. The control system must acquire data from various analogue and digital signals provided by the sensors. Software layers corresponding to the four MARVIN-related projects must exchange information. Finally a Graphical User Interface (GUI) and data logging system is required for testing purposes.

3.1 Applications

3.1.1 HI-TECH C

Programming languages based on C are among the most widely used languages in existence today. C was originally developed in 1972 based on two previous languages, B and BCPL, but it has since undergone numerous revisions. The language is hardware independent, and modified versions are commonly used in embedded controllers. MARVIN's motor driver software was developed using the HI-TECH 8051 C Compiler (HI-TECH C) by HI-TECH Software.

3.1.2 MATLAB

MATLAB (*Mat*rix *lab*oratory, shown in Figure 3.1) is a programming language designed for technical computing. Unlike other languages such as C, MATLAB's basic data element is a dynamic array of floating point numbers. Solutions are calculated numerically, so there is an error between the exact solution and the calculated one, but variables are of such high precision that this error is reduced to insignificant levels in most applications.

MATLAB provides a large selection of specialised toolboxes for applications such as control systems, signal processing, system identification and data acquisition. MATLAB's level of support in these areas makes it an ideal platform on which to base MARVIN's high level software.
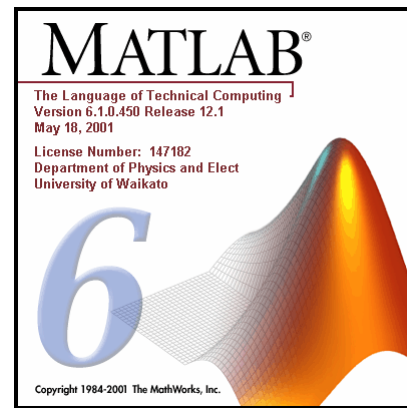
Figure 3.1: MATLAB 6.1

3.1.3 LabVIEW

LabVIEW (Figure 3.2) is a program development application based on the graphical programming language, G, developed by National Instruments. It is designed primarily for test and measurement purposes, making it useful as an interface to the data acquisition (DAQ) hardware. LabVIEW allows developers to create programs, called *virtual instruments* (VIs) to recreate the appearance and functionality of real instruments such as amplifiers and filters.
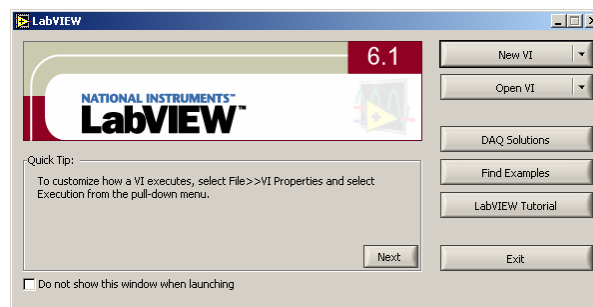
Figure 3.2: LabVIEW 6.1

Data objects are represented by blocks linked together by wires on a block diagram rather than lines of text. This form of programming allows developers with limited programming experience to create simple programs that perform useful tasks. However, an experienced software developer would take longer to implement most algorithms in G than they would in a text-based language. G code also tends to be

more difficult to follow at a glance than its text-based equivalent because many G structures consist of multiple subdiagrams that cannot be observed at the same time. Figure 3.3 gives an example of this.


```
if c > 10
    a = 0;
else
    a = b + 2;
end;
```
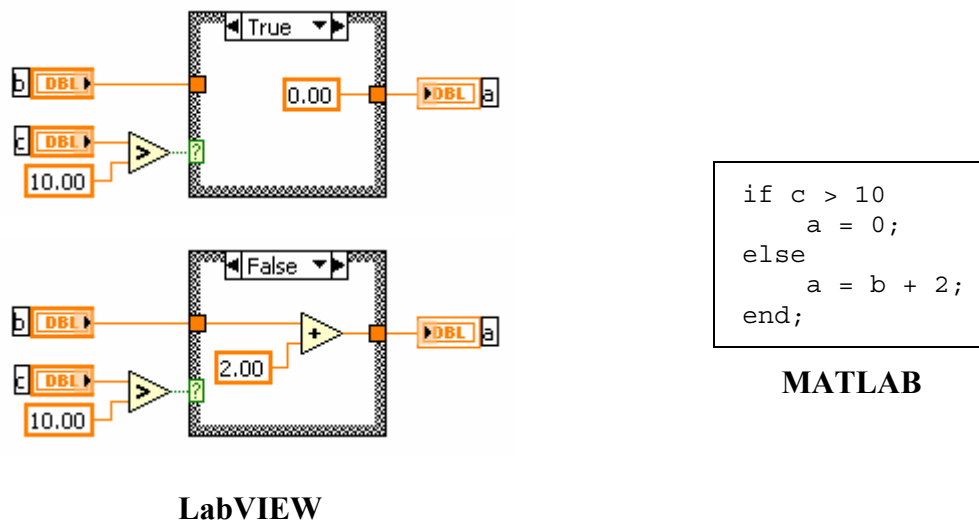
MATLAB

LabVIEW

Figure 3.3: If Structure MATLAB – LabVIEW Comparison

The inputs and outputs of a VI are called the *controls* and *indicators* respectively. Controls can be in the form of dials, slide bars, switches, buttons, check boxes or text input boxes. Graphs, charts, tables, meters, lights and text output boxes are indicators.

3.1.4 Microsoft Word

Microsoft Word is a popular word processor that includes a large selection of auto correction options for grammar and spelling. Ashil Prakash's human-machine interface uses Microsoft Word's inbuilt speech recognition rather than a stand-alone package such as Dragon Dictate. Since speech recognition is generally rather unreliable, Microsoft Word is useful as a means to automatically correct some of its mistakes [Prakash & Carnegie, 2003].

3.2 Inter-Application Interfaces

Prior to the onset of this project, most PC-based robotic software at the Mechatronics Group consisted of relatively simple LabVIEW programs. LabVIEW provides many user-friendly data acquisition VIs, and it has built in support for National Instruments DAQ cards such as the Lab-PC+ and 6025E, so it is well-suited to the task of controlling MARVIN's hardware. However, due to its graphical nature, LabVIEW is less suitable for designing the complex logic necessary for MARVIN to become autonomous.

A high level of program complexity is more easily accomplished in MATLAB. Due to its growing popularity and support in the academic community, MATLAB has become the preferred programming language to use for future projects in the Mechatronics Group. The eventual goal is to replace all current LabVIEW software with Craig Jensen's MATLAB hardware interface [Jensen & Carnegie, 2003], or an equivalent system. However, it will not be ready in time for this project.

In the meantime, an interface between the two programs has been developed. Using this interface, LabVIEW provides the low-level hardware interface, while the MATLAB code is responsible for the high-level control tasks. A similar interface has also been developed between MATLAB and Microsoft Word, so that voice commands can be passed to MATLAB for the navigation system to execute.

The following interfaces were considered:

- ActiveX Control Containment
- ActiveX Automation
- MATLAB Script Node
- Dynamic Data Exchange
- File I/O

3.2.1 ActiveX Control Containment

ActiveX is a marketing label that describes a loosely defined set of technologies developed by Microsoft to allow interaction between multiple programs without requiring developers to have knowledge of each program's inner workings. It is based on two other Microsoft technologies: COM (Component Object Model) and OLE (Object Linking and Embedding). Although ActiveX encompasses a very broad range of technologies, the only ones that are supported in MATLAB and LabVIEW are *ActiveX Control Containment* and *ActiveX Automation*.

An ActiveX *control* is an application that can be embedded in the client's *control container*. The control can send notifications back to the client in the form of *events*, which can trigger the client's event handler routine. Since MATLAB's ActiveX Automation lacks support for events, ActiveX Control Containment is potentially the more powerful of the two interfaces.

MATLAB can control another application in this manner from within a figure window, using the **actxcontrol** function (refer to Figure 3.4 for an example). Similarly, ActiveX control container blocks can be created in LabVIEW. However, neither program can itself be an ActiveX control, so this is unsuitable for interfacing these two programs.

```
% Windows Media Player ActiveX control.
hf=figure('Position',[120 370 316 100]);
h=actxcontrol('MediaPlayer.MediaPlayer.1',[20,10,260,80],hf);
set(h,'FileName','c:\WINNT\Media\Windows Logon Sound.wav');
```
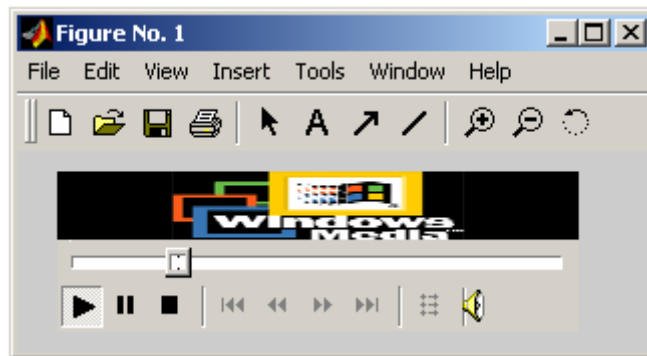

Figure 3.4: Windows Media Player Control in MATLAB Figure Window

3.2.2 ActiveX Automation

Like the ActiveX Control Containment, ActiveX Automation allows one program (the *client*) to control another (the *server*), but an Automation server is generally not embedded in the client application. The client simply calls the server like an ordinary function, and it must wait for the server to finish its task before it can continue.

MATLAB and LabVIEW support ActiveX Automation as both clients and servers. Of particular interest is the MATLAB function **actxserver**, which can be used to set up a LabVIEW server when it is given LabVIEW's *program ID* as a parameter. The ActiveX program ID is a unique entry in the registry used by other programs to identify it. Each ActiveX object has a set of *properties*, which are variables controlled by that object, governing, for instance, the appearance of its GUI or the files and directories it can access. Equally important are an object's *methods*, similar to function calls, which are requests for the object to perform an action, such as returning the value of a variable. MATLAB can access LabVIEW's properties and methods, and through them gain read/write access to a LabVIEW VI's controls and indicators. This interface is relatively simple to use, and it allows the two programs to communicate with reasonable efficiency. Figure 3.5 shows the creation and manipulation of a LabVIEW ActiveX server in MATLAB.

```
% Set up LabVIEW ActiveX server. Open VI window.
lvserv = actxserver('LabVIEW.Application');
vi = invoke(lvserv,'GetViReference',...
    'c:\Project\Code\LabVIEW\Wheel Controller.vi');
vi.FPWinOpen = 1;
```

Figure 3.5: LabVIEW ActiveX Automation Server in MATLAB

3.2.3 MATLAB Script Node

LabVIEW 5.1 has built in support for MATLAB via *script nodes* – blocks on the LabVIEW block diagram that can execute MATLAB code using ActiveX. An example of a MATLAB script node is shown in Figure 3.6 below.
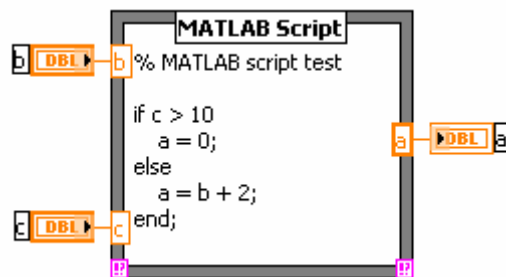
Figure 3.6: MATLAB Script Node in LabVIEW

National Instruments recommends the script node as the simplest, most efficient interface between the two programs. However, it is not the best approach for this task. The high-level code will be in MATLAB, with LabVIEW only providing the interface to the hardware. Consequently, MATLAB should be the controlling program, not LabVIEW. The problem can be circumvented if the MATLAB functions are called from within a main VI's loop, but this is an inelegant solution. It would result in further difficulty when the control system is converted into pure MATLAB code. Compatibility issues might also arise with some MATLAB toolboxes.

3.2.4 Dynamic Data Exchange

Dynamic Data Exchange (DDE) is a protocol used to send data between programs. Unlike ActiveX, it does not give one program direct control over another – a program can only be manipulated in this way if it can treat data as commands. The data is exchanged asynchronously, so a handshaking or interrupt mechanism is needed to ensure that a program receives the data that was sent.

MATLAB supports DDE through a group of client functions (**ddeadv**, **ddeexec**, **ddeinit**, **ddepoke**, **ddereq** and **ddeunadv**) that set up the link and perform the necessary data exchanges. LabVIEW can then be configured as a DDE server using the **DDE Srv Register Service** VI. The disadvantage of this method is that both the client and server must be set up and run independently, which introduces unwanted complexities such as timing issues. Overall, the difficulties of setting up a reliable DDE interface for this project outweigh its advantages.

3.2.5 File I/O

This approach involves writing data into a file with one program, and reading it from the file with the other. This is the only method that is virtually guaranteed to work with any program on any platform. However, disc access is very slow, so this is the least efficient method. Moreover, synchronising the two programs this way is difficult. It would be used only as a last resort.

MATLAB supports file I/O through functions such as **fopen**, **fclose**, **fread**, **fwrite**, **fprintf**, and **fscanf**. Similarly, LabVIEW provides a range of file I/O VIs, including **Open/Create/Replace File**, **Write Characters To File** and **Read Characters From File**.

3.2.6 Selected Interface: ActiveX Automation

Although some of these interfaces are not viable in this application, they each have situations where they are useful. File I/O must be used when the programs lack a common alternative interface. DDE is useful for communication between programs on different machines, or on non-Windows operating systems. MATLAB script nodes are the most convenient technique for calling MATLAB functions from LabVIEW VIs. ActiveX Control Containment provides a powerful interface for applications that support it.

However, due to its ease of use, and its level of support in MATLAB, LabVIEW and Microsoft Word, ActiveX Automation was selected over these other interfaces.

3.2.7 MATLAB – LabVIEW Interface Details

ActiveX Automation is used extensively throughout the control system wherever it communicates with the sensor and motor driver VIs.

Some LabVIEW properties and methods can be invoked directly in MATLAB. Methods with more than one argument must be accessed using the **invoke** function. This function calls an object's methods with the given arguments, and outputs the methods' return values.

LabVIEW communicates using two distinct ActiveX classes of object: the Application class, and the Virtual Instrument class. The Application class gives MATLAB access to the properties and methods that affect LabVIEW as a whole. It can be created directly in MATLAB. The Virtual Instrument class allows MATLAB to manipulate individual LabVIEW VIs. It cannot be created directly, but it can be instantiated using the **GetVIReference** Application class method.

The **FPWinOpen** Virtual Instrument class property opens up a front panel window of the LabVIEW VI object, so that the VI's controls and indicators can be manipulated and viewed. This is not necessary for the final control system – the GUI is implemented in MATLAB – but it is used for testing purposes.

The most important methods for this interface are **GetControlValue** and **SetControlValue** from the Virtual Instrument class, which can read and write to a VI's controls and indicators. Invoking **GetControlValue** returns the requested variable in its original data format, so care must be taken to only return values in data types compatible with MATLAB. For example, in MATLAB versions earlier than 6.5, Booleans should be converted to integers or floating-point numbers in the LabVIEW VI before they are passed to MATLAB. **SetControlValue** accepts only

strings as parameter inputs, so all numeric data types must be converted in MATLAB, using the **num2str** or **int2str** functions.

After the VI's controls have been set, the VI is run using the Virtual Instrument class method, **Run**. Then the controls and indicators are passed back to MATLAB, where the process is repeated.

3.2.8 MATLAB – Microsoft Word Interface Details

Figure 3.7 shows a test program that opens a Microsoft Word document, reads text strings into MATLAB to be processed, and deletes them from the Microsoft Word document.

```
% Set up MS Word ActiveX server. Open document.
wordserv = actxserver('Word.Application');
wordserv.Visible = 1;
set(wordserv.Options,'ReplaceSelection',0);
invoke(wordserv.Documents,'Open','h:\Project\test.txt');

% Select text. Return it to MATLAB. Delete text.
invoke(wordserv.Selection,'SetRange',0,10000);
strng = get(wordserv.Selection,'Text')
invoke(wordserv.Selection,'Delete');

% Quit MS Word. Don't save changes.
invoke(wordserv,'Quit',0);

% Clean up ActiveX objects to help prevent memory leaks.
release(wordserv);
```

Figure 3.7: Sample MATLAB – Microsoft Word Interface

The **Word.Application.Visible** property opens a Microsoft Word window. It can be set directly in MATLAB. However, the **Word.Options.ReplaceSelection** property must be set using the MATLAB **set** function, since a **Word.Options** object has not been declared. This property sets the option "Typing Replaces Selection", normally accessible from the "Edit" tab of the options menu in Microsoft Word. It causes new text to appear in front of the selected text, rather than overwriting it.

Word.Documents.Open   is   the   method   used   to   open   a   document. **Word.Selection.SetRange** selects characters for processing in Microsoft Word – in this case the range is set large enough for every character in the document to be

selected. The **Word.Selection.Text** property returns the selected characters as a text string, ignoring any formatting information. It is returned to MATLAB using the **get** function. After it has been retrieved, the selected text is deleted from the document using **Word.Selection.Delete**.

Finally, the application window is closed (without saving changes to the document) using **Word.Application.Quit**, and the ActiveX server is released from memory.

3.3 Sensor Interfaces

Before the LabVIEW hardware/software interface VIs were written, the various sensor signals were configured in the Measurement and Automation Explorer. This utility also includes a test panel (Figure 3.8) that provides direct access to the DAQ card's analogue inputs and outputs, digital I/O ports and counters.
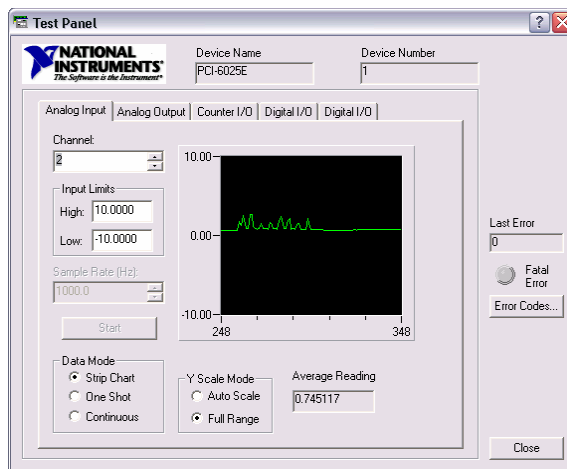
Figure 3.8: Measurement and Automation Explorer Test Panel

Following configuration and testing, the following interface VIs were developed for MARVIN's sensors:

- Digital Switch Input
- Encoder Counter
- IR Analogue Input

3.3.1 Tactile Sensors and Beacon Receivers

The tactile sensors and beacon receivers provide digital signals that are accessed using the **Digital Switch Input** VI (block diagram given in Figure 3.9). This VI simply polls the **Dig Line** library VI for the first six lines on port 0 of the DAQ card. The **iteration** control must be set to 0 during the first call, indicating that **Dig Line** should initialise the line.
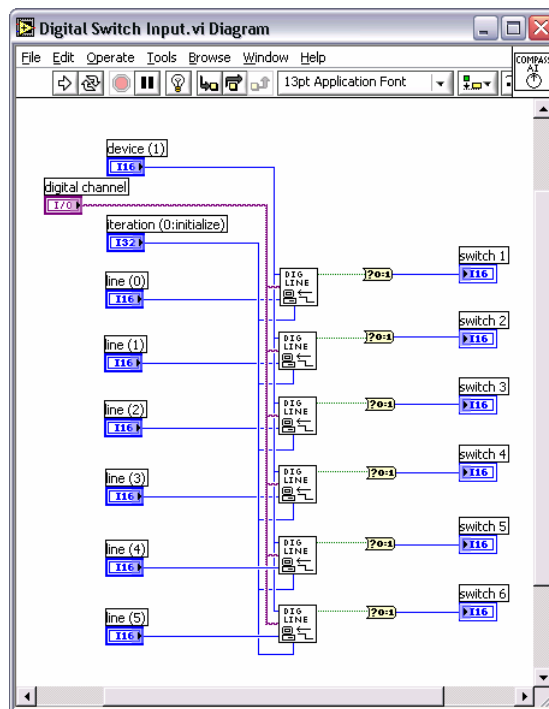
Figure 3.9: Digital Switch Input VI Block Diagram

3.3.2 Optical Encoders

The counters that are attached to the optical encoders are controlled using **Encoder Counter** (block diagram given in Figure 3.11). During the first call (designated by setting **iteration** to zero), each counter is configured to increment on the rising edge of the encoder signal using the **Event Or Time Counter Config** library VI. The **Counter Start** VI sets the counters to begin incrementing on the next rising edge of the encoder signal. **Counter Read** is used to access the counter value in each subsequent call. Both **Counter Start** and **Counter Read** require a task ID input – a

value representing the device being addressed, and the I/O operation. **Event Or Time Counter Config** provides this value during the first call, but a predefined constant is used in subsequent calls.
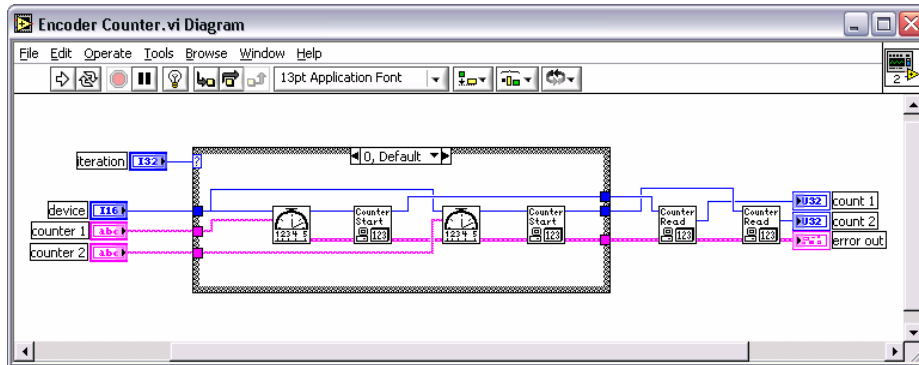
Figure 3.10: Encoder Counter VI Block Diagram

3.3.3 Infrared Rangefinders

IR Analogue Input measures analogue voltages from the six Analogue to Digital Converters (ADCs) attached to the infrared rangefinders. It simply calls the **AI Sample Channel** library VI once for each rangefinder. The DAQ card is configured in its RSE mode of operation using the Measurement and Automation Explorer.
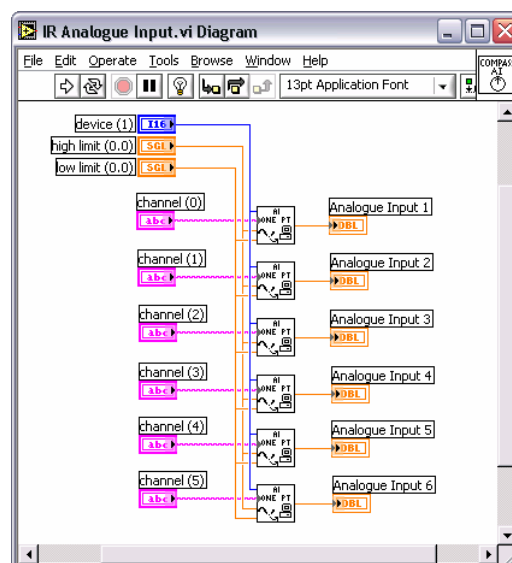
Figure 3.11: IR Analogue Input VI Block Diagram

3.4 Microcontroller Interface

Andrew Payne has developed a protocol for communication between his microcontroller board and a PC [Payne & Carnegie, 2003]. The microcontroller software was developed in HI-TECH C, and required few modifications to operate on MARVIN. However, the PC side of Andrew Payne's interface, coded in Visual C++, was not used because it would require an additional inter-application interface, and only a single application can communicate with the DAQ card at a time. Thus, a replacement interface was developed in LabVIEW that utilised the same protocol so that the microcontroller code could be retained.

3.4.1 Communication Protocol

Instructions are delivered to the microcontroller as two bytes of data. The first byte controls which motor to address and which direction to drive it, while the second byte represents the PWM value. If a valid instruction executes correctly, the microcontroller returns the inverse of the received byte (i.e. each bit is toggled), otherwise it returns an error signal. This allows the PC software to resend the instruction if it fails. The bit pattern of the protocol is given in Table 3.1.

Table 3.1: PC – Microcontroller Communication Protocol

Instruction Type	Bit Pattern								Example (Hex)
Header – Sent[*]	0	0	0	0	0	L	R	D	05
Header – Returned[*]	1	1	1	1	1	!L	!R	!D	FA
Data – Sent	1	PWM/2							AD
Data – Returned	0	!(PWM/2)							52
Error - Returned	0	1	1	1	1	1	1	1	7F

*   Left wheel: L = 1, R = 0

         Right wheel: L = 0, R = 1

         Forwards: D = 0

         Reverse: D = 1

3.4.2 Microcontroller Software

The microcontroller software translates the received data into a PWM signal and direction bit for the selected motor driver. The software also filters out any transient direction changes that might result from noise on the data or handshaking lines. Additionally, if the microcontroller receives no instructions for 500 ms, the software times out and sets the PWM duty cycle to zero. This is a safety precaution to prevent collisions in the event of a PC lockup.

If necessary, the PWM duty cycles can be limited by setting the **Motor0Limit** and **Motor1Limit** variables to nonzero values. An optional acceleration limit also exists to protect the motor drivers from damage that could result from rapidly increasing or decreasing the PWM duty cycle, or reversing direction. This is implemented using a timer interrupt that updates the PWM every 100 ms. The PWM changes are not allowed to exceed the values of **Motor0AccMax** and **Motor1AccMax**.

3.4.3 PC – Microcontroller Interface Software

The microcontroller interface is implemented in the **Set Motor Power** LabVIEW VI (Figure 3.12). Firstly, the DAQ card's 82C55A PPI is configured for bi-directional communication on Ports A and C using the **Digital Group Config** library VI. This allows the software to utilise the automatic handshaking capabilities of the OBF_A, ACK_A, IBF_A and STB_A lines on Port C.

Data is written to the microcontroller using **Digital Single Write**, and read using **Digital Single Read**. These VIs are configured to access Port A only when the

handshaking lines indicate that the microcontroller is ready to send or receive data. This prevents the PC from writing to the port at the same time as the microcontroller, and ensures that data is not lost. The VIs must be called sequentially – a write, followed by a read (to ensure that the instruction was received and executed) – but LabVIEW does not execute code in an explicit sequence by default. Consequently, the VI's are executed inside a *sequence* structure – a structure that executes subdiagrams, or *frames*, in a predefined order.

The header and PWM values are written and verified for each motor, yielding a total of four read-write operations that are carried out whenever the VI is called. Each time a read-write operation is executed, the value returned from the microcontroller is compared with the value written to it. If the returned value is not the inverse of the written value, the read-write operations are repeated. If an error is returned after writing the PWM value, both the header and PWM instructions are redelivered. The error corrections are repeated up to ten times using *while* structures. A small *while* structure that encloses the header operation is in turn enclosed by a larger structure that encompasses both operations.

Much of the error correction code could be implemented in MATLAB rather than LabVIEW. However, programs that utilise the MATLAB-LabVIEW interface are an order of magnitude slower than those that run directly in LabVIEW. In order to ensure that erroneous instructions can be redelivered in time to prevent the motors from responding to them, it was necessary to implement the error correction directly in LabVIEW.
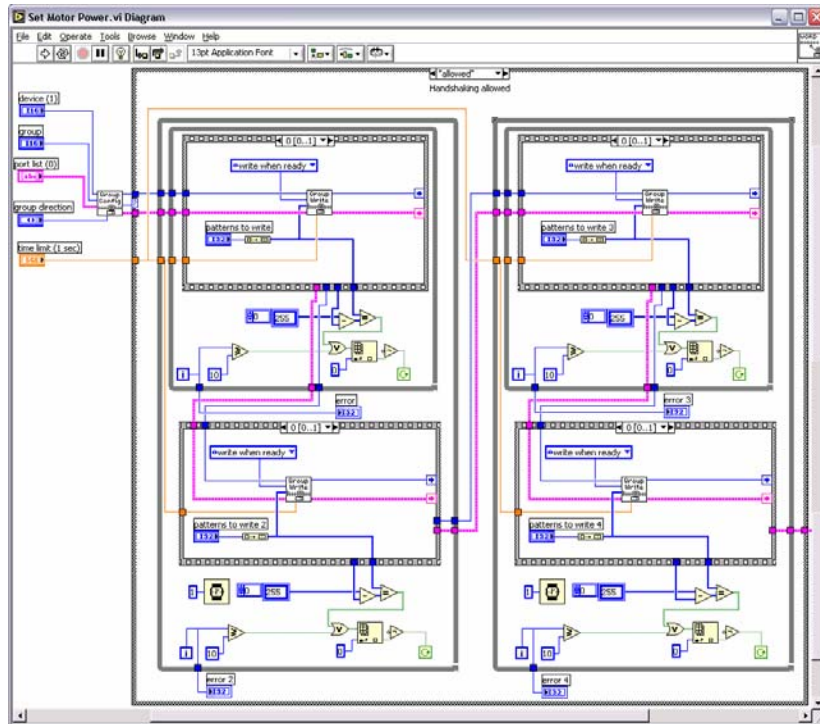
Writing to Microcontroller
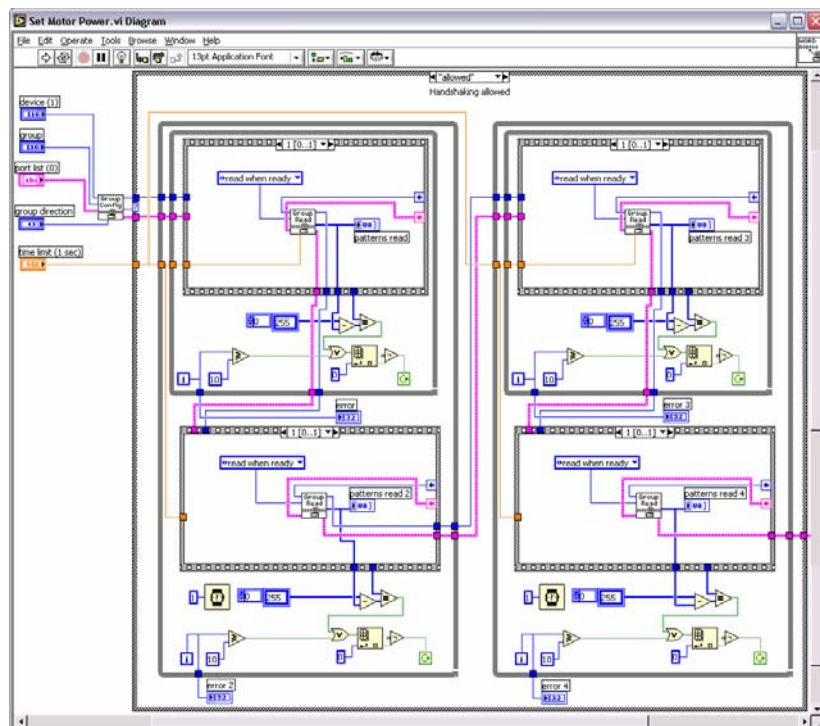
Reading from Microcontroller

Figure 3.12: Set Motor Power VI Bock Diagram

3.5 MATLAB Interface

MARVIN's program structure consists of four layers, as shown in Figure 1.3, loosely corresponding to the four elements of MARVIN's project hierarchy given in Figure 1.3. The top layer is Ashil Prakash's speech recognition human-machine interface [Prakash & Carnegie, 2003]. The second layer is the navigation system designed by Lucas Sikking [Sikking & Carnegie, 2003]. This project consists of the two bottom software layers – the control system developed in MATLAB and the LabVIEW hardware interface. Craig Jensen's generic hardware interface [Jensen & Carnegie, 2003], or an equivalent system, may replace the LabVIEW system at a later date.
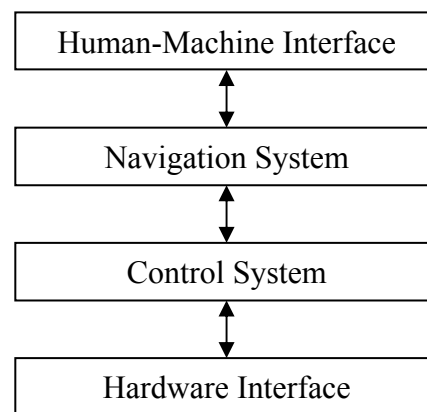
Figure 3.13: Program Structure

The three upper layers are implemented in MATLAB (although the human-machine interface also utilises Microsoft Word for speech recognition). Since the control system does not communicate directly with the human-machine interface, this report only details the interface between the control system and the navigation system.

3.5.1 MATLAB Interface Details

MATLAB lacks generic support for real-time mechanisms such as interrupts and multithreading. It does provide asynchronous behaviour for certain specialised tasks such as GUIs and serial communication, in the form of *callback* functions − functions that are called when a particular event occurs. Some toolboxes also provide generic real-time support for Simulink, a graphical modelling and simulation tool for MATLAB. However, the options for MATLAB itself are more limited.

These limitations, coupled with the complexities inherent in interrupt-driven systems, mean that the software relies on polling for most tasks. Each software layer is implemented as a function that must be called by the preceding layer often enough to operate correctly. The main control system function, **marvin_control** (Appendix B.2), is called by the navigation system at a frequency of at least 10 Hz. A delay loop within **marvin_control** ensures that the control cycle period is approximately constant. This system is adequate for the small number of layers involved, but once more high-level algorithms are added it will likely become necessary to convert to a less sequential approach.

Data that must be maintained between function calls is stored in *persistent variables*. A persistent variable is like a global variable that can only be utilised by the function in which it was declared. It remains in memory between function calls, retaining the value it held during the previous call. Persistent variables must be initialised during the function's first call, so each function that uses them includes a condition check to determine if the current call is the first. This introduces a small overhead, but the main system bottlenecks reside elsewhere, so the overall impact on performance is negligible. The use of persistent variables greatly reduces the number of arguments that must be passed to a function. Unnecessary details can thus be hidden from higher-level functions, improving code readability and reusability.

3.5.2 Control System Inputs

- **Instruction Flag** – This flag indicates whether the call is the first call, a new instruction, an emergency brake instruction or a request to continue executing the last instruction that was given.

- **Basic Instruction** – Distance and angle variables comprise the simplest form of instruction. If a position is delivered with an angle of zero, the control system will attempt to move MARVIN in a straight line. An angle given with zero distance is a request for a stationary turn. Any combination of non-zero values indicates a moving turn along a circular path.

- **Offset Angle** – Normal heading corrections require a sequence of two or more instructions – first MARVIN is reoriented, then the original instruction is resumed. The offset angle provides a simpler alternative that is useful for small heading adjustments. A non-zero offset angle indicates a heading error that the control system will attempt to correct while in motion.

- **Rangefinder Weights** – These indicate the priority level of the infrared rangefinders for localisation purposes.

- **Corridor Coordinates** – The corridor offset and angle coordinates represent MARVIN's offset from the corridor centre axis and the direction of the centre axis respectively. They are only adjusted when MARVIN enters a new section of corridor or room.

- **Wall Offsets** – This two-element array represents the offset coordinates of the corridor walls with respect to the corridor centre axis.

- **Origin Coordinates** – These are a set of Cartesian coordinates and an angle that represent MARVIN's initial position and orientation.

3.5.3 Control System Outputs

- **Time** – The time since the first call, in seconds.

- **Absolute Coordinates** – A set of Cartesian coordinates and an angle that represent MARVIN's position and orientation with respect to the origin.

- **Relative Coordinates** – Distance along the corridor centre axis, offset from the centre axis, and heading with respect to the corridor angle. These values are reset when the corridor angle changes.

- **Target Coordinates** – Relative coordinates representing MARVIN's intended position and orientation on the target trajectory.

- **Rangefinder Coordinates** – Offset and heading measured by the rangefinders.

- **Wheel Velocities** – Measured velocity of each wheel.

- **Rangefinder Data** – Raw distances measured by the rangefinders.

- **Tactile Sensor Data** – Booleans representing the state of each contact sensor.

- **Beacon Data** – Booleans representing the state of each beacon receiver.

3.6 Graphical User Interface

Although the control system is designed to receive instructions from the navigation system, it must also be manually controllable during testing. The software simulation (Section 5.7) requires a real-time graphical output of MARVIN's motion. This would also provide the ability to monitor the localisation algorithm's accuracy during real-world testing.

These requirements are fulfilled through the use of a GUI. The MATLAB Layout Editor (GUIDE) provides a graphical means to add and adjust GUI elements. It generates a generic MATLAB file that can be edited to perform specific tasks. MATLAB GUIs utilise callback functions to provide asynchronous responses to user inputs. When the user triggers a button or slider on the GUI, the corresponding callback function is activated. Input and output states are stored in a data structure that is accessible from all the callback functions.

The control system GUI is given in Figure 3.14, and the corresponding MATLAB file, **gui_marvin_control.m**, is given in Appendix B.1. The main GUI function **gui_marvin_control** is largely unaltered from the generated function. The only additions are initialisations of various elements stored in the main data structure.
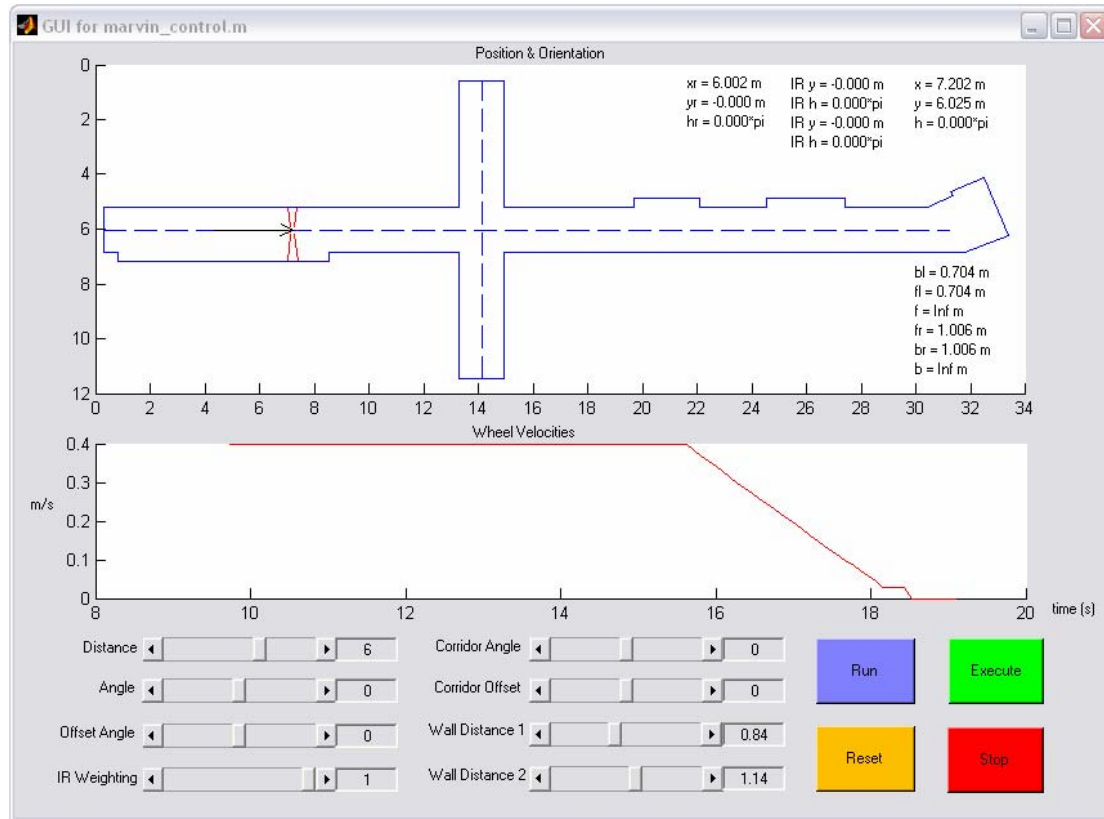
Figure 3.14: GUI Window for Control System

Each input button or slider has a corresponding callback function that updates its value in the main data structure. Slider values are also written to the GUI as text strings. The main callback function, **run_Callback**, is triggered from the GUI's **Run** button. It calls **marvin_control** continuously until the **Run** button is triggered a second time. The **marvin_control** function's input parameters are continuously adjusted to match the values on the sliders and buttons, while its outputs are plotted on the GUI's two figure axes, or displayed as text.

The standard function used for plotting data in MATLAB, **plot**, is too inefficient for real-time plotting because it redraws the entire figure each time it is called, even if the data has not changed. Instead, the **line** function is used to plot a static number of

straight-line segments. These line segments are updated in a First-In-First-Out (FIFO) arrangement, so that for each program cycle the oldest line is replaced with new values using the **set** function.

MARVIN's trajectory is plotted in the upper axis of Figure 3.14. Also included in the figure is a direction arrowhead with endpoints derived from the current heading using Equations 3.1 and 3.2, and a set of lines indicating the range detected by each rangefinder. The velocity of each wheel is plotted over time in the lower figure.

$$x_a = x - l_a \cos(\theta \pm \psi)$$ **Equation 3.1**

$$y_a = y - l_a \sin(\theta \pm \psi)$$ **Equation 3.2**

x: MARVIN's x coordinate (m)
y: MARVIN's y coordinate (m)
θ: MARVIN's heading (rad)
x_a: Arrowhead endpoints' x coordinates (m)
y_a: Arrowhead endpoints' y coordinates (m)
l_a: Length of arrowhead lines (m)
ψ: "Sharpness" angle of arrowhead (rad)

As well as being plotted in real time, the data returned from **marvin_control** is logged to a file for debugging purposes. Data is written to file as a table of values (stored as strings) separated by tab characters. Unique filenames are generated from their creation time using the **datestr** library function. The ISO 8601 notation given below is used because it is the only representation available that includes time values but omits characters that are illegal for filenames.

ISO 8601 standard notation for date and time stored in a single data field:

    yyyymmddTHHMMSS

    y: Year

    m: Month

    d: Day

    T: Separator between date and time

    H: Hour

    M: Minute

    S: Second

4 Sensor Software

Just as an animal obtains knowledge of its surroundings using one or more of its senses (vision, hearing/sonar, smell, taste and touch), a mobile robot uses data from its sensors to produce an internal model of its state with respect to its environment. In MARVIN this model consists of simple distances, angles and velocities that the robot uses to autonomously maintain its intended trajectory, or take evasive action if necessary.

The three main steps to building this model are as follows:

- **Data Acquisition** – Obtaining raw sensor data.
- **Internal Representation** – Converting raw data into a usable form.
- **Sensor Fusion** – Combining data from multiple sensors.

4.1 Data Acquisition

The LabVIEW VIs that interface to the sensor hardware via the data acquisition card have already been detailed in Section 3.3. This section deals with the MATLAB functions that read data from the VIs. The values returned from these functions are not true raw data – various modifications have been made so that useful information can be obtained – but they are in the same form as the original data (i.e. counts from the optical encoders, voltages from the infrared rangefinders).

4.1.1 Odometers

The **acq_en_count** function (Appendix B.3) reads in counter values from the **Encoder Counter** LabVIEW VI (Section 3.3.2), and outputs the number of counts measured since the last call, and the time interval between calls.

MARVIN's software utilises the **cputime** library function to measure the time elapsed between program cycles. This function outputs the time, in seconds, since the program started, with a resolution that depends on the hardware platform. MARVIN's xPC provides millisecond precision, which is adequate for the control cycle period of 93 ms.

During each call, the time measured during the previous call is subtracted from the current value. If the measured time difference is less than the intended control cycle period, the function enters a delay loop. This results in an approximately constant period (which is necessary for an efficient control algorithm), and it ensures that the measured time interval is large enough to yield accurate velocity measurements.

Similarly, the previous counter values for each encoder are subtracted from the current values returned from **Encoder Counter**. Negative values indicate when the 24-bit counters have overflowed, at which time the values are adjusted accordingly. Given the wheel speeds MARVIN encounters, and the control cycle period used, the counters never overflow more than once in a single control cycle.

The raw counter values do not take wheel direction into account, so they are adjusted according to the direction that the motors are being driven. In order to protect the motor drivers, the software does not reverse a wheel's direction while it is in motion. This means that the wheel direction could only be calculated incorrectly if an external force moved the wheels in opposition to the driving motors, which is unlikely to occur in the intended indoor operating environment.

4.1.2 Rangefinders

Utilising the **IR Analogue Input** VI (Section 3.3.3), **acq_ir_voltage** (Appendix B.4) returns an array of voltages sampled on each of the six analogue input ports attached to the IR rangefinders. The only modification made to the raw voltages is a software filter that reduces noise. At long ranges a small change in voltage results in a large change in the measured distance, so it is necessary to filter out noise before the voltage-distance conversion is carried out. A number of different software averaging techniques were considered for the filter, including:

- Mean
- Median
- Weighted Mean

4.1.2.1 Mean

$$\bar{E} = \frac{\sum_{i=1}^{n} E_i}{n}$$ **Equation 4.1**

E_i: i^{th} sample

$\bar{E}$:   Mean average of samples

n: Number of samples

This technique is easily implemented in MATLAB using the library function **mean**. It provides fast response to change, but extreme values can significantly affect the result (unless a large number of samples are taken, which would negate the speed benefit). Thus the mean is most effective when filtering small amounts of noise.

4.1.2.2 Median

$$E_{med} = E_{n/2}, \quad E_1 \le E_2 \le \ldots \le E_{n-1} \le E_n \qquad \textbf{Equation 4.2}$$

E_{med}:   Median average of samples

This technique can be applied in MATLAB using the **median** function. In general, the median average is less sensitive to extreme values than the mean, but it is also slower to respond.

4.1.2.3 Weighted Mean

$$\overline{E} = \frac{\sum_{i=1}^{n} w_i E_i}{n}, \quad \sum_{i=1}^{n} w_i = 1 \qquad \textbf{Equation 4.3}$$

w_i: i^{th} weighting

No library functions are available to calculate the weighted mean, but the scheme is relatively easy to implement using simple arithmetic. A number of different techniques can be used to assign weights to each sample. For example, weights can be allocated according to the time at which samples were taken – i.e. more recent samples are given higher weights. Another possibility is to apply weights according to a sample's proximity to the median value, providing a compromise between the mean and median techniques.

4.1.2.4 Selected Implementation

Due to factors such as ambient light and electrical crosstalk, a high level of noise can be observed on the rangefinder inputs – especially at long range. Consequently, in this application the median average is superior to the mean. Depending on the implementation, a weighted mean could also be effective. However, the median average provides sufficient filtering, and with greater efficiency than a weighted average could achieve, so it is the logical choice. The median average is taken over ten samples (corresponding to approximately 935 ms), providing an optimal compromise between accuracy and speed of response.

4.1.3 Tactile Sensors and Beacon Receivers

The tactile sensor switches and beacon receiver relays are both acquired using the **Digital Switch Input** VI (Section 3.3.1). The **acq_switch** function (Appendix B.5) reads in signals from the indicators representing each of the six digital lines, and returns them as an array of Boolean values.

Since the MATLAB-LabVIEW interface lacks support for callbacks or interrupts, the digital lines must be polled. This situation is less than ideal when responding to collisions, since it causes a delay of up to one control cycle period (approximately 100 ms). In practice however, this delay is insignificant compared to the motors' response time, so its effect on the system is not noticeable.

4.2 Internal Representation

If data from multiple sensors is to be combined effectively, it must first be converted into an internal representation that is shared by all the sensors. The most significant data that can be derived from multiple sensors is a set of coordinates defining MARVIN's position and orientation. Since MARVIN is primarily designed to operate in a narrow, rectangular corridor or laboratory environment, its position is given in

Cartesian coordinates – distance x_M along the corridor or room, and offset y_M from its centre axis – while its heading is defined as an angle θ_M, in radians (Figure 4.1).
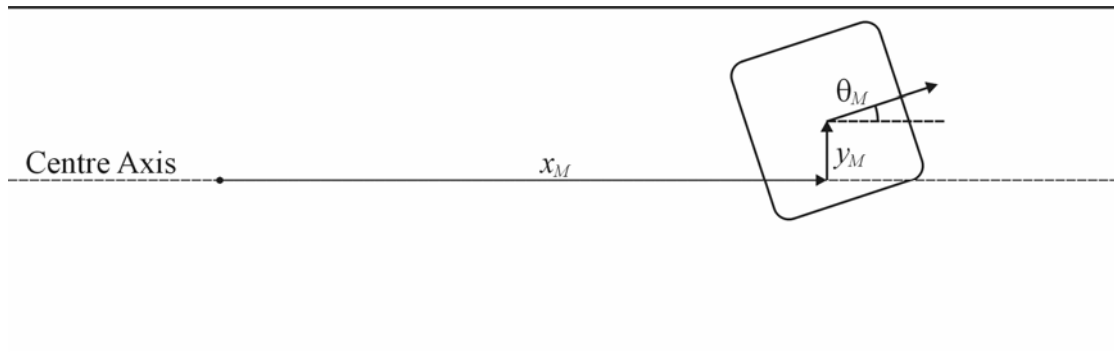
Figure 4.1: Internal Representation Coordinate System

The issue of representation is not entirely limited to overlapping sensor data. Data that is unique to a single type of sensor, such as MARVIN's wheel velocities and object distances, must still be provided in a form that the control system can utilise.

4.2.1 Odometers

The MATLAB function **rep_en_velocity** (Appendix B.6) obtains velocity information from the encoder counts and time given by **acq_en_count**. The **rep_en_coord** function (Appendix B.7) converts individual wheel distances to an overall position and heading for MARVIN.

4.2.1.1 Odometer Conversion Factors

Due to factors such as wheel slippage, missed counts, gear slop and non-uniform tyre radii, the theoretical odometer count/distance conversion factor (24867 pulses per metre) only approximates the actual count/metre ratio. More accurate conversion factors for each wheel are obtained experimentally.

A number of straight-line motion instructions with different velocity limits (or peak velocities for the velocity profiles) are executed in the same manner as the tests shown in Section 6.1. The ratio between the actual distance travelled and the distance measured by the encoders for each instruction is plotted in Figure 4.2.
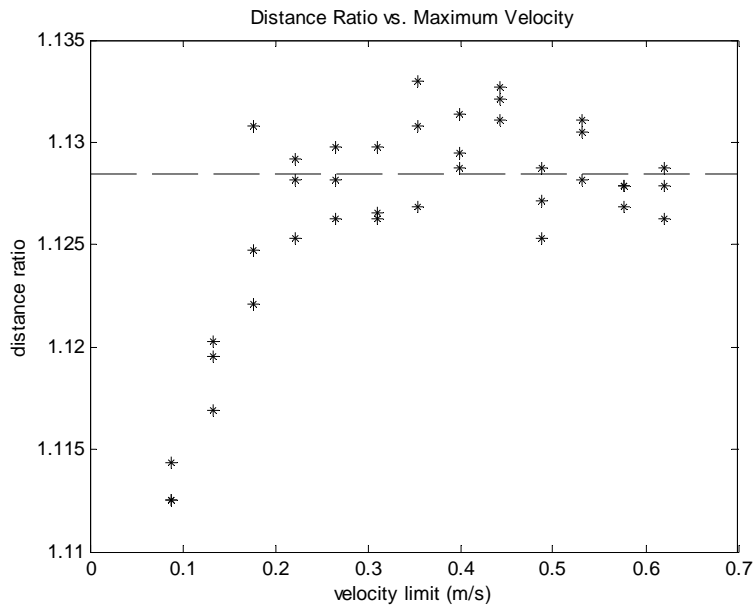
Figure 4.2: Obtaining Odometer Correction Factor

The distance ratios are approximately uniform for velocity limits greater than 0.2 m/s, which suggests that there is no measurable change in wheel slippage over this range. The theoretical conversion factor is multiplied by the average of these ratios to eliminate the systematic error. Since 0.2 m/s is the minimum value intended for the velocity limit during normal operation, only the distance ratios for velocity limits greater than this are averaged. The resulting average distance ratio of 1.1285 yields a conversion factor of 28062 pulses per metre.

The real-world conversion factor is not equal for both wheels – an overall drift to the left is observed if they are assigned equal values, most likely due to unequal tyre radii. Consequently, the value obtained above must be adjusted experimentally for each wheel to reduce the systematic error.

A preliminary measure of the degree of odometer asymmetry can be accomplished by recording the average ratio of distances measured by each wheel while manually

pushing MARVIN along a straight line. However, the results are distorted because the forces exerted on the wheels when MARVIN is being pushed are different from those provided by the motors during autonomous operation. The average ratio resulting from these tests is 0.9817, which yields multipliers of 1.00915 for the left wheel, and 0.99085 for the right wheel. If they are applied to the conversion factors, these values overcompensate for the error, resulting in a significant drift to the right. Consequently, the multipliers are adjusted manually until the systematic error is reduced to satisfactory levels. The final conversion factors are:

Left Wheel: 28202 pulses per metre
Right Wheel: 27795 pulses per metre

4.2.1.2   Wheel Velocities

The conversion factors given above provide the distance travelled by the perimeter of each wheel. These distances divided by the measured time interval provide the wheel velocities.

The limited resolution of the time measurement introduces noise into the measured velocities, so filtered velocities are also provided for algorithms where low noise is more important than speed of response. A weighted mean average is used, where the weighting for each value is twice that of the preceding value (0.5, 0.25, 0.125, ...). This provides a satisfactory level of filtering while minimising the delay.

4.2.1.3   Position and Orientation

Assuming no wheel slippage occurs, each wheel movement results in a change in MARVIN's position and/or heading. If both wheels move the same distance in the same direction, MARVIN travels in a straight line – its position changes, but its heading remains constant. If each wheel moves the same distance, but in opposite directions, a stationary turn results – MARVIN's position remains constant but its heading shifts. Any motion other than these two extremes will result in a moving turn

– a shift of both position and heading. The correlation between individual wheel movements and MARVIN's overall motion is given by Equations 4.4-4.6. It is illustrated in Figure 4.3.
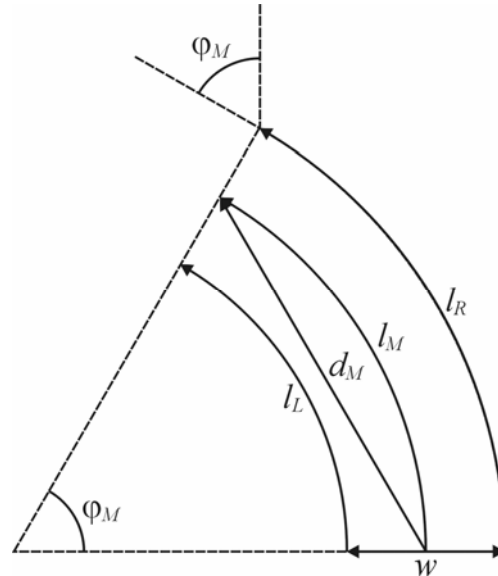
Figure 4.3: Correlation Between MARVIN's Motion and Wheel Motion

$$\varphi_M = \frac{l_L - l_R}{w}$$ **Equation 4.4**

$$l_M = \frac{l_L + l_R}{2}$$ **Equation 4.5**

$$d_M = \frac{l_C \sqrt{2(1 - \cos(\varphi_M))}}{\varphi_M}$$ **Equation 4.6**

φ_M: Angle travelled by MARVIN's centre (rad)

w: MARVIN's wheel separation distance (m)

l_L: Arc-length travelled by left wheel (m)

l_R: Arc-length travelled by right wheel (m)

l_M: Arc-length travelled by MARVIN's centre (m)

d_M: Distance travelled by MARVIN's centre (m)

During testing a small systematic error is observed on the measured angle, possibly due to non-uniform tyre radii and/or inaccurate wheel separation measurements. In

order to compensate for this, an experimentally obtained correction factor of 0.971 is applied to Equation 4.4.

Finally, the calculated distance and angle are converted into a set of Cartesian coordinates using Equations 4.7-4.9, which are added to MARVIN's position and orientation.

$$\Delta\theta_M = \varphi_M \qquad\qquad\qquad\textbf{Equation 4.7}$$

$$\Delta x_M = D\cos(\theta) \qquad\qquad\qquad\textbf{Equation 4.8}$$

$$\Delta y_M = D\sin(\theta) \qquad\qquad\qquad\textbf{Equation 4.9}$$

θ_M: MARVIN's heading (rad)

x_M: MARVIN's distance along centre axis (m)

y_M: MARVIN's offset from centre axis (m)

4.2.2 Rangefinders

The function **rep_ir_distance** (Appendix B.8) converts the filtered voltages provided by **acq_ir_voltage** into distances. Two different techniques were considered for the voltage-distance software model – polynomials and lookup tables. Localisation information is extrapolated from the measured distances using **rep_ir_coord** (Appendix B.9).

4.2.2.1 Polynomial Model

Equation 4.10 gives a polynomial that fits relatively closely to the experimentally obtained voltage-distance points given in Figure 4.4. However, each rangefinder has a slightly different voltage-distance curve, so for best accuracy each rangefinder requires a unique polynomial. Also, even on the closest-fitting rangefinders, the polynomial begins to diverge from the measured curve when the range is small. This

is a significant problem, since at close range the risk of collision is highest, so this is where a rangefinder's accuracy is of utmost importance.

$$d_{IR} = \frac{A + BV_{IR}}{1 + CV_{IR} + DV_{IR}^{2}}$$ **Equation 4.10**

$A = 8.271 \times 10^{-5}$

$B = 9.369$

$C = -3.398$

$D = 17.339$

d_{IR}:   Infrared rangefinder distance (m)

V_{IR}:   Infrared rangefinder voltage (V)
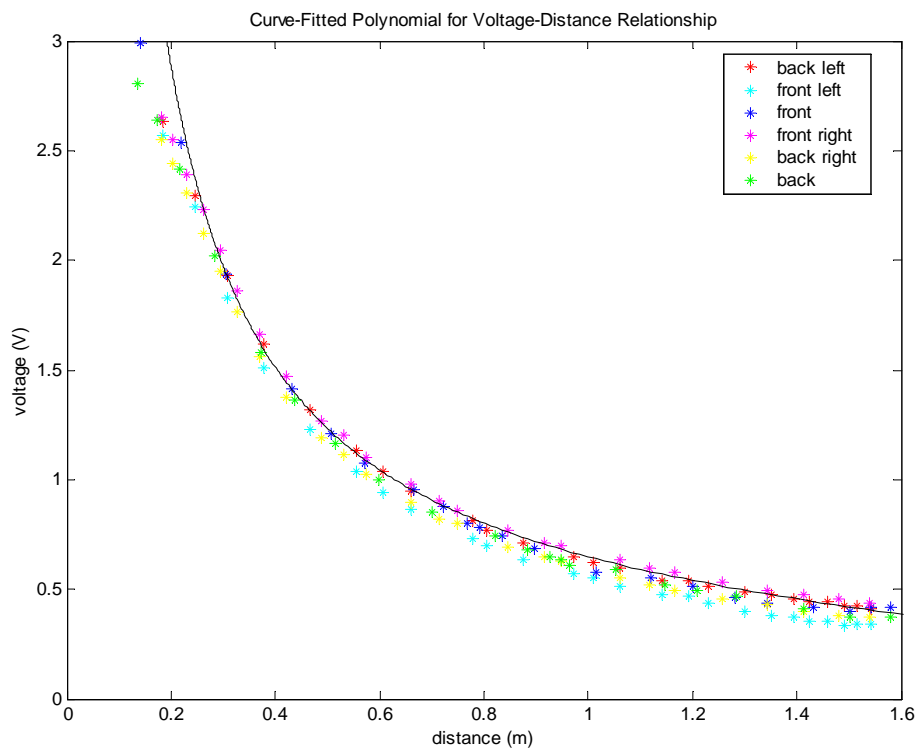
Figure 4.4: Polynomial Matched to Data

4.2.2.2 Lookup Table

Instead of using a single polynomial to model the entire voltage-distance relationship, this technique involves dividing the curve up into a series of straight lines. The upper and lower voltage limits of each line segment are recorded in a table (or, in the case of a programming language such as MATLAB, an array). Calculating distance from voltage then becomes a simple matter of determining which table entry contains the measured voltage, and applying the appropriate straight-line equation.

When no low-order polynomial exists that can closely match the data (as is the case for MARVIN's rangefinders), the lookup table results in a closer fit, and therefore greater accuracy. Conversely, this method can be computationally less efficient than the polynomial, especially if the curve is divided up into a large number of table entries. However, in this project, where more significant bottlenecks reside in other sections of code, the advantages of this technique outweigh its disadvantages.

The **rep_ir_distance** function stores a table of voltages for each rangefinder in the form of a single 28×6 matrix, with the corresponding distances stored in a 28×1 matrix. The resulting lookup table curves are plotted in Figure 4.5, along with the experimental data from which they were obtained.

The function determines the table entry containing the measured voltage using an incremental condition check. If the voltage is below the table's lowest entry (approximately 0.4 V, corresponding to a distance of 1.5 m), the distance is set to infinity, indicating that the measured object is out of range. If it is above the highest table entry (approximately 2.75 V, corresponding to a distance of 0.15 m), the distance is set to zero. Otherwise, the distance is calculated using Equation 4.11.
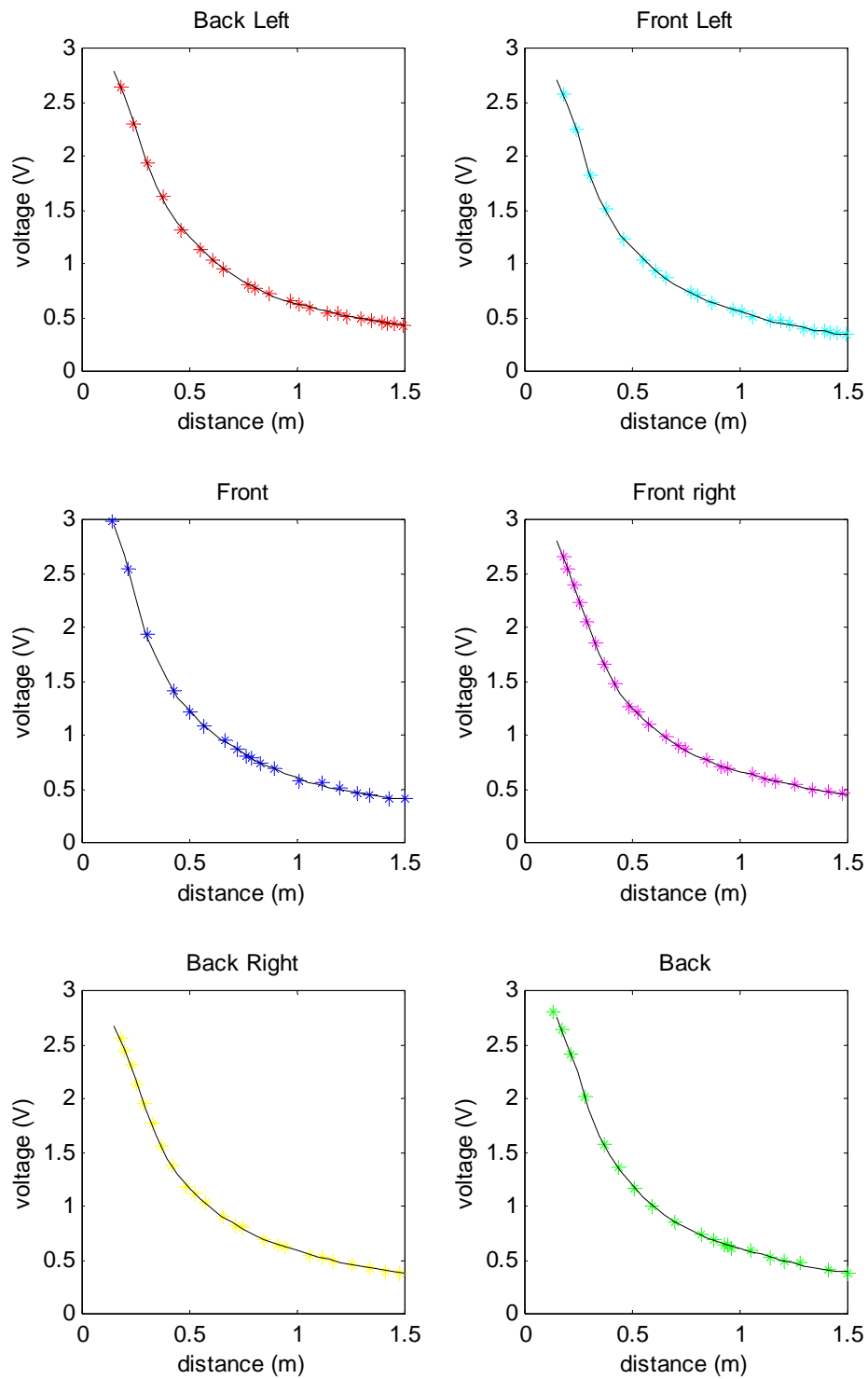
Figure 4.5: Lookup Table Curves Matched to Data

$$d_{IR} = d_{L1} + \frac{(V_{IR} - V_{L1})(d_{L2} - d_{L1})}{V_{L2} - V_{L1}}$$

Equation 4.11

V_{L1}:   Lookup table entry's lower voltage limit (V)

V_{L2}:   Lookup table entry's upper voltage limit (V)

d_{L1}:   Lookup table entry's lower distance limit (m)

d_{L2}:   Lookup table entry's upper distance limit (m)

4.2.2.3   Localisation Using Rangefinders

Ranges are converted into offset and heading information using the **rep_ir_coord** function. The function calculates offset by comparing measured wall distances with the expected position of each wall. Heading is derived from the relative distances measured by two or more adjacent rangefinders.

Prior to their use in this function, the rangefinder distances are filtered to eliminate transient signals caused by objects momentarily blocking the rangefinders. This helps to prevent MARVIN from reacting to people walking past in the corridor. Each range is compared with the mean average of the last ten values. If the difference is larger than 0.2 m, the range is not used in the offset and heading calculations.

The position and orientation of each rangefinder with respect to MARVIN's position and heading are recorded in a set of coordinate arrays. They are added to the offset and heading derived from the odometers to obtain coordinates relative to the corridor or room centre axis (Equations 4.12 and 4.13). These coordinates are used in Equation 4.14 to predict the wall distance that each rangefinder will measure, so that the algorithm can reject those rangefinders that are not facing towards a wall.

$$y_{IR} = y_M + x_{IR}'\cos(\theta_M) + y_{IR}'\sin(\theta_M)$$ **Equation 4.12**

$$\theta_{IR} = \theta_M + \theta_{IR(M)}$$ **Equation 4.13**

$$d_{IR(pr)} = \frac{y_W - y_{IR}}{\sin(\theta_{IR})}$$ **Equation 4.14**

y_{IR}:   Rangefinder offset (m)

θ_{IR}:   Rangefinder heading (m)

x_{IR}':   Rangefinder distance with respect to MARVIN (m)

y_{IR}':   Rangefinder offset with respect to MARVIN (m)

θ_{IR}':   Rangefinder heading with respect to MARVIN (rad)

y_W: Wall offset (m)

$d_{IR(pr)}$:   Predicted distance measured by rangefinder (m)

Equations 4.15 and 4.16 yield the coordinates of measured objects relative to MARVIN's position and orientation. An offset is obtained from each valid rangefinder distance using Equation 4.17, while a heading calculation requires valid distances from adjacent rangefinder pairs, as shown in Equation 4.18. The heading resulting from Equation 4.18 can be equal or opposite to the actual heading, so both possibilities are compared with the odometer-measured heading, and the closest match is selected. Figure 4.6 summarises the various parameters used in these equations.
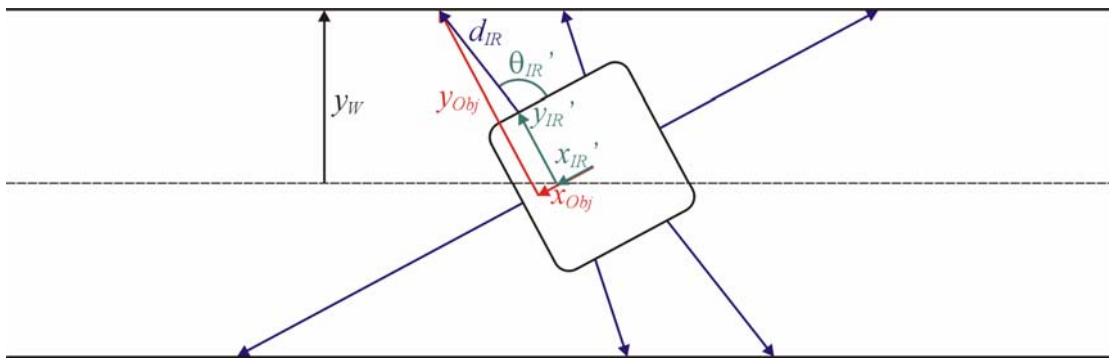
Figure 4.6: Obtaining Offset and Heading from Rangefinders

$$x_{ob} = x_{IR}' + d_{IR} \cos(\theta_{IR}')$$ **Equation 4.15**

$$y_{ob} = y_{IR}' + d_{IR} \sin(\theta_{IR}')$$ **Equation 4.16**

$$y_{M(IR)} = y_W - x_{ob} \sin(\theta_M) - y_{ob} \cos(\theta_M)$$ **Equation 4.17**

$$\theta_{M(IR)} = -\tan^{-1}\left(\frac{y_{ob2} - y_{ob1}}{x_{ob2} - x_{ob1}} \right)$$ **Equation 4.18**

x_{ob}:   Object's distance with respect to MARVIN (m)

y_{ob}:   Object's offset with respect to MARVIN (m)

$y_{M(IR)}$: MARVIN's offset calculated from rangefinder (rad)

$\theta_{M(IR)}$: MARVIN's heading calculated from rangefinder (rad)

Each offset and heading is allocated into one of two arrays according to which wall the rangefinder is facing. The mean average of each nonempty array is taken, yielding four values – an offset and heading for each wall. If an array is empty, the corresponding offset or heading is set to **NaN**, indicating that it should not be used in the sensor fusion algorithm.

This algorithm is flexible enough that it can be used for any number of rangefinders in any combination of positions and orientations where at least two rangefinders face each wall, as long as the appropriate values are recorded in the coordinate arrays.

4.2.3 Coordinate Transformations

The control system's internal representation of MARVIN's position and orientation is calculated relative to the centre axis, and the coordinates are reset when MARVIN moves into a room or section of corridor with a different centre axis. This is because the rangefinders' localisation algorithm requires the relative coordinate system to be aligned properly with the walls. However, the constant changes in reference frame result in difficulties in plotting MARVIN's internal representation for testing purposes. Also, the navigation system requires an absolute set of coordinates for its

internal map. Consequently, the control system transforms the relative coordinates into absolute coordinates for external use.

The coordinate transformation function **coord_trans** (Appendix B.10) applies axes rotations given in Equations 4.19 and 4.20. Using this function, absolute values are obtained for measured and target coordinates, and rangefinder origins. The **rel_coord** function (Appendix B.11) resets MARVIN's relative coordinates and updates their origin on the absolute coordinate axis whenever the centre axis angle changes.

$$x = x'\cos\alpha - y'\sin\alpha$$ **Equation 4.19**

$$y = x'\sin\alpha + y'\cos\alpha$$ **Equation 4.20**

α: Axes rotation angle (rad)

4.3 Sensor Fusion

Each of MARVIN's sensors provides useful data, but their individual importance varies with circumstance. For example, odometers are very accurate over short distances, but they are susceptible to cumulative error, which limits their long-term usefulness. Rangefinders are less accurate, but their error does not increase over time.

In order to minimise these problems, MARVIN utilises sensor redundancy – that is, multiple sensors providing the same information, but with different degrees of accuracy and precision. Overlapping sensor signals are combined, or *fused,* in a manner that takes advantage of each sensor's strengths and reduces its weaknesses.

Although this chapter concentrates on the fusion of overlapping data from different sensors, the term *sensor fusion* has a broader meaning that also encompasses non-redundant sensor signals and multiple samples from a single sensor. In general, a sensor data fusion algorithm consists of one or more of the following implementations [van Dam et al, 1999]:

- **Complementary** – Fusion of sensor data that does not overlap. Individually, each sensor produces only a partial model of the robot's state, and the complete model is assembled from the disparate components. For example, MARVIN's beacons provide distance information, while the rangefinders provide offset and heading. In combination, this yields a complete representation of MARVIN's position and orientation.

- **Competitive** – Fusion of independent, overlapping sensor data in order to reduce errors. This can involve data from different sensors measured at the same time, or different measurements carried out by the same sensor over time. Given enough sensor redundancies, competitive fusion can allow a robot to continue to function at a reduced capacity in the event of individual sensor failures. However, none of MARVIN's overlapping sensors can be fused in a strictly competitive manner, since unavoidable dependencies exist between them.

- **Cooperative** – Fusion of data from sensors that are dependent on each other. MARVIN's rangefinder localisation algorithm (Section 4.2.2.3) fits into this category, since individual rangefinders are selected or rejected in accordance with information from the odometers, and the rangefinders' offset and heading calculations are directly influenced by the odometers' heading measurement. The navigation system's beacon identification is also somewhat dependent on localisation information provided by the other sensors.

4.3.1 Sensor Fusion Techniques

A number of techniques were considered for MARVIN's sensor data fusion algorithm, including:

- Boolean Logic
- Dynamic Weighted Average
- Bayesian Inference
- Dempster-Shafer Inference
- Fuzzy Logic
- Neural Network

4.3.1.1 Boolean Logic

With this scheme the sensor with the greatest perceived accuracy in a given situation is used exclusively. All other sensors are ignored since their data is less likely to match the robot's real-world motion. This is the simplest algorithm to implement, but it discards a significant amount of useful data.

If used on MARVIN, the odometers would be favoured most of the time, since they are the most accurate sensors for short-term measurements. However, once their readings began to deviate from the real-world motion due to cumulative error, the odometer data would need to be reset using data from the rangefinders and beacons.

4.3.1.2 Dynamic Weighted Average

A weighted average allows each sensor to make a contribution to the internal model, but the sensors are still prioritised according to estimated uncertainties. Dynamic weights can also be assigned on a situational basis, providing the benefits of the purely Boolean logic, without its disadvantages.

On MARVIN, the odometer weights would be much higher than the other sensors, given their superior accuracy. Even with very low weights, the rangefinders would correct odometer errors over time. Selecting the exact rangefinder weights is simply a trade-off between accuracy and speed of response. In situations where the rangefinder data is misleading – when MARVIN passes a corridor intersection or an open door, for example – the weights can be temporarily zeroed.

4.3.1.3 Bayesian Inference

In this implementation sensors and their uncertainties are represented as probability density functions (often Gaussian distributions, but other functions can also be used). Given two overlapping probability density functions, a function for the fused sensors can be determined using Equation 4.21 [van Dam et al, 1999].

$$p(z \mid r_1, r_2) = \frac{p(z \mid r_1).p(z \mid r_2)}{p(z)}$$
  Equation 4.21

p: Probability density function

z: Common internal representation

r_1, r_2:   Raw sensor data from independent sensors

This technique provides a more structured approach than simpler techniques such as the weighted average. However, it is not applicable between the two main sensors utilised in MARVIN's sensor fusion algorithm – the odometers and rangefinders – because they are not independent.

4.3.1.4   Other Techniques

- **Dempster-Shafer Inference** – Dempster-Shafer theory is an extension of Bayesian inference. It allows probabilities to be applied to groups of states (e.g. the sensor detects an object that is likely either "A" or "B") and unknown states (e.g. the object detected is likely undefined) [Wu et al, 2002].

- **Fuzzy Logic** – Unlike Bayesian and Dempster-Shafer techniques, fuzzy logic does not depend upon rigid probabilities. Sensors are assigned membership to sets whose boundaries are loosely defined, and may overlap. They can be prioritised according to intuitive concepts such as "possibly," "probably" and "definitely" [Godjevac, 1995].

- **Neural Network** – A learning algorithm can be trained with the sensor measurements as inputs and the internal model as the output. This can potentially result in a more intuitive interpretation of the data than traditional techniques [van Dam et al, 1996].

4.3.1.5   Selected Implementation: Dynamic Weighted Average

For a complex system with a large number of sensors, high-level techniques such as neural networks can produce the most reliable localisation data. However, for a system such as MARVIN with relatively few sensors and a comparatively simple operating environment, they do not provide enough of an improvement to justify the complexity of their implementation, and the increased CPU overhead that would result from their use. MARVIN's sensor data fusion algorithm utilises a form of dynamic weighted average, the simplest technique that can be used without sacrificing performance.

4.3.2 MARVIN's Implementation

The beacons are utilised in conjunction with the navigation system's internal map, so they are not included in the sensor fusion algorithm detailed in this thesis. Instead, the raw beacon data is passed directly to the navigation system.

The **sensor_fusion** function (Appendix B.12) corrects the odometers' offset and heading data using similar data obtained from the rangefinders. Each corrected value is a weighted average of three inputs – the original odometer input and a separate rangefinder input for each wall.

Rangefinder weights are comprised of two factors. The first of these, assigned inside **sensor_fusion**, determines the maximum weighting that can be applied to a rangefinder offset or heading. This value remains constant throughout normal operation, but it is increased during initialisation, so that MARVIN's initial offset and heading can be calibrated from the measured wall positions.

The second factor is received from the navigation system, which uses its internal map to decide when to utilise the rangefinders for localisation, and when to ignore them. This prevents errors from arising when MARVIN passes an open door, corridor intersection or change in corridor wall position. A separate factor is assigned for each wall, so a disturbance on one side does not disrupt the other (which is the reason for deriving two independent sets of rangefinder localisation data).

MARVIN's sensor data fusion algorithm can be extended to incorporate additional sensors, such as a compass (which would provide an absolute heading reference) and a laser rangefinder (which would provide localisation information similar to that produced by the infrared rangefinders). Once the new sensor data is converted into MARVIN's distance-offset-heading representation it can be assigned a weighting in the same manner as the other sensors.

5 Motor Control Software

If a robot's sensors are its eyes and ears, its actuators are its muscles. Actuators provide physical motion, allowing a robot to react to the environment observed by its sensors. MARVIN's primary actuators are its two driving motors, which are used to control its overall velocity, position and orientation. The motor control system consists of the following steps:

- **Motion Planning** – Translating a motion instruction into efficient velocity profiles and trajectory.
- **Heading Control** – Controlling the heading in order to maintain the trajectory.
- **Velocity Control** – Controlling wheel velocities in order to maintain the heading and velocity profiles.
- **Collision Avoidance** – Reacting to obstacles in order to avoid collisions.
- **Driving Motors** – Supplying power to the motors in order to drive them at the intended velocities.

5.1 Motion Planning

Motion instructions delivered by the navigation system primarily consist of a distance and angle input. From this, the control software generates a target trajectory for MARVIN to travel, and a velocity profile for each wheel. The target trajectory provides a reference against which MARVIN's position and orientation can be compared, allowing the control system to dynamically correct any deviations that occur. Similarly, wheel velocities are compared against the velocity profiles in order to provide smooth acceleration and deceleration and maintain their ratio at the value necessary to produce the intended motion.

5.1.1 Generating Target Trajectory

The function **gen_tgt_trj** (Appendix B.13) translates a given distance/angle instruction into a set of coordinate arrays – distance, offset and heading – representing MARVIN's intended position and orientation at regular intervals along the trajectory. The trajectory is characterised as a series of straight lines whose end points are given by these coordinates. This representation only approximates curved trajectories, but it provides a large degree of flexibility in the trajectories that can be generated.

For moving turns, the radius of the circular path (Figure 5.1) is given by Equation 5.1. The heading at each point along the trajectory is calculated in Equation 5.2. Distance and offset are then obtained using Equations 5.3 and 5.4. The sign of the ± term of each of these equations is selected according to the direction of the given instruction. Straight-line trajectories are obtained using Equations 5.5 and 5.6.
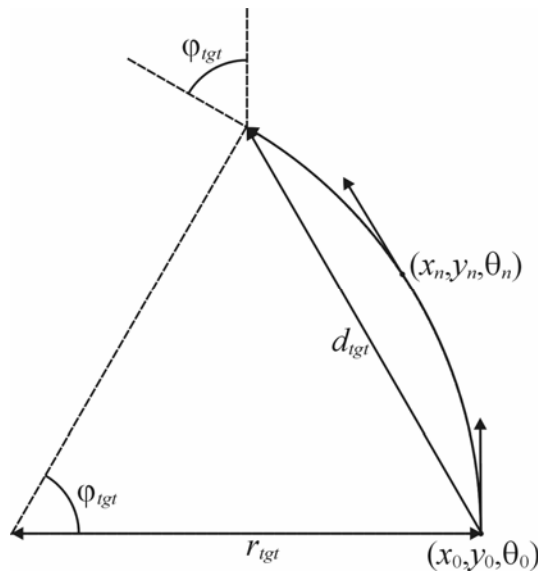
Figure 5.1: Circular Trajectory from Distance and Angle Inputs

Circular Trajectories

$$r_{tgt} = \frac{|d_{tgt}|}{\sqrt{2(1-\cos(\varphi_{tgt}))}}$$

Equation 5.1

$$\theta_n = \theta_0 + \frac{n}{N}\varphi_{tgt}$$

Equation 5.2

$$x_n = x_0 \pm r_{tgt}(\sin(-\theta_0) + \sin(\theta_n))$$

Equation 5.3

$$y_n = y_0 \pm r_{tgt}(\cos(-\theta_0) - \cos(\theta_n))$$

Equation 5.4

Straight Trajectories

$$x_n = x_0 + \frac{n}{N}d_{tgt}\cos(\theta_0)$$

Equation 5.5

$$y_n = y_0 + \frac{n}{N}d_{tgt}\sin(\theta_0)$$

Equation 5.6

d_{tgt}:   Distance between initial and target positions (m)

φ_{tgt}:   Target angle to turn through (rad)

r_{tgt}:   Radius of circular target trajectory (m)

n: Target trajectory coordinate element number

N: Total number of target trajectory coordinate elements

θ_0:   Initial heading (rad)

θ_n: Heading element (rad)

x_0:   Initial distance (m)

x_n: Distance element (m)

y_0:   Initial offset (m)

y_n: Offset element (m)

If an instruction results in a target trajectory that passes too close to a wall, the trajectory is clipped to prevent potential collisions. Any part of the target trajectory that exceeds the allowable offset range is converted into a straight-line trajectory parallel to the centre axis.

5.1.2 Generating Velocity Profile

The **wheel_pos** function (Appendix B.14) calculates the target wheel position from the arc-length that each wheel must travel in order for MARVIN to correctly execute the given instruction. This function carries out the inverse of the calculations in Section 4.2.1.3. Equation 4.6 is rearranged to obtain a central arc-length from the given straight-line distance, while the individual wheel arc-lengths are obtained using Equations 5.7 and 5.8, which are derived from Equations 4.4 and 4.5.

$$l_L = l_M + \frac{w\varphi_M}{2} \qquad\qquad \textbf{Equation 5.7}$$

$$l_R = l_M - \frac{w\varphi_M}{2} \qquad\qquad \textbf{Equation 5.8}$$

The target wheel arc-lengths and the velocity limit are utilised in **gen_vel_prof** (Appendix B.15) to generate a velocity profile for each wheel. This algorithm operates in the distance domain rather than the time domain, because the most important goal is to drive MARVIN to the intended location, whereas the time taken to get there is less important. Velocity profiles are represented as arrays of velocities and distances, which can be considered a form of lookup table.

The first task of this function is to obtain the maximum velocity that the wheels can be safely driven at for a given instruction. This value is highest for straight-line instructions and lowest for stationary turns.

In order to maintain a straight trajectory, the wheel velocities should be equal at all times. For stationary turns they should be equal in magnitude and opposite in direction. For circular trajectories the wheel velocities should be proportional to each other, with the proportionality constant for the slower wheel equal to the ratio of the target wheel positions. In each case the wheels must arrive at their target positions simultaneously.

The step response of MARVIN's wheels (in the time domain) is of the form shown in Figure 5.2. However, in order to avoid wheel slippage, and to ensure that one wheel does not accelerate or decelerate faster than the other (which would result in heading errors), acceleration should be limited to less than the maximum obtainable value. For maximum convenience, MARVIN's velocity profiles are of the linear form given in Figure 5.3.
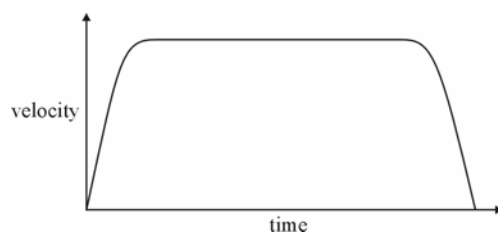
Figure 5.2: Wheel Step Response
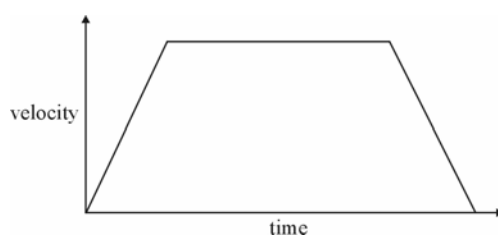
Figure 5.3: Velocity-Time Profile

This form can be divided into up to three sections – acceleration, constant velocity and deceleration. Though linear in time, the velocity profile is modelled in the distance domain where the acceleration and deceleration sections are not linear. The first step to developing the model is to obtain the position where the acceleration and deceleration curves would intersect if no upper velocity limit were present (as shown in Figure 5.4), using Equation 5.10. The corresponding velocity is calculated using Equation 5.9. If this velocity is below the upper velocity limit, the velocity profile will be approximately triangular. Otherwise it will be approximately trapezoidal, and the positions where the constant velocity line intersects with the acceleration and deceleration curves are calculated using Equation 5.11 and 5.12, respectively. In each case Equation 5.9 is used to calculate velocities from positions, given the various substitutions shown in Table 5.1.
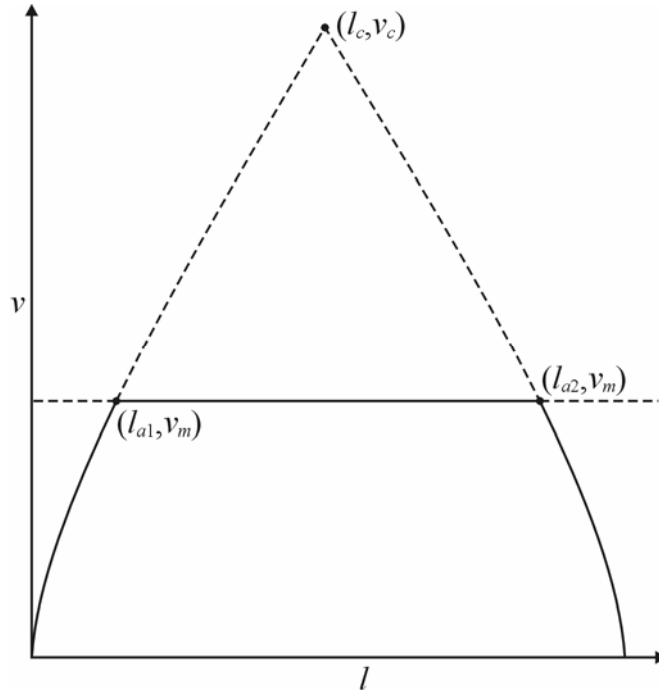
Figure 5.4: Velocity-Distance Profile

$$v = \sqrt{v_0^2 + 2a(l - l_0)}$$ **Equation 5.9**

$$l_c = \frac{v_i^2 + 2a_2 l_f}{2(a_2 - a_1)}$$ **Equation 5.10**

$$l_{a1} = \frac{v_m^2 - v_i^2}{2a_1}$$ **Equation 5.11**

$$l_{a2} = l_f + \frac{v_m^2}{2a_2}$$ **Equation 5.12**

Table 5.1: Substitutions for Equation 5.9

Original	Substitution if $v_c \le v_m$		Substitution if $v_c > v_m$		
	$l \le l_c$	$l > l_c$	$l < l_{a1}$	$l_{a1} \le l \le l_{a2}$	$l > l_{a2}$
v_0	v_i	v_c	v_i	v_m	v_m
a	a_1	a_2	a_1	0	a_2
l_0	0	l_c	0	l_{a1}	l_{a2}

l_f: Final position (m)

v_i: Initial velocity (m/s)

v_m: Upper velocity limit (m/s)

a_1:   Acceleration (m/s^2)

a_2:   Deceleration (m/s^2)

v_c: Velocity at acceleration-deceleration intersection (m/s)

l_c: Position at acceleration-deceleration intersection (m)

l_{a1}:   Position at acceleration-upper velocity limit intersection (m)

l_{a2}:   Position at deceleration-upper velocity limit intersection (m)

The final task that this function performs is to update the direction flags for each wheel. Ordinarily this would be unnecessary since wheel velocities are recorded as signed values. However, the microcontroller returns an error if the wheel direction bit is inverted while the current PWM magnitude is nonzero, even if the new magnitude is zero. Without the direction flags it would not be possible to indicate the sign of a zero velocity instruction.

5.2 Control Theory

Two control loops are implemented in MARVIN's control algorithm. The outer loop, heading control (Section 5.3), adjusts target wheel velocities so that MARVIN is always facing the direction necessary to follow the target trajectory. The inner loop, velocity control (Section 5.4), ensures that the wheels are driven at the target velocities. Several alternative control techniques were considered for these control loops:

- PID
- Fuzzy Logic
- Neural Networks
- Neuro-Fuzzy

5.2.1 PID

PID control utilises three different feedback elements – Proportional, Integral, and Derivative – to produce an output (called the *control variable*) that depends on the tracking error between a target value (called the *set point*) and a measured value (called the *process variable*).

With proportional control (Equation 5.13) the control variable is proportional to the error. Proportional control is generally fast and stable (for low proportional gains), but it results in a steady-state offset error. Increasing the proportional gain can reduce the offset error, but it also increases the system's instability. In many systems the stability requirements limit the proportional gain to a value that yields an unacceptably large offset error.

$$u(t) = Ke(t)$$
<div align="right">**Equation 5.13**</div>

$u(t)$:   Control variable

$e(t)$:   Tracking error

K: Proportional gain

Integral control (Equation 5.14) involves the summation of errors over time. Unlike proportional control, it does not produce an offset error, but it is slower to reach steady state than proportional control. In most control algorithms the integral sum should be limited to prevent integrator windup, an effect where it can increase indefinitely when the system is in saturation.

$$u(t) = \frac{K}{T_I} \int_0^t e(\eta)d\eta$$
<div align="right">**Equation 5.14**</div>

T_I: Integral time.

Derivative control (Equation 5.15) is dependent on the rate of change of error. It is generally faster than both proportional and integral control, so the derivative element

of PID control is primarily responsible for a system's speed of response. Derivative control greatly amplifies noise, so it is usually applied only to filtered signals.

$$u(t) = KT_D\dot{e}(t)$$ **Equation 5.15**

T_D: Derivative time.

Combining the three control elements allows a system to exploit their advantages, and eliminate their disadvantages. Equations 5.13-5.15 are added together to form Equation 5.16, which represents the entire PID control system. This yields the transfer function (in the Laplace domain) given in Equation 5.17.

$$u(t) = K\left(e(t) + \frac{1}{T_I}\int_0^t e(\eta)d\eta + T_D\dot{e}(t) \right)$$ **Equation 5.16**

$$D(s) = \frac{U(s)}{E(s)} = K\left(1 + \frac{1}{T_I s} + T_D s \right)$$ **Equation 5.17**

5.2.2 Fuzzy Logic

Fuzzy logic involves the use of qualitative reasoning instead of purely quantitative measurements. It is particularly useful when dealing with systems that are ill-defined or difficult to model. Like Boolean logic, fuzzy logic involves selecting an action if a set of conditions is satisfied. However, rather than deciding whether the conditions are true or false, fuzzy logic estimates each condition's degree of truth (generally a number between 0 and 1), and calculates output values based on the relative estimates.

Process variables are assigned membership to one or more *fuzzy sets*, which are labelled with qualitative descriptions such as "small", "medium" and "large". The degree of truth of a given set is calculated from its *membership function*, which can be a number of shapes, including triangular, trapezoidal and Gaussian. Various techniques can be used to obtain an output value from the membership functions. One

of the simplest methods is to apply a weighted average of the target value for the relevant fuzzy sets. Weights can be derived from their relative degrees of truth.

5.2.3 Neural Network

A neural network utilises simplified models of biological neurons in an attempt to simulate the adaptive processes that occur in the brain. The main advantage of a neural network is its ability to learn through training instead of requiring the developer to design a hard-coded algorithm.

A model for an artificial neuron is given in Figure 5.5. A weighted sum of each of the neuron's inputs is calculated, a *threshold*, θ, is subtracted and the result is fed into an *activation function*, f(a). Activation functions that are commonly utilised include step, ramp, sigmoid and Gaussian functions. The neuron's output may represent the control variable, or it may pass to another neuron, where the process is repeated. Training a neural network is generally accomplished by adjusting the weights until the output of the network matches the target output.
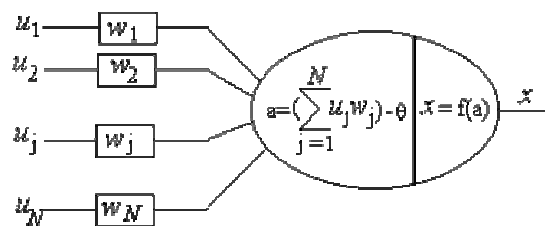
Figure 5.5: Artificial Neuron

5.2.4 Neuro-Fuzzy

One of the main disadvantages of fuzzy control systems is the lack of a systematic design approach. Tuning these systems is a time-consuming process. One solution to this problem is to use neural networks to tune a system automatically. These self-tuning controllers can be developed faster than conventional systems, and they often provide better performance [Godjevac, 1995].

5.2.5 Selected Implementation: PID

The simplest and most widely used technique, PID control, is selected due to its ease of implementation. Although a PID control system does not necessarily provide the best response characteristics, it is sufficient for the purposes of this project. The same discrete PID algorithm is utilised by each of the control loops, but they are tuned independently, resulting in different control constants.

The digital PID control algorithm is derived from Equation 5.16 in its differential form (Equation 5.18). This is approximated in the discrete domain using Euler's method, resulting in Equation 5.19, which can be implemented directly in MATLAB.

$$\dot{u}(t) = K\left(\dot{e}(t) + \frac{1}{T_I} e(t) + T_D \ddot{e}(t) \right) \qquad \textbf{Equation 5.18}$$

$$u(k) = u(k-1) + K\left[\left(1 + \frac{T}{T_I} + \frac{T_D}{T} \right) e(k) - \left(1 + 2\frac{T_D}{T} \right) e(k-1) + \frac{T_D}{T} e(k-2) \right]$$

$$\textbf{Equation 5.19}$$

In order to prevent integrator windup, upper and lower thresholds can be imposed on the control variable so that the integral sum does not add to the control variable if it is already at the maximum value that can have an effect on the system. Alternatively, the three control elements can be singled out, with separate limits imposed on each. Separating out the components also allows the algorithm to assign different levels of

filtering to each element's error signal. Thus, noise can be reduced on the derivative element's error signals without adversely affecting the speed of the proportional and integral components.

5.3 Heading Control

MARVIN's heading control algorithm is implemented in three steps. Firstly, uncorrected target wheel velocities are obtained using the velocity profiles. Then a heading error is obtained, representing the difference between the measured heading and the heading necessary to maintain the correct trajectory. Finally, the heading error is utilised in conjunction with the uncorrected target velocities to produce a set of velocity errors for each wheel.

5.3.1 Uncorrected Target Wheel Velocities

The uncorrected target velocity for each wheel is obtained from the velocity profile using **tgt_velocity** (Appendix B.16). This function applies an algorithm similar to that used to obtain rangefinder distances from the lookup table described in Section 4.2.2.2. The velocity for a given distance is calculated from the velocity profile lookup table using Equation 5.20.

$$v = v_{L1} + \frac{(l - l_{L1})(v_{L2} - v_{L1})}{l_{L2} - l_{L1}}$$

Equation 5.20

l_{L1}:   First velocity profile position (m)

l_{L2}:   Second velocity profile position (m)

v_{L1}:   First velocity profile velocity (m/s)

v_{L2}:   Second velocity profile velocity (m/s)

Two minimum velocity thresholds are required to start the wheels moving correctly, and to prevent them from stopping prematurely. The rising velocity threshold

represents the velocity that the wheels must be driven at in order to overcome static friction and start moving. The falling velocity threshold represents the minimum velocity that the wheels can be driven at once they are in motion. Thus, the rising velocity threshold is applied during the acceleration phase of the velocity profile, and the falling velocity threshold is applied at all other times, unless the target position has been reached. The nature of the velocity control system means that these thresholds can be lower than the physical thresholds of the system, but they must be nonzero to ensure smooth operation.

5.3.2 Heading Error

The target trajectory provides a reference that is used to ensure that MARVIN's position and orientation coordinates are as close as possible to the correct values at all times. If MARVIN begins to drift off course, it becomes necessary to adjust its heading in order to return to the correct path.

The **heading_error** function's (Appendix B.17) first task is to locate the target point – the point on the target trajectory that is closest to the measured position. Equation 5.21 is intended for instructions that involve a significant physical displacement (i.e. straight line motion, or moving turns where the centre of rotation lies outside MARVIN's perimeter). It utilises the measured position in conjunction with each set of coordinates comprising the target trajectory to obtain the separation distance for each point. For instructions where rotation is more easily measured than displacement (i.e. stationary turns), the heading separation (Equation 5.22) is calculated instead.

$$d_n = \sqrt{(x_n - x)^2 + (y_n - y)^2}$$
   Equation 5.21

$$\varphi_n = |\theta_n - \theta|$$
   Equation 5.22

d_n: Separation distance for target trajectory element n (m)

φ_n: Separation angle for target trajectory element n (rad)

The elements that produce the two smallest separation distances or separation angles are obtained using the library function **min**. The corresponding coordinates form the endpoints of a line that is orthogonal to the line connecting the measured coordinates to the target point, as shown in Figure 5.6. The point of closest approach is the intersection between these two lines, as calculated in Equations 5.23-5.26. This point of intersection may be outside the two endpoints if MARVIN is outside the range of the target trajectory. Certain calculations require that the target point be inside the target trajectory, while this limitation causes problems with other calculations. Thus, two sets of coordinates are generated – one for each of the conflicting requirements.
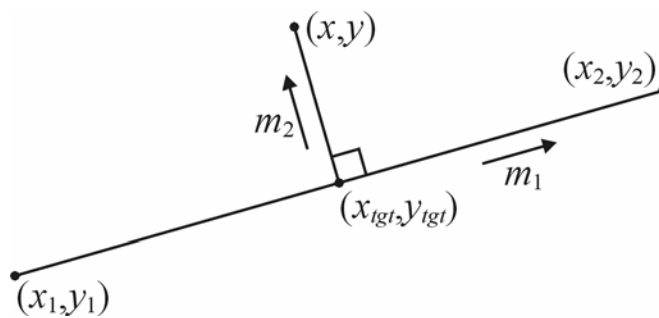
Figure 5.6: Closest Point on Target Trajectory

$$m_1 = \frac{y_2 - y_1}{x_2 - x_1}$$
<div align="right">**Equation 5.23**</div>

$$m_2 = \frac{x_1 - x_2}{y_2 - y_1}$$
<div align="right">**Equation 5.24**</div>

$$x_{tgt} = \frac{m_1 x_1 - m_2 x + y - y_1}{m_1 - m_2}$$
<div align="right">**Equation 5.25**</div>

$$y_{tgt} = \frac{m_1 y - m_2 y_1 + x - x_1}{m_1 - m_2}$$
<div align="right">**Equation 5.26**</div>

m_1:   Gradient of line connecting two points on target trajectory

m_2:   Gradient of line connecting measured position to target point

x_{tgt}:   Target distance coordinate (m)

y_{tgt}:   Target offset coordinate (m)

Because the target trajectory is divided into straight lines, the target position coordinates are only approximations of the ideal value, but the resolution of the target trajectory is high enough to ensure sufficient accuracy. These coordinates yield two important values: separation heading and separation distance. The separation heading is the heading that MARVIN must obtain in order to move towards the target position. It is calculated using the four-quadrant inverse tangent function **atan2** in conjunction with Equation 5.27. The separation distance – the distance between the actual and target positions – is calculated using Equation 5.28.

$$\theta_s = \tan^{-1}\left(\frac{y_{tgt} - y}{x_{tgt} - x}\right)$$ **Equation 5.27**

$$d_s = \sqrt{(x_{tgt} - x)^2 + (y_{tgt} - y)^2}$$ **Equation 5.28**

θ_s: Separation heading (rad)

d_s: Separation distance (m)

The third important value for heading error calculations is the target heading – the heading that corresponds to the target position on the trajectory. Like the target position, the target heading derived in this algorithm is not equal to the ideal value. It is approximated as a weighted average of the endpoint headings, with the weighting derived from the position of the target point relative to the endpoint positions, as shown in Equations 5.29-5.31. A weighted average of two angles is not as simple as the equivalent calculation for ordinary numbers, because angles overflow after a single rotation. The function **average_angle** (Appendix B.18) was created for this purpose – it applies the necessary modifiers so that the angles being averaged are in the same range.

$$p_1 = \sqrt{(x_{tgt} - x_1)^2 + (y_{tgt} - y_1)^2}$$ **Equation 5.29**

$$p_2 = \sqrt{(x_{tgt} - x_2)^2 + (y_{tgt} - y_2)^2}$$ **Equation 5.30**

$$\theta_{tgt} = \frac{p_1\theta_1 + p_2\theta_2}{p_1 + p_2}$$ **Equation 5.31**

p_1:   Position of target point relative to first endpoint (m)

p_2:   Position of target point relative to second endpoint (m)

θ_{tgt}:   Target heading coordinate (rad)

To correct heading errors for reverse motion along the trajectory, MARVIN needs to turn in the opposite direction from that required for forward motion. Consequently, if MARVIN is travelling in reverse, the target heading coordinate is folded over an axis formed by the measured heading.

An ideal trajectory to follow in order to correct position and heading errors is of the form given in Figure 5.7. This trajectory is achieved by incrementally adjusting MARVIN's heading error, which is the difference between the intended heading and the measured heading. The intended heading at a given time is a weighted average of the target heading and separation heading, with the weights proportional to the separation distance. Thus, if MARVIN is far from its target position, the separation heading is given higher priority than the target heading so that the position error can be reduced quickly. However, if MARVIN's position is approximately correct, its heading must be maintained at the correct value in order to prevent it from drifting away from the target trajectory – in this situation the target heading is given higher priority than the separation heading. Equation 5.32 gives the overall heading error calculation. The three most recent heading errors are output for the PID control algorithm to utilise.
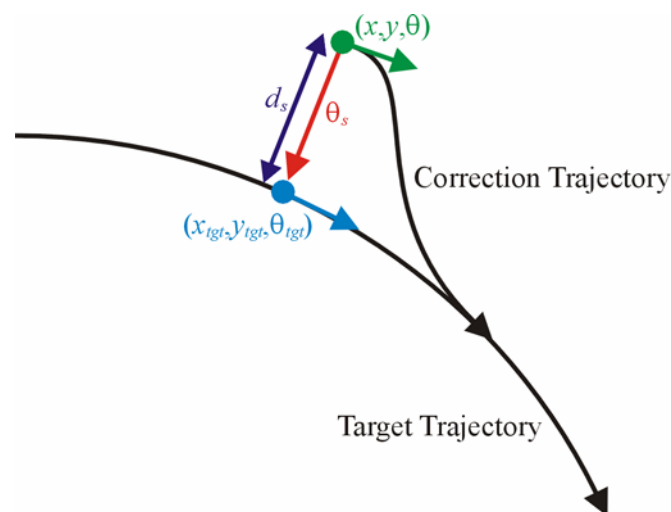
Figure 5.7: Reacquiring Target Trajectory

$$\theta_e = K_s d_s \theta_s + (1 - K_s d_s)\theta_{tgt} - \theta \qquad\qquad \textbf{Equation 5.32}$$

θ_e: Heading error (rad)

K_s: Proportionality constant for separation distance (0.5 m^{-1})

The final variable produced by this function is a value representing the proportion of the trajectory that MARVIN has covered at a given time (Equation 5.33). If certain parameters exceed their predefined thresholds, MARVIN is halted prematurely by setting the proportion to 1. This indicates that MARVIN is unable to complete the instruction accurately, so it awaits a new instruction from the navigation system.

$$T_p = \frac{n(p_1 + p_2) + p_1}{N(p_1 + p_2)} \qquad\qquad \textbf{Equation 5.33}$$

T_p: Proportion of trajectory covered

n: First trajectory element

N: Total number of trajectory elements

5.3.3 PID Heading Control

The function **heading_control** (Appendix B.19) adjusts the target wheel velocities by applying a PID control algorithm to the heading error information. A modifier representing the fractional change in wheel velocities required to correct a given heading error is obtained using the basic PID algorithm given in Equation 5.19. The heading errors are relatively clean signals, so no additional filtering is necessary for the derivative control element. Consequently, the control algorithm is implemented as a single equation.

The modifier is converted into a multiplication factor that is applied to a single wheel, reducing its velocity to the value required to implement the heading correction, as shown in Table 5.2. It is limited to a magnitude of 1 or less, preventing the wheel from reversing direction, and protecting the system from integrator windup.

Table 5.2: Wheel Velocity Multiplication Factors

	Left wheel	Right wheel
modifier < 0	1+*modifier*	1
modifier ≥ 0	1	1-*modifier*

The heading control system is tuned through experimentation using the simulation (Section 5.7). An appropriate proportional gain is obtained that results in a reasonable response time with minimal oscillation. The integral time is adjusted to reduce the steady-state offset from the target trajectory (Figure 5.8). Finally, the derivative time is adjusted to a value that improves the response time without adversely affecting system stability.
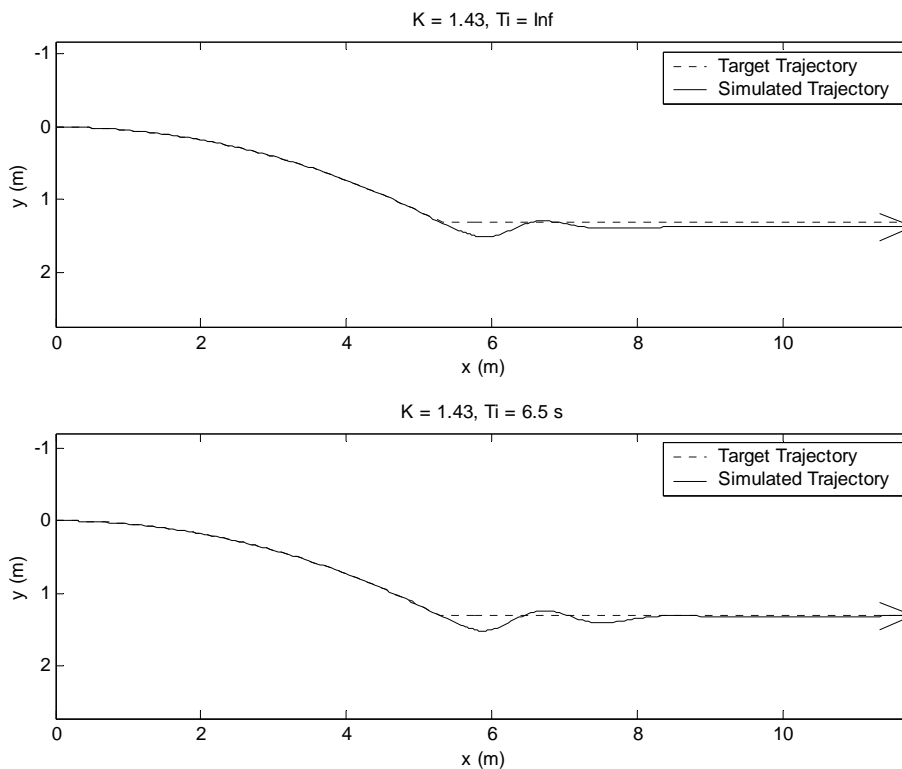
Figure 5.8: Using Simulation to Tune Heading Control System

The optimal low-velocity control gains result in instability at high velocities, so the proportional gain is not kept constant over the full range. Instead, the high-velocity gain becomes inversely proportional to the target velocity.

Limits are imposed on the wheels' acceleration in order to maintain stability and minimise wheel slippage. Two different measures of acceleration are limited: target acceleration and real acceleration. Target acceleration is derived from the change in target velocity since the last control cycle. Real acceleration involves the difference between the measured velocity and the target velocity. The limit for real acceleration is less stringent than that for target acceleration, given the amount of noise on the real velocity measurement. Thus, the target acceleration limit is favoured during normal operation, and the real acceleration limit is only imposed if an error or external disturbance causes the measured velocity to diverge significantly from the target velocity.

Once the target velocities are finalised, **heading_control** produces a set of velocity errors representing the differences between each wheel's measured velocities and target velocities for the last three control cycles.

5.4 Velocity Control

As long as the velocity-PWM relationships are modelled accurately, the measured wheel velocities will closely match the target velocities under normal conditions. However, external disturbances such as uneven floors affect the loading experienced by the wheels, which in turn have a significant impact on the measured velocities. Rather than waiting until a disturbance causes MARVIN to drift off course before correcting it, a second PID control loop monitors and corrects the wheel velocities directly.

The PID algorithm given in Equation 5.19 is applied to the target wheel velocities in the function **velocity_control** (Appendix B.20). The proportional and integral elements are applied directly to the control errors produced by the **heading_control** function, but derivative control is applied only to filtered errors (which are obtained from the filtered velocities described in Section 4.2.1) in order to reduce the destabilising effects of noise.

Upper and lower velocity thresholds are imposed to prevent the wheel velocities from exceeding safety limits or changing direction while in motion, as well as protecting against integrator windup. The difference between applied velocity and measured velocity is also limited for safety reasons. This prevents the applied velocity from ramping up indefinitely if the odometers or motor drivers fail, or if the wheels are obstructed.

The velocity control algorithm must be tightly tuned, since it is the primary factor that limits the performance of the heading control algorithm. Tuning the velocity control loop is accomplished in approximately the same manner as for heading control: the proportional gain, integral time and derivative time are selected experimentally in order to provide a satisfactory trade-off between stability, offset and speed of convergence. Once preliminary tuning is complete, the two control loops are tested in combination, and final adjustments are made to each.

5.5 Collision Avoidance

The control system's collision avoidance scheme halts MARVIN's forward motion in the event of an impending (or actual) collision, and awaits further instructions from the navigation system. It does not attempt to plot a course around the obstacle, since the navigation system is better suited to this task.

Impending collisions are grouped into three levels of threat according to the proximity of a measured obstacle. The first level is implemented in the **heading_error** function (Section 5.3.2, Appendix B.17). If the rangefinder facing the direction of motion (i.e. the front sensor for forward motion, and the rear sensor for reverse motion) detects an obstacle within 0.8 m, and the trajectory crosses the measured obstacle, the instruction ends prematurely. Due to various acceleration limits imposed on the control system, MARVIN will decelerate smoothly. This prevents wheel slippage and protects the motor drivers from the current surges that accompany rapid changes in applied power.

The two higher threat levels are implemented in the **stop_wheel** function (Appendix B.21). Medium level collision avoidance is implemented if the rangefinder facing the direction of motion measures an obstacle within a range of 0.4 m, or if the navigation sends a stop instruction (by setting the distance and angle inputs to zero). In this situation the target velocity is set to zero. However, the microcontroller contains its own acceleration limits that prevent the PWM duty cycles from decreasing too quickly.

If one or more of the contact sensors detect a collision, or if the navigation system sends an emergency brake instruction, the brake flag is set, which instructs the microcontroller to ignore its acceleration limits and stop the wheels immediately. The strain on the motor drivers is preferable to the damage that would result from a collision.

5.6 Driving Motors

The target velocity must be converted into a value representing the PWM duty cycle that would drive the wheels at that velocity. The microcontroller represents this PWM value as an 8-bit integer, so the duty cycle is given by Equation 5.34.

$$D = \frac{P}{255}$$
 Equation 5.34

D: PWM duty cycle
P: PWM value

The microcontroller is configured to limit the PWM to half duty cycle (or a value of 128) to prevent high-speed collisions. The duty cycle limit also protects the motor drivers from excessive current surges during acceleration and deceleration. This precaution is doubly necessary because three separate thesis projects would be compromised if MARVIN were seriously damaged.

5.6.1 Velocity-PWM Relationship

If a given PWM value does not drive a wheel at the target velocity, the PID control system compensates by adjusting it until the velocity is correct. However, the wheel will settle on the correct velocity more slowly if the mismatch between PWM value and target velocity is large, so for optimal performance a close match under normal conditions is necessary. In order to obtain the velocity-PWM relationships for each wheel, a range of step responses are recorded (refer to Figure 5.2 for samples).
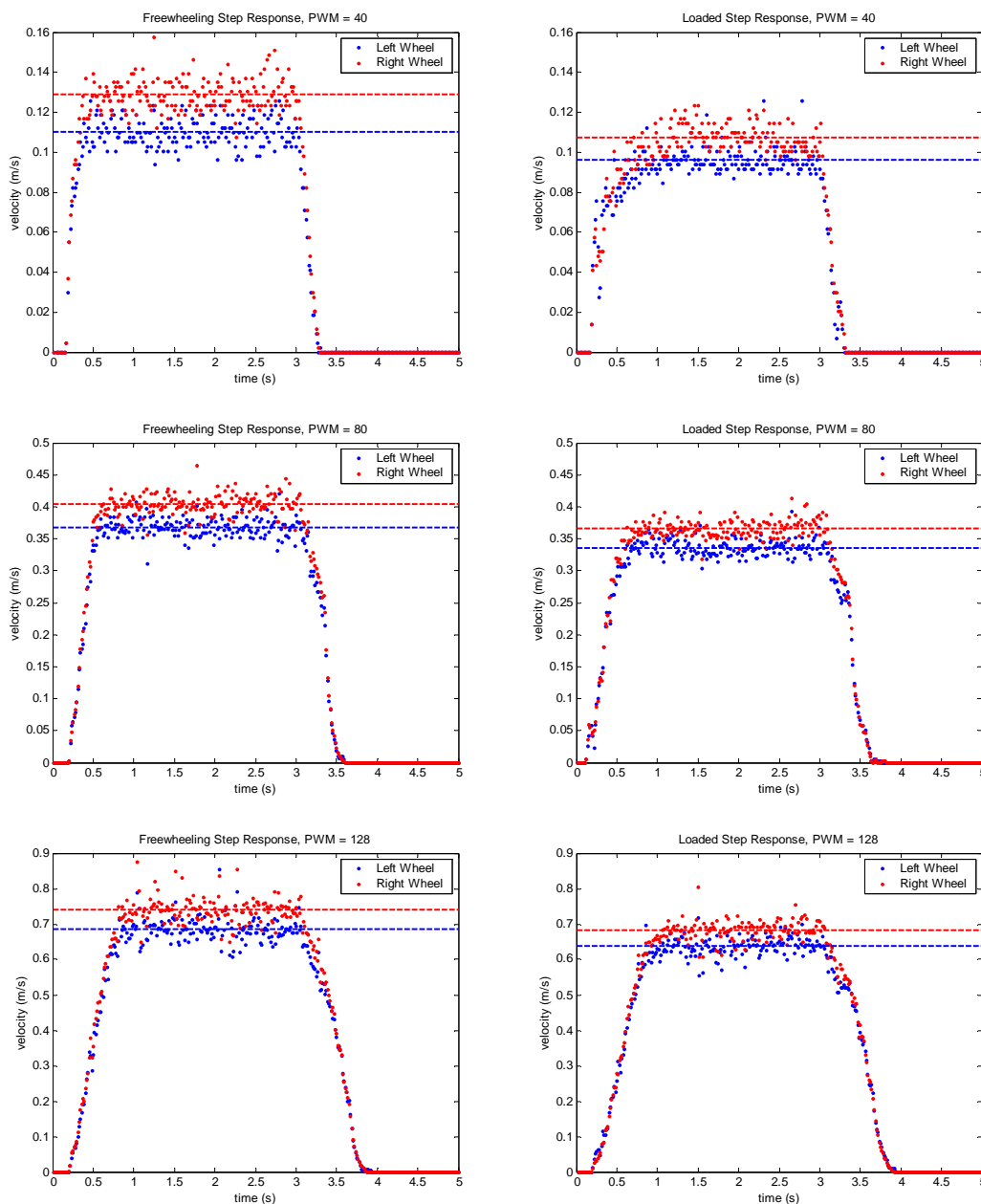
Figure 5.9: Freewheeling and Loaded Step Responses

The average steady-state velocity is measured for each step response, and the resulting data is plotted in Figure 5.10. Both the freewheeling and loaded relationships are linear except where they cross the PWM axis.
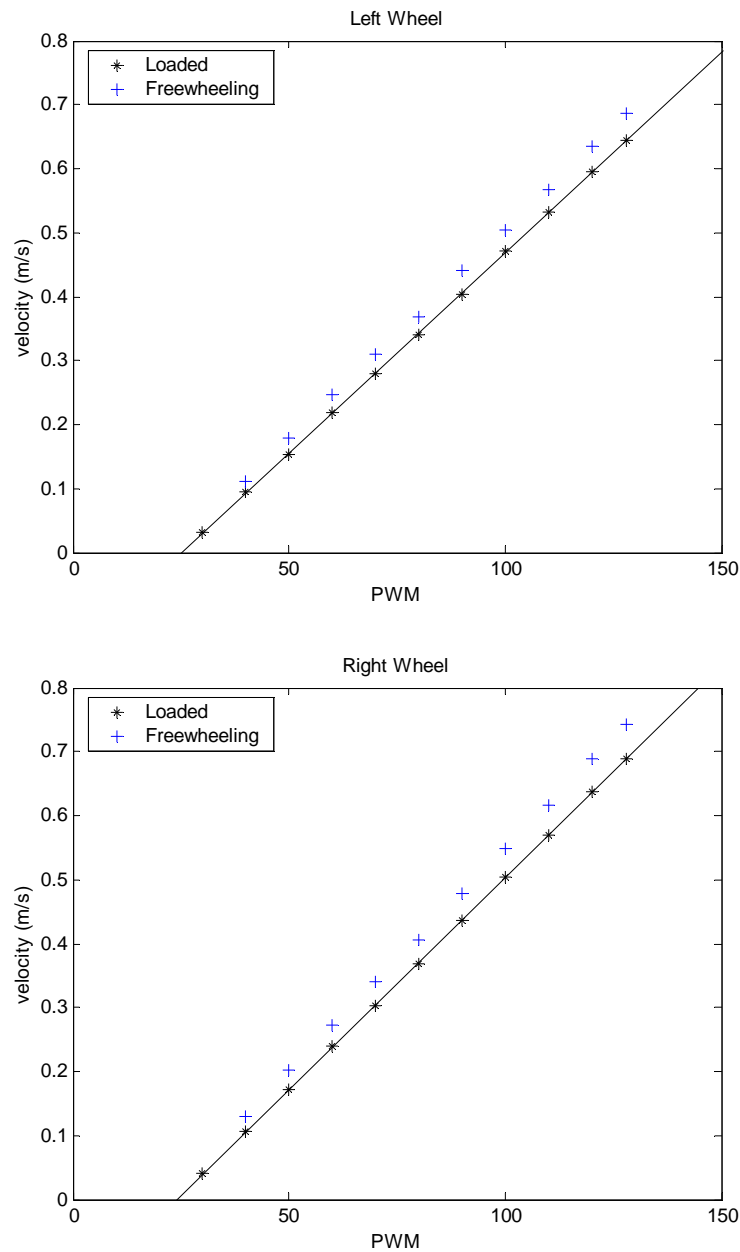
Figure 5.10: Loaded and Freewheeling Velocity-PWM Plots

The function **get_motor_power** (Appendix B.22) models the relationship as a pair of straight lines with equal gradients but different PWM-intercepts for each wheel direction (Equations 5.35 and 5.36).

$$P_l = 159.6 \times v_l \pm 25.2$$ **Equation 5.35**

$$P_r = 151.0 \times v_r \pm 24.0$$ **Equation 5.36**

P_l: Left wheel's PWM value

P_r: Right wheel's PWM value

5.6.2 Writing PWM Value to Microcontroller

PWM values produced by **get_motor_power** are signed floating-point numbers between –255 and 255. These are converted into header and data bytes for the microcontroller communication protocol using **set_motor_power** (Appendix B.23). The header byte is set to a value between 2 and 5, selecting the appropriate motor and direction according to the protocol detailed in Section 3.4.1. The data byte is derived from the raw PWM value using Equation 5.37, with the sign of the ± term depending on direction. A direction flag (Section 5.1.2) is utilised for this purpose rather than the sign of the raw value, because it provides direction information even if the PWM value is zero. If the brake flag is set to 1, the header and data bytes are zeroed, indicating that an emergency stop is necessary.

$$B = \frac{|P|}{2} \pm 128$$ **Equation 5.37**

B: Data byte

The header and data bytes are written to the microcontroller using the **Set Motor Power** VI (Section 3.4.3). **Set Motor Power** returns four error counts indicating the number of times each header or data byte was redelivered.

5.7 Simulation

A number of factors necessitated the development of a software simulation of MARVIN's behaviour, including:

- Delays with the motor driver hardware.
- The simultaneous testing requirements of multiple developers.
- The need for an ideal, controllable environment in which to test algorithms.

This simulation consists of a series of MATLAB functions that model the responses of MARVIN's sensors and actuators, replacing the hardware interface functions. The main function **marvin_control** (Section 3.5.1, Appendix B.2) is structured so that a single flag switches between simulation mode and real mode, and the program selects between the simulation functions and hardware interface functions accordingly.

The odometers are simulated using **sim_en_count** (Appendix B.24), which is a replacement for **acq_en_count** (Section 4.1.1, Appendix B.3). The encoder pulse count over a given control cycle is calculated using Equation 5.38. For testing purposes, cumulative error can be simulated by multiplying a count by a known factor.

$$c_s = ECv_sT \hspace{4cm} \textbf{Equation 5.38}$$

c_s: Simulated encoder count (pulses/m)

E: Cumulative error factor

C: Conversion factor (26996 pulses/m)

v_s: Simulated wheel velocity (m/s)

T: Control cycle period (s)

The **sim_ir_voltage** function (Appendix B.25) replaces **acq_ir_voltage** (Section 4.1.2, Appendix B.4), simulating the infrared rangefinders' voltage outputs. Since the control system lacks an internal map, simulated corridor walls are placed at positions

given by the navigation system or the user. Simulated rangefinder distance calculations are the same as those performed by the rangefinder localisation algorithm to predict distances (Section 4.2.2.3, Equations 4.12-4.14). They are then converted into voltages using the same lookup table as **rep_ir_distance** (Section 4.2.2.2, Appendix B.8). Equation 4.11 is rearranged to obtain voltage from distance (Equation 5.39).

$$V_{IR} = V_{L1} + \frac{(d_{IR} - d_{L1})(V_{L2} - V_{L1})}{d_{L2} - d_{L1}}$$

Equation 5.39

The tactile sensors and beacons are not simulated by the control system. MARVIN's tactile sensors are only useful as an emergency collision warning system, which is not necessary in simulation. The control system does not utilise the beacons, and they cannot be simulated accurately without a map, so the navigation system is better equipped to simulate them.

The function **sim_motor_power** (Appendix B.26) replaces **set_motor_power** (Section 5.6.2, Appendix B.23). Accurate motor driver characteristics were not known at the time this function was developed, so the simulation is not used to test the velocity control algorithm. Instead, it provides an ideal response, setting the velocity of each wheel to the target velocity. The only real-world properties that are simulated are velocity thresholds, representing the slowest velocities that the wheels will tolerate before they stop.

Plots of MARVIN's simulated motion are presented in Chapter 6, where they are compared with data obtained from real world tests.

6 Results

To measure the control system's performance, a number of tests are performed in simulated and real environments. The enclosed CD (detailed in Appendix C) contains captured data, figures and video footage obtained from these tests.

6.1 Open Environment Test Results

The first set of tests is performed inside a 6 × 4 m section of laboratory that is treated as an open environment because the various desks and other obstructions along the walls, as well as the large wall separation, mean that the rangefinders cannot be used for localisation purposes. Various motion instructions are delivered to the control system to execute at 0.2 m/s, 0.4 m/s and 0.6 m/s.

The maximum velocity limit for these tests, 0.6 m/s, approximates the speed that the left wheel (the slowest wheel for any given PWM value) travels at when the PWM value is set to half duty cycle (the present upper limit – refer to Section 5.6 for details).

The control system utilises only the odometers for localisation during these tests, so it is susceptible to a number of errors, including:

- **Initial Misalignments** – The odometers can only measure changes in position and orientation, so the control system cannot detect or correct initial position or heading errors. While position errors contribute little to the final position error, even small initial heading errors can significantly alter the final position. An origin point is marked on the floor as a reference so that MARVIN can be aligned consistently, minimising the heading error. Using the floor markings, the errors due to position misalignments can be reduced to approximately 1 cm in either direction. Initial heading errors are limited to approximately 1.5°. For a 4 m linear trajectory this yields an offset error of 10 cm.

- **Odometry Errors** – Odometry errors can be both random and systematic. An attempt has been made to reduce the systematic errors through odometer calibrations. Random errors are unavoidable in tests such as these that rely on dead reckoning for localisation. Undetected heading errors caused by wheel slippage or missed counts on a single wheel are the most significant source of position error, especially if they occur near the start of the trajectory.

6.1.1 Linear Forward Trajectory

Figure 6.1 gives the result of a simulated 4 m straight-line instruction executed at 0.4 m/s. The equivalent instruction implemented in the real world is shown in Figure 6.2. The simulation can be considered an ideal response that the real-world results should aspire to match. A comparison of Figure 6.1 and Figure 6.2 shows that the real-world results do closely match the behaviour of the simulation after taking the signal noise and motor response characteristics into account.
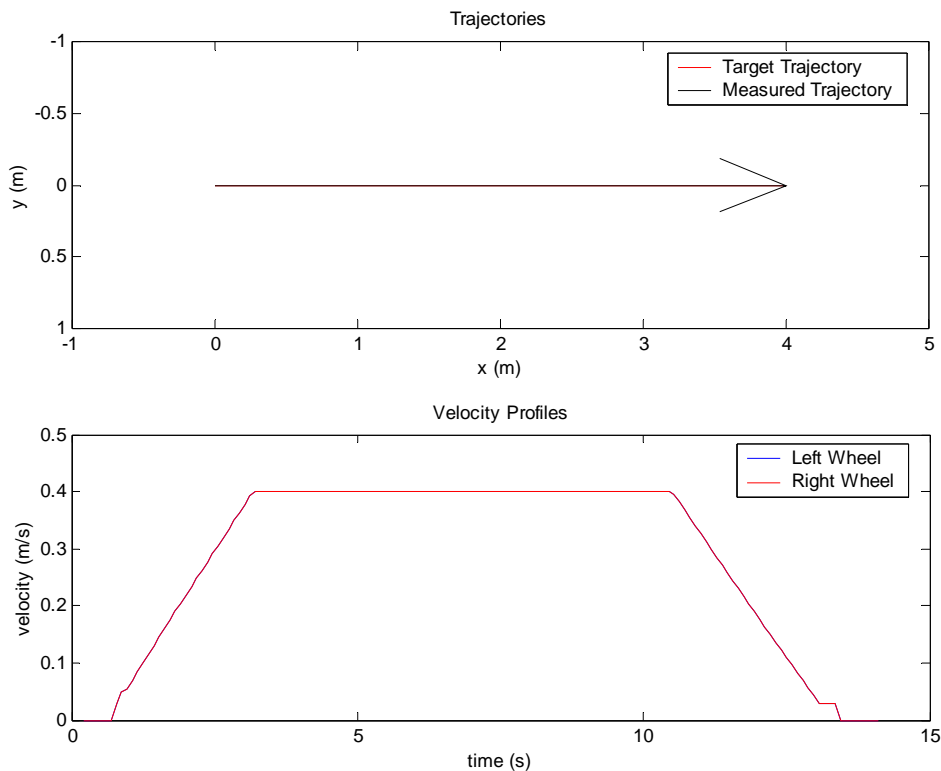
Figure 6.1: Simulation, Distance = 4 m/s, Velocity Limit = 0.4 m/s
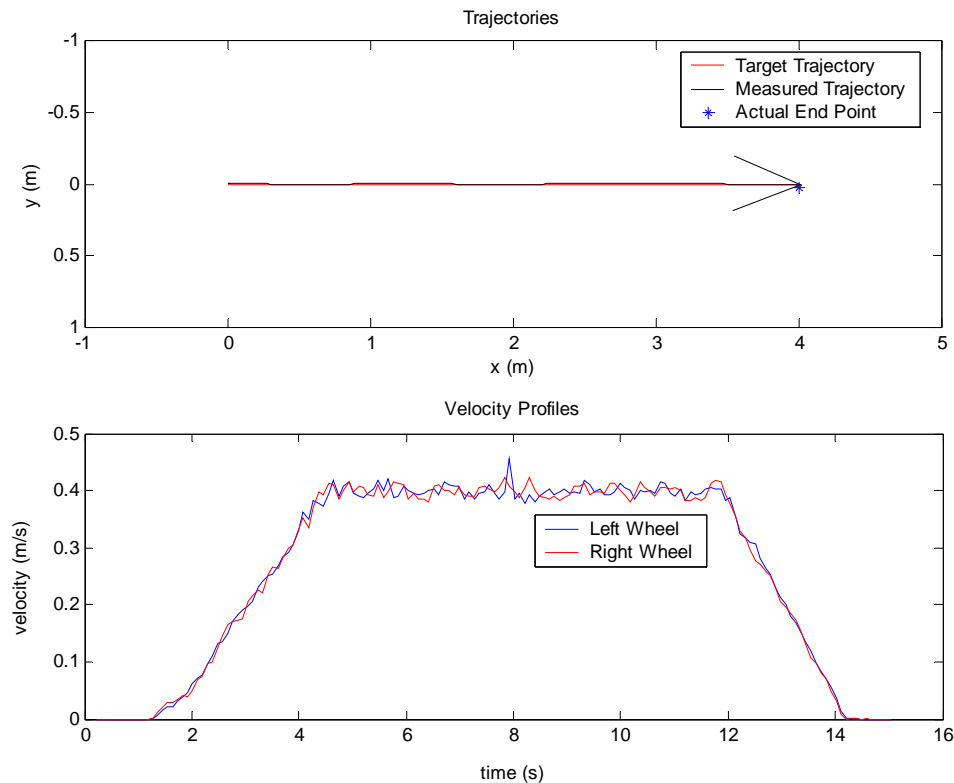
Figure 6.2: Real World, Distance = 4 m/s, Velocity Limit = 0.4 m/s

The target trajectory is plotted for each of these results along with control software's internal representation of the path travelled (the measured trajectory). For the real-world test, the actual trajectory that MARVIN follows is not shown, because it is impractical to externally measure MARVIN's position while it is in motion. Instead, only MARVIN's real-world destination position is plotted. Since this is within 3 cm of the internal measurement of MARVIN's final position, it can be seen that the real-world motion does closely match the internal representation of the trajectory.

The lower plot in each figure contains the velocity profiles measured for each wheel. The only "imperfections" in the simulation velocity profiles are caused by the lower velocity limits in the acceleration and deceleration stages. These limits are in place to prevent the control system from driving the wheels at such low velocities that oscillations occur. They are not visible on the real-world velocity profiles because the motor responses smooth out sharp edges such as these. Much of the noise observed on the real-world velocity profiles (including the small spike at the centre of the left

wheel's profile) is likely due to the limited resolution of the PC's timer (Section 4.2.1.2) than rather than actual variations in wheel velocity.

6.1.2 Linear Reverse Trajectory

A real-world instruction executed at 0.4 m/s in the reverse direction is plotted in Figure 6.3. The result is very similar to the forward motion, with MARVIN maintaining a straight trajectory and arriving within 5 cm of the internal measurement of position. In this test, a slight velocity mismatch during deceleration results in a -1° heading error at the end of the trajectory, but since the system detects the error it can be corrected in subsequent instructions.
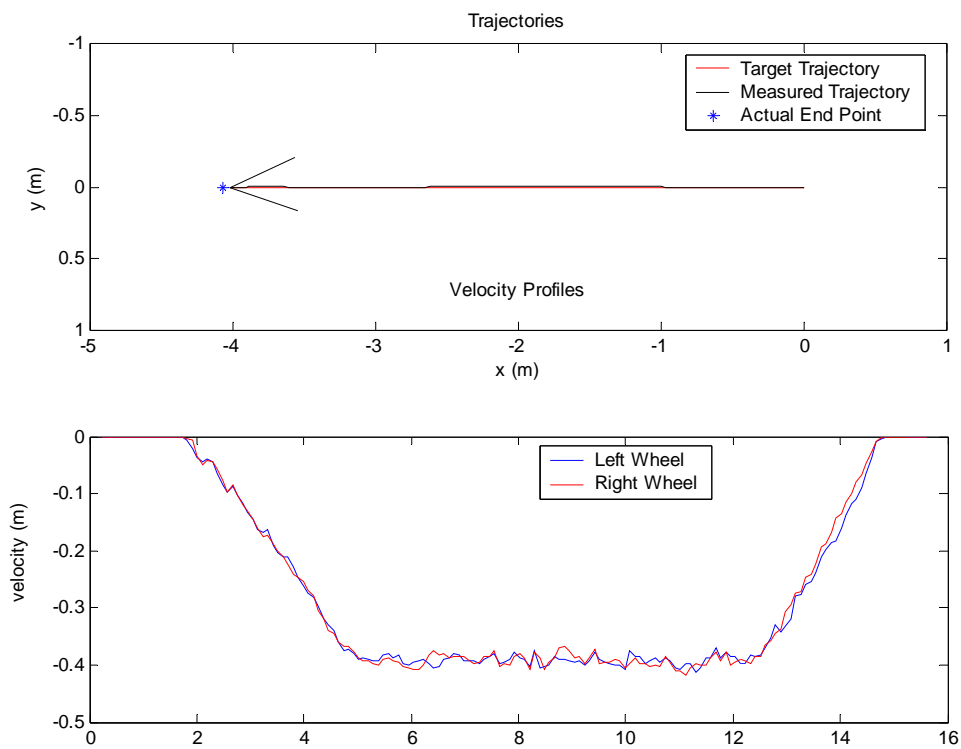
Figure 6.3: Real World, Distance = -4 m, Velocity Limit = 0.4 m/s

6.1.3 Moving Turn

Figure 6.4 shows an 18° moving left turn instruction executed with a 0.6 m/s velocity limit in the real world. The left wheel is maintained at a slightly lower velocity than the right wheel in order to produce a controlled drift to the left. A small disturbance is apparent at the beginning of the left wheel's velocity profile as the control system ramps up the applied velocity to overcome static friction, then slows the wheel to compensate for the initial "velocity surge".
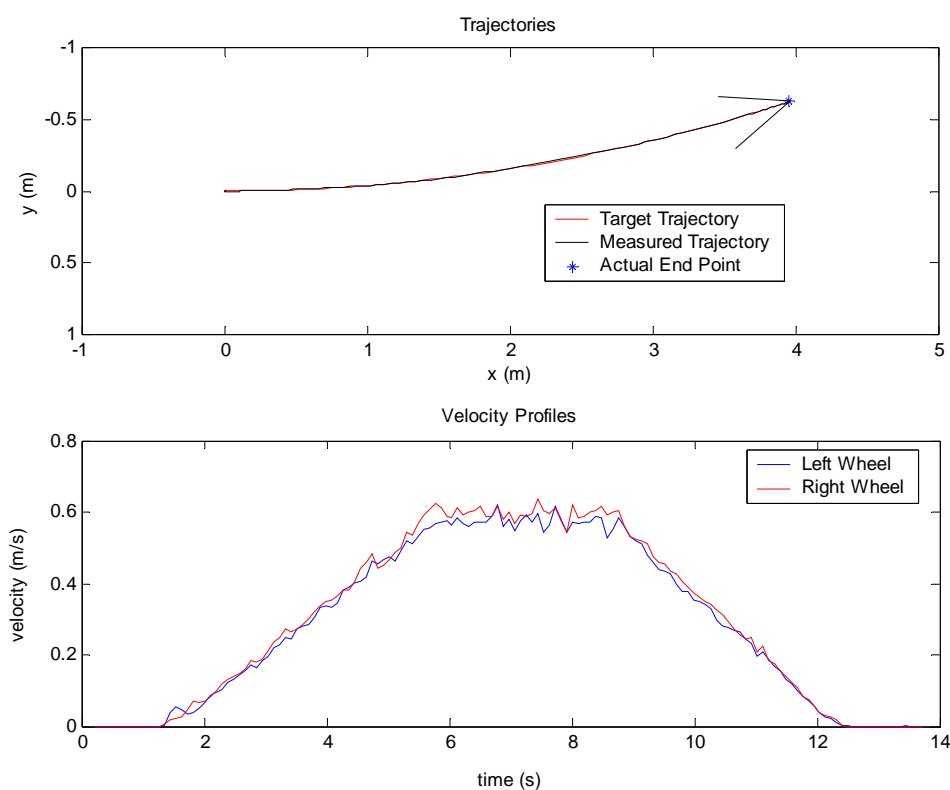
Figure 6.4: Real World, Turning Angle = -18°, Velocity Limit = 0.6 m/s

6.1.4 Linear Trajectory with Offset Angle

A linear distance instruction with a 7.2^o initial offset angle executed at 0.6 m/s is given in Figure 6.5. Of the instructions shown, this is the only instruction that results in significant deviation from the target trajectory. This is to be expected, since the offset angle is implemented as a means to dynamically correct an initial heading error.

MARVIN begins the trajectory aligned horizontally, so the right wheel velocity is reduced in order to steer MARVIN to the right. The left wheel velocity then reduces to line MARVIN up with the target trajectory.
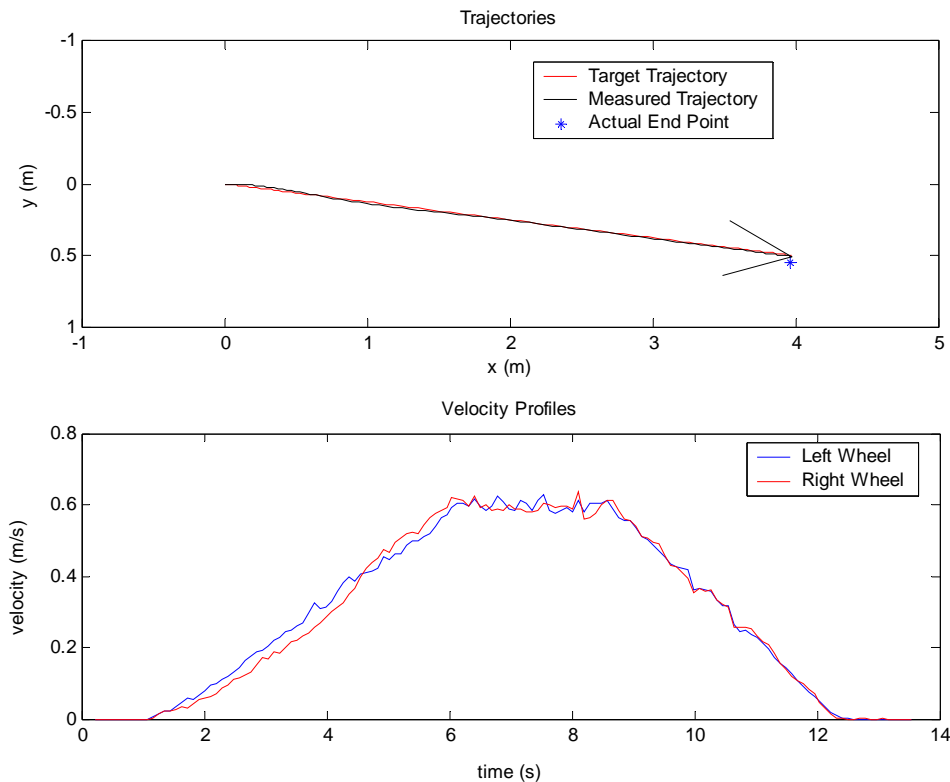
Figure 6.5: Real World, Offset Angle = 7.2°, Velocity Limit = 0.6 m/s

6.1.5 Position Errors

A thorough analysis of the system's accuracy requires that a range of instructions be executed multiple times at each speed. The resulting position errors for each test are given in Figure 6.6. These plots generally show the final positions clustered around the target position. The errors are spread more widely across the y-axis (error range = $-14 \rightarrow 13$ cm, average error magnitude = 5 cm) than the x-axis (error range = $-9 \rightarrow 8$ cm, average error magnitude = 2 cm) because undetected heading errors due to initial misalignment and odometry errors affect the MARVIN's offset more than the overall distance it travels. There is no obvious correlation between velocity and position

error, which is further evidence that changes in velocity do not affect wheel slippage over the measured range.
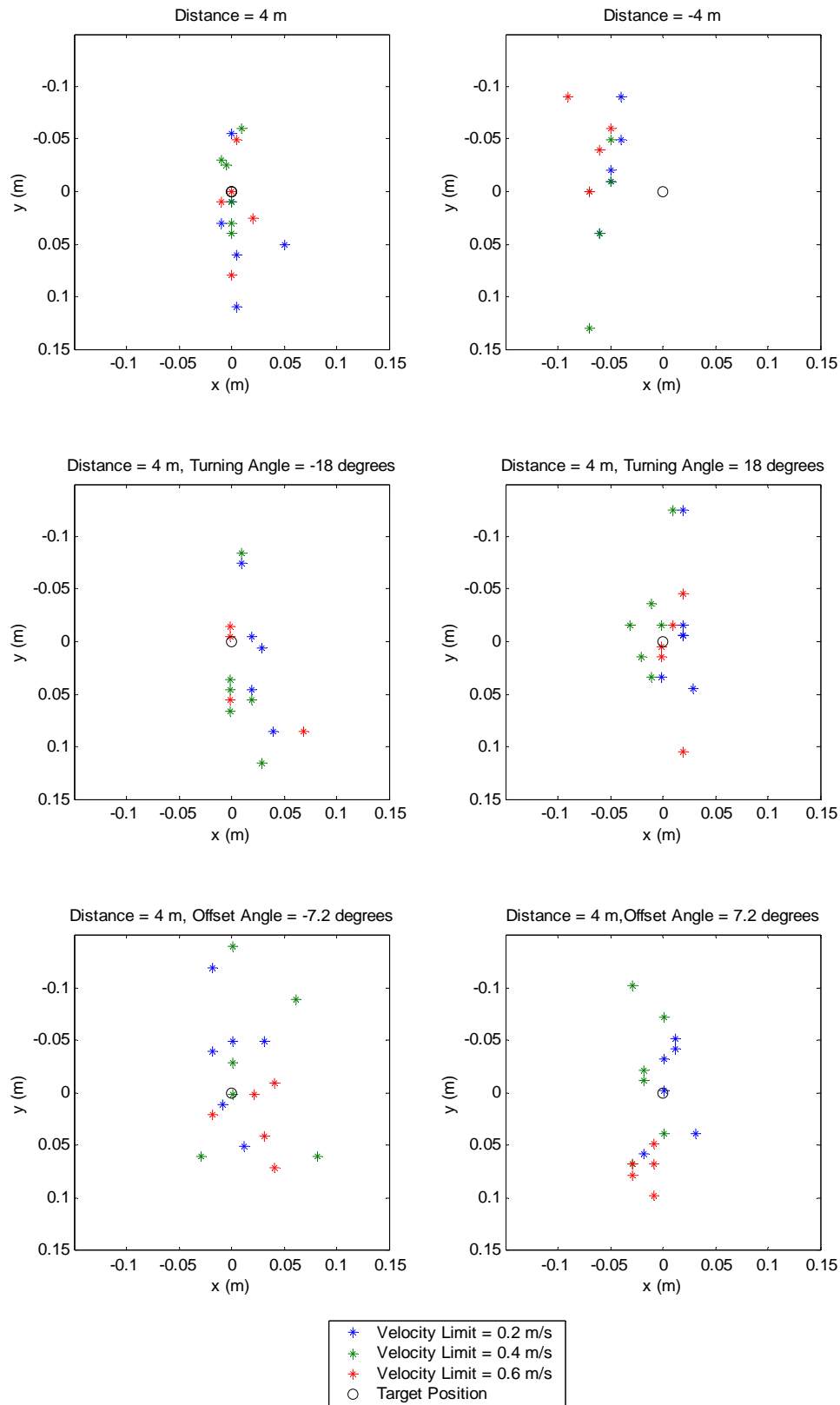
Figure 6.6: Position Errors in an Open Environment

6.2 Corridor Environment Test Results

The coordinate system that the navigation system uses for MARVIN's primary intended operating environment, the corridors on the first floor of C Block at the University of Waikato, is shown in Figure 6.7. Although the navigation system is not utilised in the tests detailed in this section, these results are presented in the same coordinate system. The largest straight section of corridor is selected for these tests so that the expected wall positions can be set to constant values. In an environment as restrictive as these corridors, the only motion instructions that can be executed safely are linear trajectories and stationary turns.
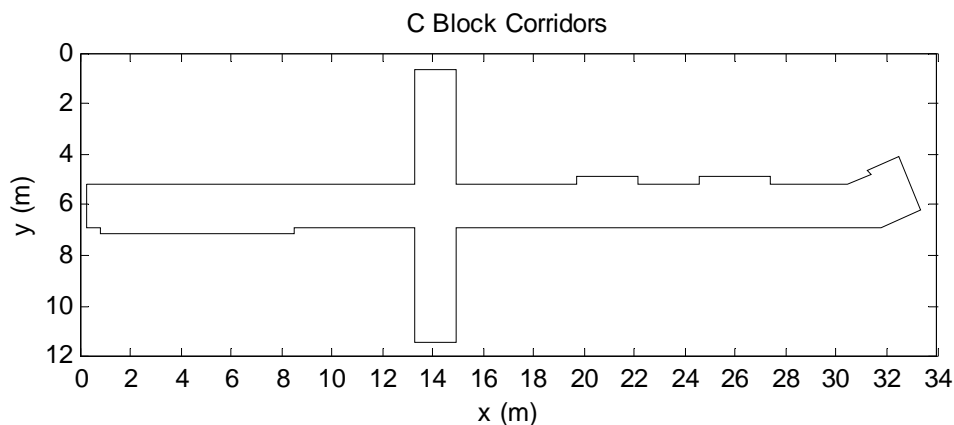
Figure 6.7: Corridor Coordinate System

6.2.1 Linear Trajectories

A simulated 6 m trajectory executed at 0.4 m/s is given in Figure 6.8. Figure 6.9 shows the same instruction executed in the real world with the rangefinder weights zeroed so that they do not contribute to the localisation algorithm. Figure 6.10 is a plot of the instruction executed on the completed system that localises MARVIN using fused odometer and rangefinder data. These plots are similar to those obtained in Section 6.1, but they also include object positions detected by each rangefinder. Like the measured trajectory, the object positions indicate the software's internal measurement of position rather than an actual location in the real world.
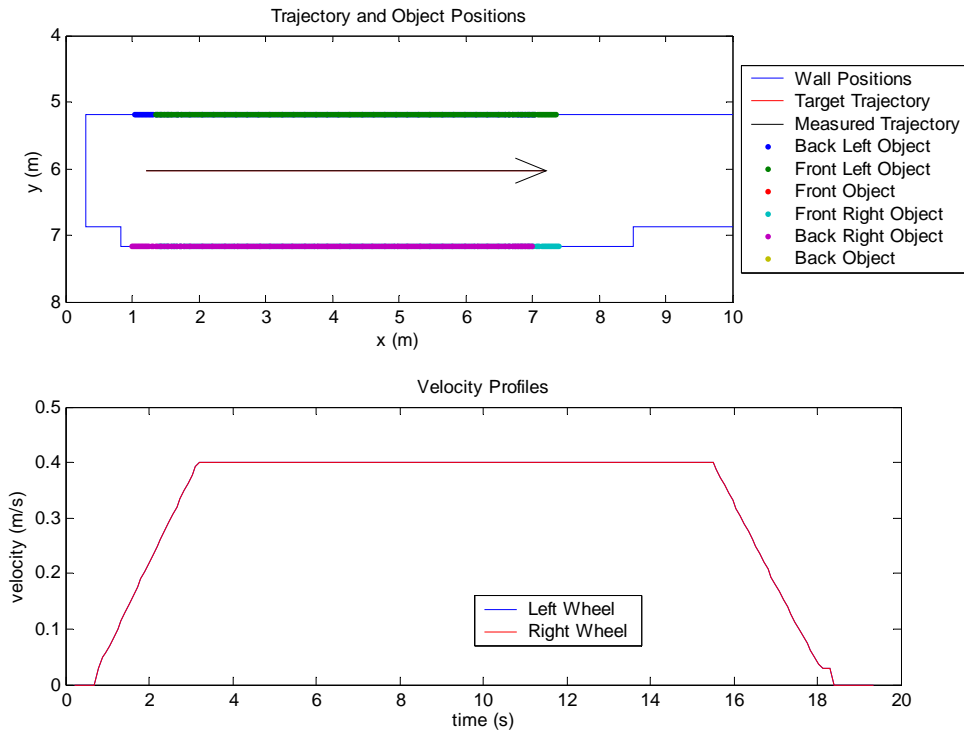
Figure 6.8: Simulation, Velocity Limit = 0.4 m/s
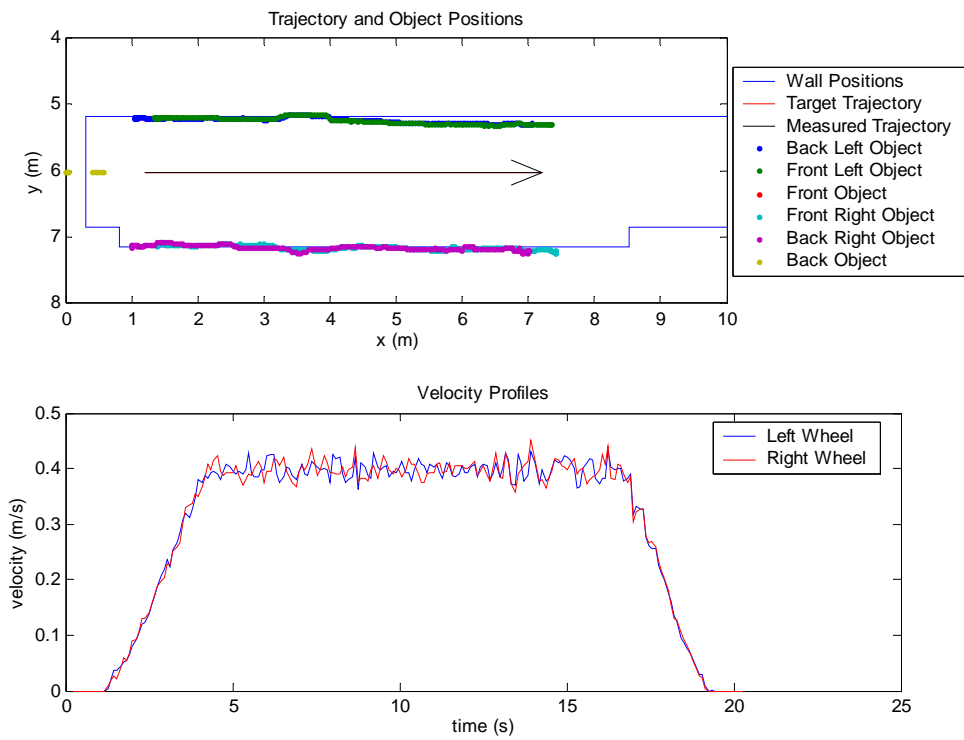
Figure 6.9: Real World, Odometers Only, Velocity Limit = 0.4 m/s
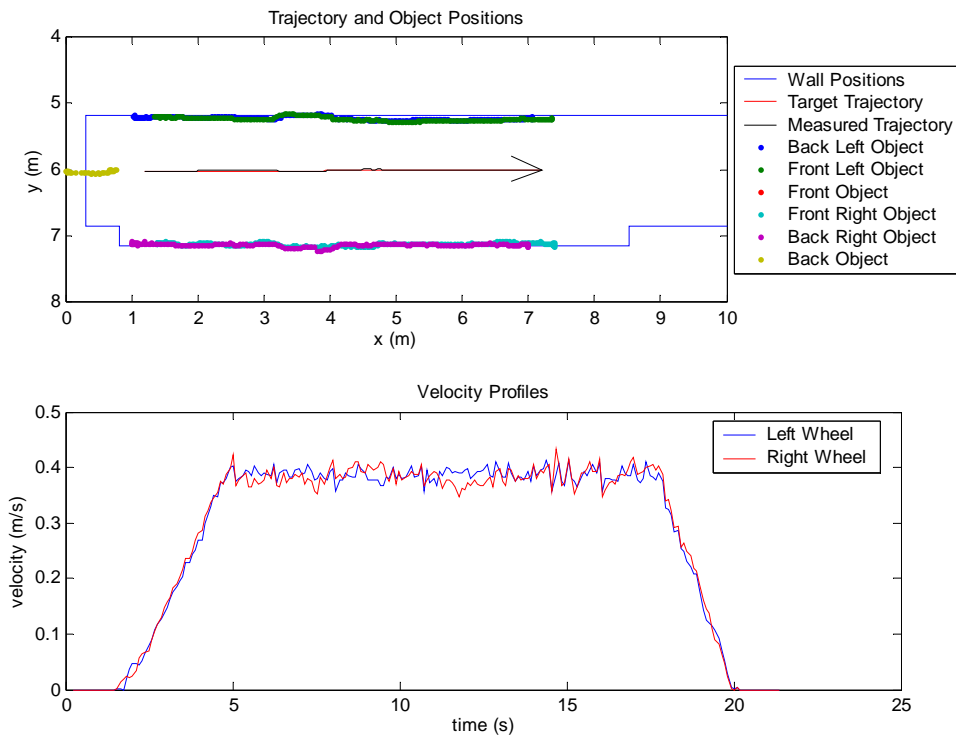
Figure 6.10: Real World, Odometers and Rangefinders, Velocity Limit = 0.4 m/s

The simulation result shown in Figure 6.8 assumes that the wall positions remain constant throughout the entire test, which is not the case in the real world. The rangefinder objects shown in Figure 6.10 give a good indication of the true shape of the corridor walls. The depressions on either side of MARVIN near the mid-point of the trajectory are doors. The raised surface on the left side of the second half of the trajectory is a wall-mounted notice board.

The measured wall positions begin to drift away from their expected positions in Figure 6.9 because MARVIN has drifted to the left due to odometry errors and/or initial misalignment. This no longer occurs in Figure 6.10 because the rangefinder localisation algorithm is continuously correcting any offset and heading errors encountered by the odometer localisation algorithm.

Overall, ten instructions are executed at each speed. Half only utilise the odometers for localisation, while the other half also incorporate the rangefinder data. The resulting position errors, plotted in Figure 6.11, show that the rangefinders reduce the

system's offset (y-axis) error range from –11→18 cm to -7→2 cm. The average offset error magnitude reduces from 7 cm to 3.5 cm.
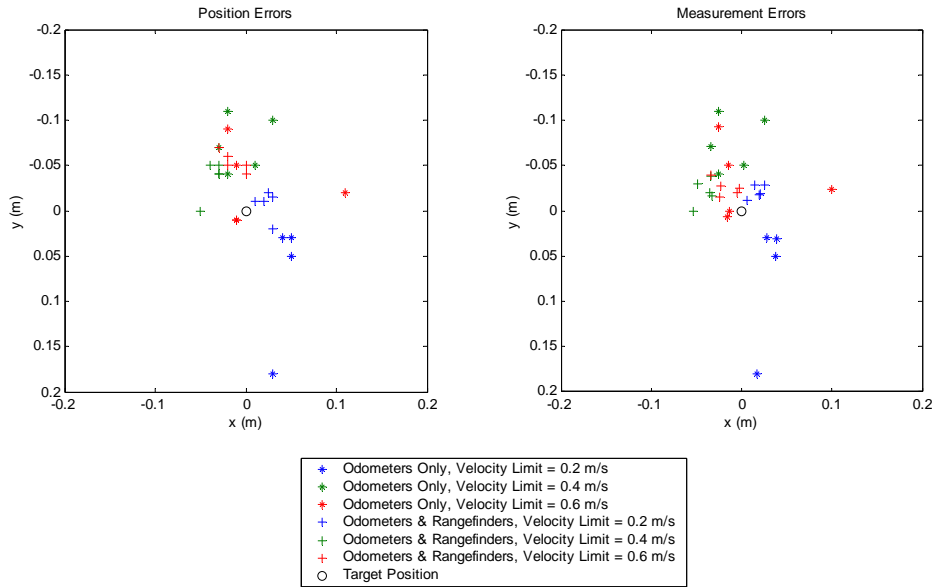
Figure 6.11: Position Errors and Measurement Errors in a Corridor Environment

Instructions executed at slower velocities appear to benefit more from the rangefinder data than those executed at faster speeds. This could be due to the delay associated with the filtering on the rangefinder voltage signals. However, the most likely cause is that the sensor fusion scheme does not take MARVIN's velocity into account. Although the weights remain the same, the rangefinders' contribution is effectively being reduced at higher speeds because the corrections are being applied at fewer positions along the trajectory. The rangefinder weights could become speed-dependant, but increased weights would result in more rapid variations in measured offset and heading. MARVIN's manoeuvrability decreases as its speed increases, so it may react unfavourably to these changes.

Figure 6.11 also includes measurement errors (the differences between the actual positions and the software's internal measurements) for each test. The odometer-only measurement errors are approximately the same as the position errors (error range = –11→18 cm, average error magnitude = 7 cm) – this means that MARVIN's position error is not detected by the software. However, the measurement errors for the results

that incorporate rangefinder data have an average magnitude of only 2 cm, and they are spread over a very narrow range (-4→0 cm) centred around the –2 cm point on the y-axis. The notice board on the left wall is the most likely cause of this minor systematic error. Since the expected wall positions are set to constant values for these tests, the rangefinder localisation algorithm cannot take into account the slight variation in expected wall positions. If it was utilised for these tests, the navigation system could adjust the expected wall positions using information from its internal map, eliminating this error.

6.2.2 Stationary Turns

The rangefinder localisation algorithm is very effective at correcting offset and heading errors for motion that is parallel to the walls, but its usefulness diminishes for other types of motion, as demonstrated by the sequence of two 90° stationary turns shown in Figure 6.12. In this test, a 470 ms delay due to the filtering on the rangefinder signals distorts the measured wall positions while MARVIN is turning.
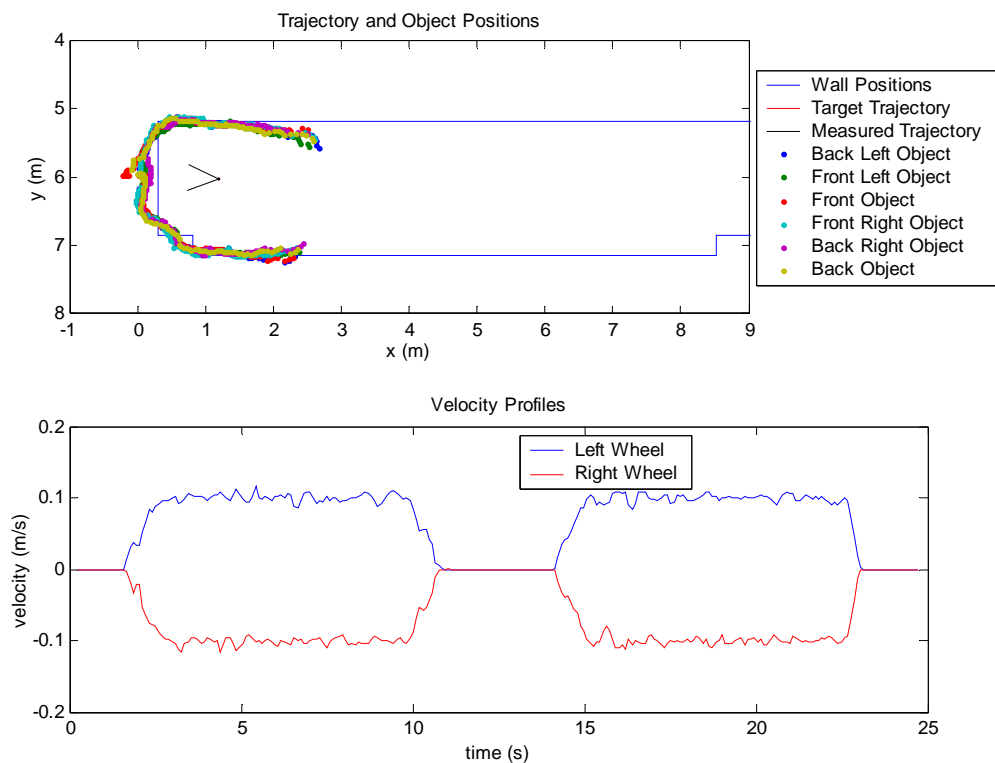
Figure 6.12: 90° Right Turns in Corridor

6.2.3 Extreme Tests

In order to gauge the true effectiveness of the rangefinder localisation algorithm, tests are performed under extreme conditions that the odometers alone would be unable to withstand. Deliberate misinformation about the initial heading is delivered to the software, and the rangefinder localisation algorithm must detect the true heading so that the control system can apply the appropriate course corrections. These tests are performed at low velocities (0.2 m/s) for safety reasons. Figure 6.13 shows the results of a test performed with MARVIN oriented approximately 20° away from the expected horizontal heading. Figure 6.14 is a similar test performed with an initial 60° heading error. In both cases MARVIN corrects the initial heading error within the first 3 m of the trajectory and arrives within 10 cm of the target position. The exact position errors for each test are given in Table 6.1.
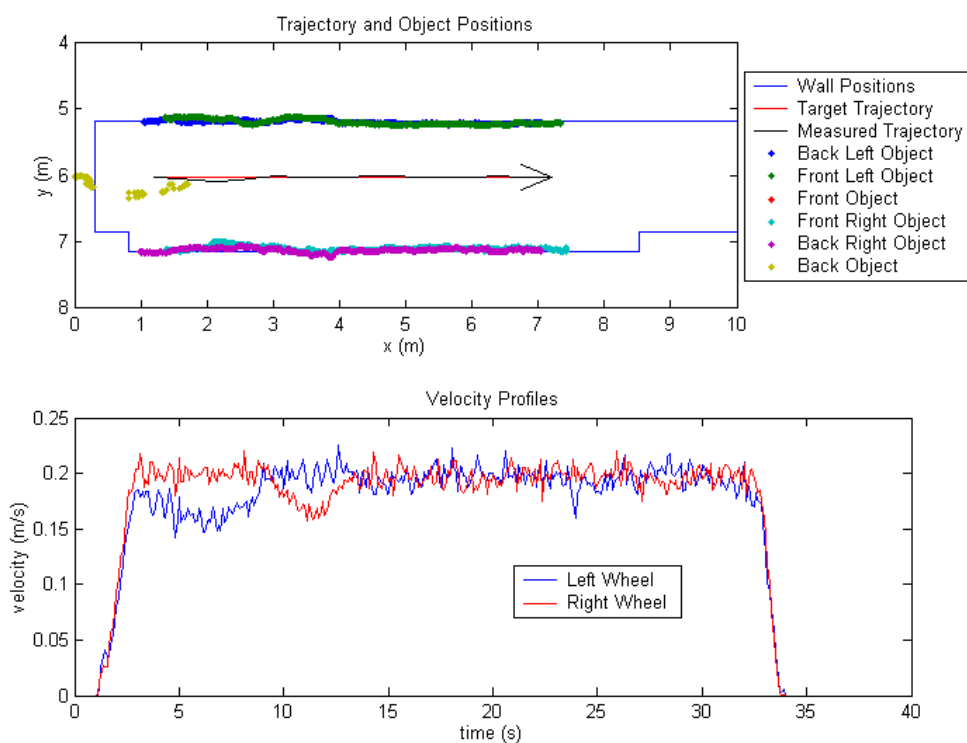
Figure 6.13: Correcting Initial 20° Heading Error
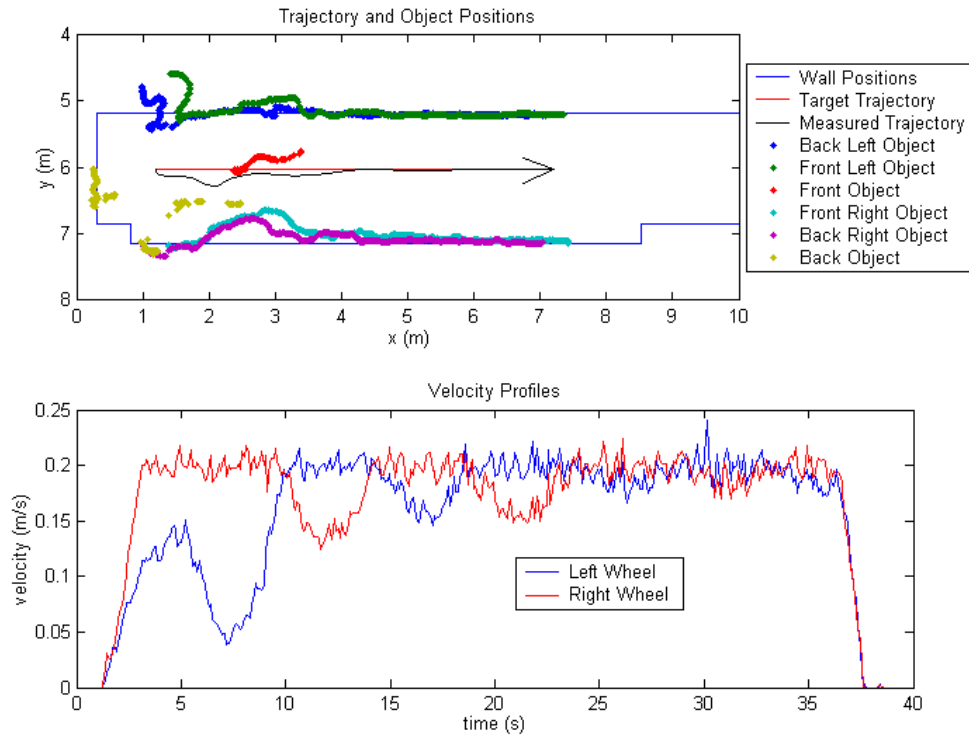
Figure 6.14: Correcting Initial 60° Heading Error

Table 6.1: Position Errors for Extreme Tests

Initial Heading Error	Position Error, x-axis	Position Error, y-axis
20°	-3 cm	0 cm
60°	10 cm	0 cm

6.2.4 Collision Avoidance

The collision avoidance tests involve placing a hapless victim in front of MARVIN as it executes a normal instruction. Figure 6.15 shows that MARVIN stops within 40 cm of the lucky individual when travelling at 0.6 m/s. The same individual has also learned through bitter experience that when all else fails the tactile sensors do indeed function correctly.
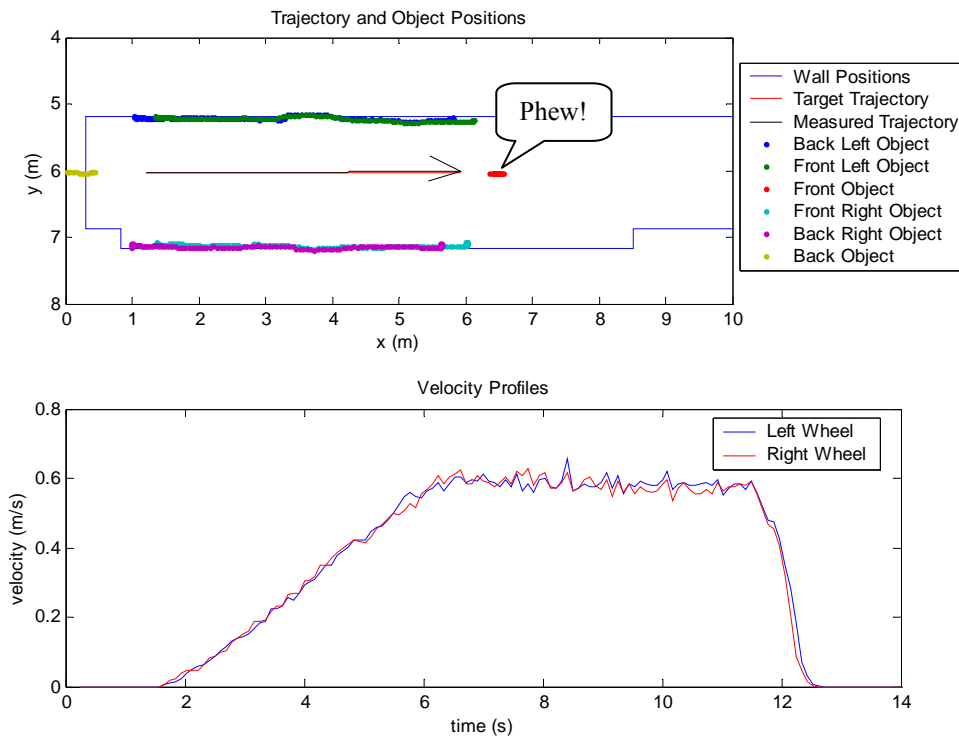
Figure 6.15: Collision Avoidance

6.3 Combined System Test Results

In order to travel long distances down a corridor while utilising the rangefinder localisation algorithm, the wall positions must be updated dynamically. Also, the rangefinder weights must be temporarily zeroed whenever MARVIN passes through regions such as the corridor intersection that may yield misleading measurements. Controlling these inputs manually does not produce reliable or repeatable results, so these tests can only be performed in conjunction with the navigation system.

6.3.1 Single Instruction

The first tests performed on the combined navigation system and control system are linear distance trajectories that traverse the entire corridor. The navigation system delivers a single 29.3 m instruction to the control system and adjusts the wall positions and rangefinder weights as it executes. Figure 6.16 shows the simulated trajectory executed at 0.6 m/s. Figure 6.17 shows the equivalent instruction executed in the real world.

The fact that the measured object distances line up with the expected wall positions in the y-axis shows that MARVIN maintains a relatively straight trajectory over the entire distance, which would not be possible using odometers only. False readings are apparent on some of the measured object positions. The "objects" detected by the front and rear rangefinders near the start and the end of the trajectory are most likely due to ambient light or reflections from the windows at each end of the corridor. The disturbance on the left wall near the end of the trajectory is a glass cabinet.
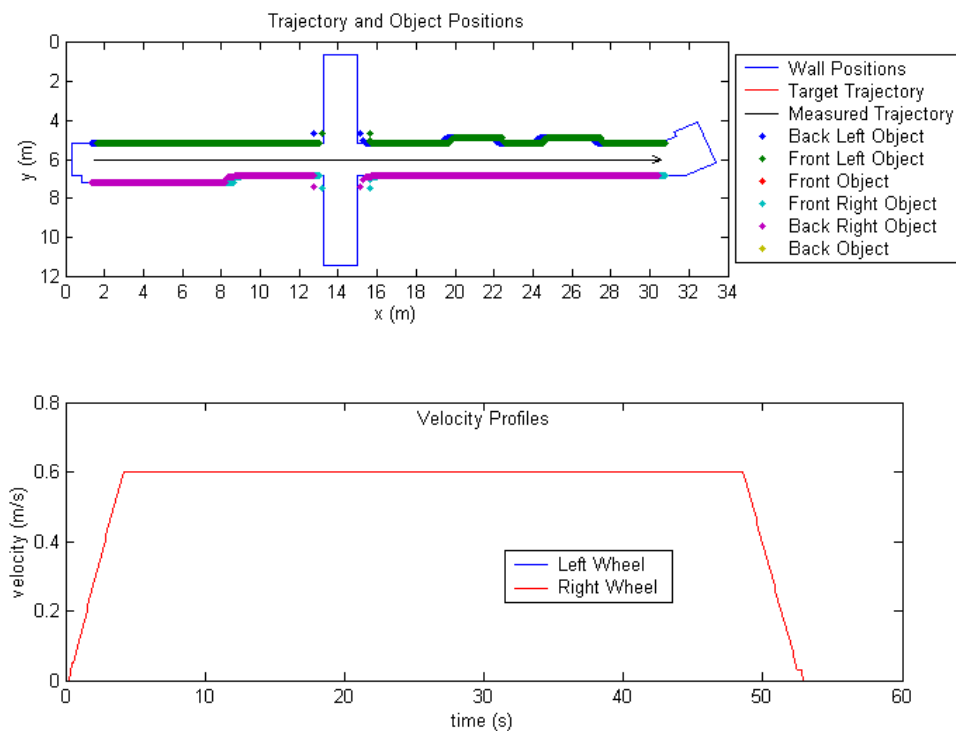
Figure 6.16: Simulation, Single Instruction, Velocity Limit = 0.6 m/s
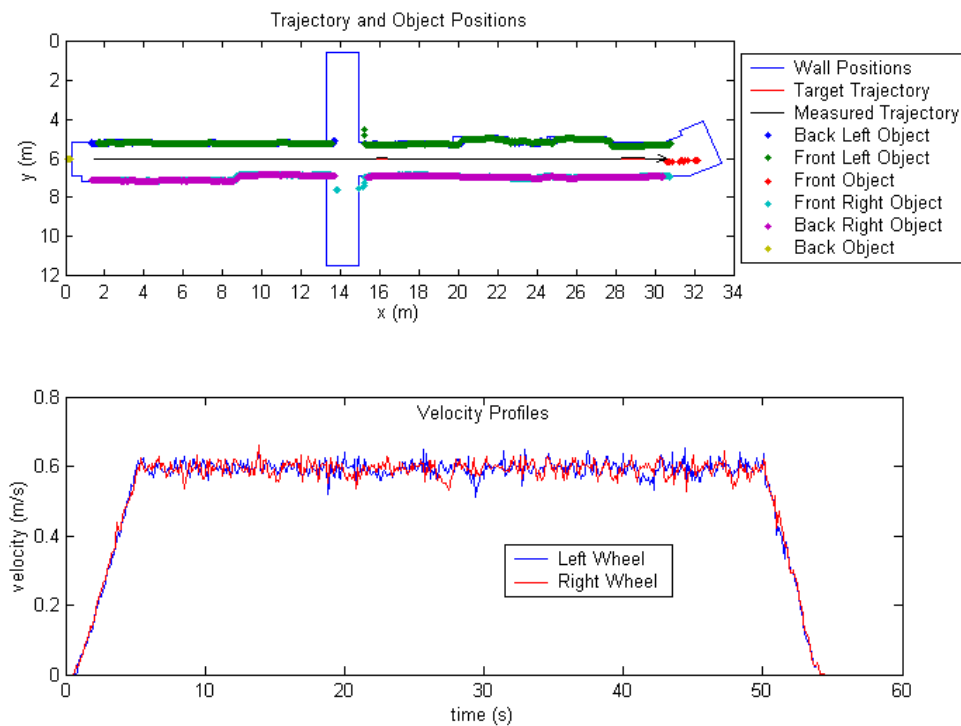
Figure 6.17: Real World, Single Instruction, Velocity Limit = 0.6 m/s

Position and measurement errors for a number of similar tests are plotted in Figure 6.18. Given the large distance travelled, it is expected the distance errors should be larger than the errors measured in previous tests, and the results confirm this (error range = -22→7 cm, average error magnitude = 11 cm).

Offset errors should ordinarily be corrected based on the rangefinder data, but the offset errors for these tests (error range = -15→5 cm, average error magnitude = 8 cm) show a general increase from those obtained in the previous tests. The most likely cause of this discrepancy is that the rangefinder weights on the left side are zeroed over a large section of corridor close to the final position to prevent the control system from reacting to the shifting wall positions and the glass cabinet. Since the left wall is only utilised briefly at the end of the trajectory, the control system does not have time to detect and correct the offset errors. This also explains why the errors are corrected more successfully at 0.2 m/s than at the higher velocities.
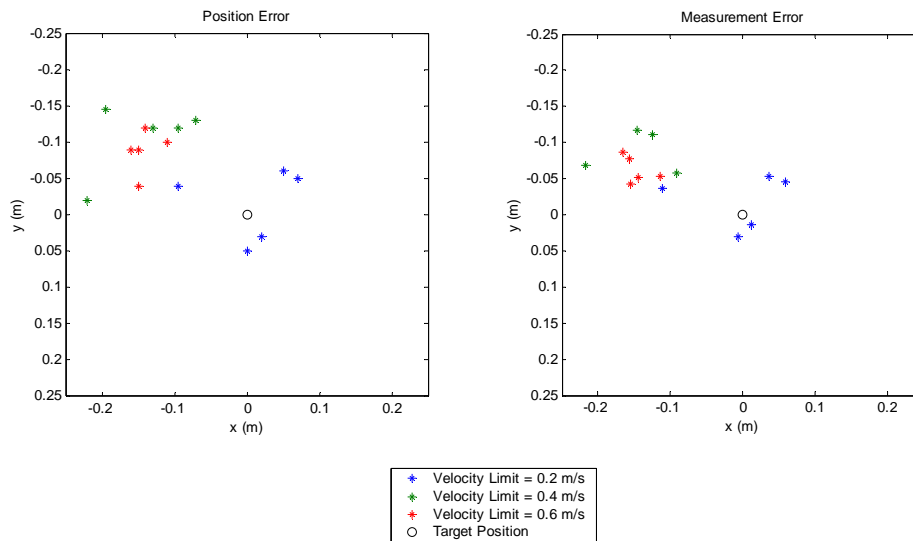
Figure 6.18: Errors for Single Instructions

6.3.2 Sequence of Instructions

The final tests involve a sequence of three instructions delivered by the navigation system – a linear distance, followed by a 90° left turn, then another linear distance. Due to odometry errors MARVIN is unlikely to be exactly in the centre of the corridor intersection following the left turn, nor will it be facing the correct direction, so the control system must acquire the intended position and orientation after it enters the new section of corridor. The resulting simulated trajectory is given in Figure 6.19 and the real-world trajectory is plotted in Figure 6.20.

The primary disturbances observed on the measured object positions occur in the corridor intersection. This is because the lag introduced by the rangefinders' software filter smears the measured corner positions when MARVIN executes a stationary turn. The slight disturbance on the right wall next to the final position is produced by a drinking fountain.
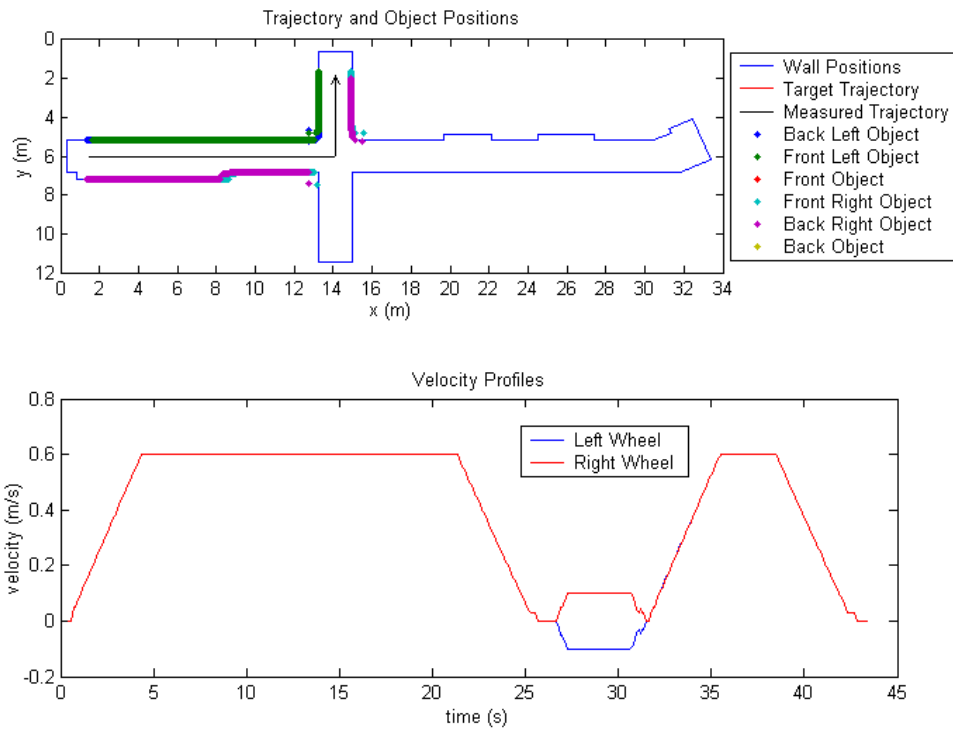
Figure 6.19: Simulation, Sequence of Instructions, Velocity Limit = 0.6 m/s
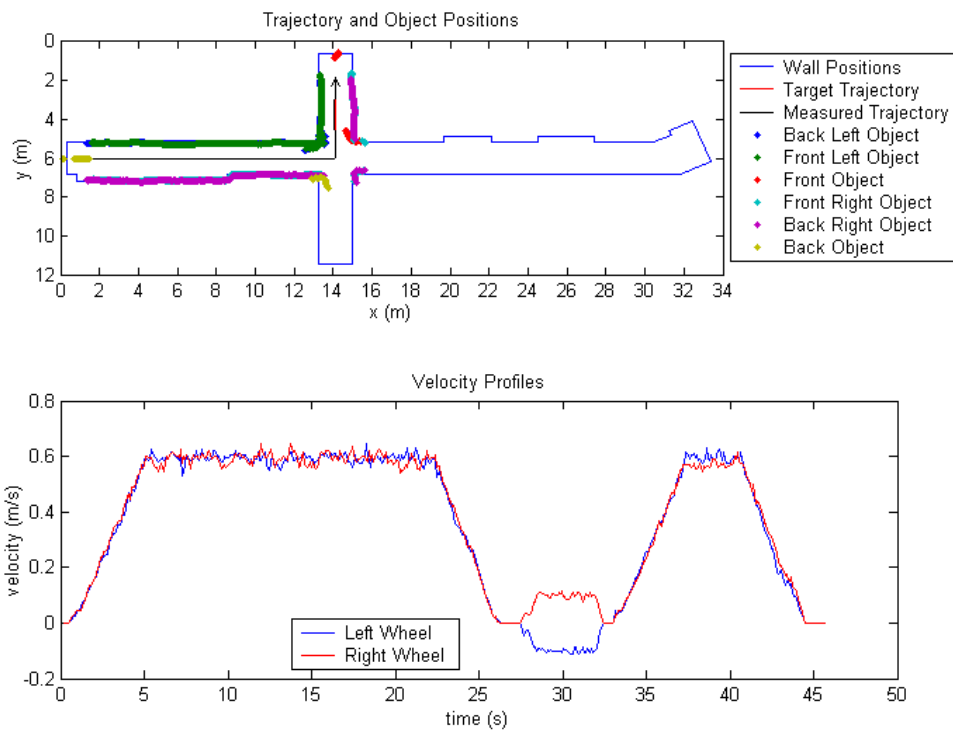
Figure 6.20: Real World, Sequence of Instructions, Velocity Limit = 0.6 m/s

Due to odometry errors, MARVIN enters the second section of corridor slightly to the left of the corridor centre axis. Once the navigation system sets the rangefinder weights to nonzero values, the control system detects the position error and steers MARVIN to the right, as evidenced by the reduced velocity of the right wheel shown on the velocity profile. The measured trajectory actually appears straighter than the observed motion in the real world because the rangefinder localisation algorithm shifts the measured position towards the left at the same time as MARVIN moves to the right to correct the error.

A slight drift to the left is observed as MARVIN travels through the corridor intersection (where it cannot use the rangefinders for localisation) before it executes the left turn instruction, resulting in an initial position error that the control system cannot detect or correct. This causes MARVIN to overshoot the target position by approximately 4 cm.
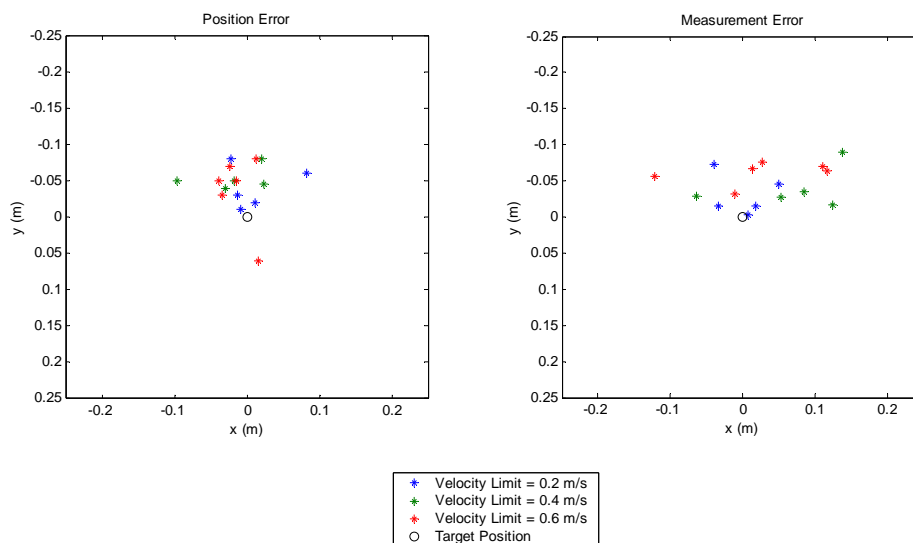
Figure 6.21: Errors for Sequences of Instructions

Figure 6.21 gives the position and measurement errors for a number of similar tests. Unlike the previous results, the position errors shown for these tests do not represent the final positions with respect to a single target position because the navigation system instructs the control system to drive MARVIN to a slightly different target position for each test. The x-axis represents the final offset errors for these tests, since

MARVIN turns into the vertically oriented section of corridor. The measurement errors are spread widely across the x-axis (error range = -12→12 cm, average error magnitude = 6 cm), probably due to the non-uniform wall positions in the target section of corridor, as well as the fountain.

7 Conclusions

7.1 Objectives Achieved

The following thesis objectives have been achieved:

- New HEDS-5500 optical encoder modules, Sharp GP2Y0A02YK infrared rangefinders, custom-built tactile sensors, Kemo B062E beacon receivers, custom-designed H-bridge motor drivers and a Phillips P89C51 microcontroller were installed on MARVIN and interfaced to the DAQ card. The hardware has remained stable throughout months of testing.

- LabVIEW software was developed that utilises the DAQ card's internal counters to measure pulses from the odometers. The software obtains voltage readings from the infrared rangefinder voltages using the DAQ card's built-in ADC. Values indicating the logic states of the tactile sensor switches and beacon receiver relays are received on the digital I/O ports. Software was created to interface with the microcontroller using the communication protocol established by Andrew Payne, performing the appropriate error corrections when necessary.

- A number of alternative inter-application interfaces were investigated: ActiveX Control Containment, ActiveX Automation, MATLAB Script Nodes, Dynamic Data Exchange and File I/O. A robust software interface was established between MATLAB, LabVIEW and Microsoft Word using ActiveX Automation.

- The sensor data measured by LabVIEW is delivered to MATLAB, where the raw signals are conditioned for use by the localisation algorithms. A technique was developed to apply direction information to the raw odometer counts. Several alternative software filters were investigated for the rangefinder signals, and a median average scheme was selected.

- Distance measurements were obtained from the odometer counts, and the corresponding conversion factors were calibrated to reduce the errors. Equations were derived to obtain MARVIN's distance, offset, heading and

wheel velocities from the individual displacements measured by each wheel over time.

- The voltage-distance relationship for each rangefinder was measured, and lookup tables were created that obtain distance measurements from the voltage signals. An algorithm was developed to obtain offset and heading information from the measured wall positions in a corridor environment. Offsets can be obtained from individual rangefinder measurements, whereas headings are derived from the relative wall positions measured by two or more adjacent rangefinders.

- An algorithm was developed that combines the odometer and rangefinder localisation information intelligently. Various sensor fusion techniques were considered, including Bayesian inference, Dempster-Shafer inference, fuzzy logic and neural networks. A dynamic weighted average scheme was selected.

- Algorithms were developed to obtain target motion trajectories and velocity trajectories from arbitrary distance and angle inputs. The target trajectory is used to ensure that MARVIN's position and orientation is correct at all times. The velocity profile provides an intended velocity for each wheel that will result in smooth acceleration and deceleration while driving MARVIN along the target trajectory.

- Various control schemes were researched for MARVIN's motion control system, including PID, fuzzy logic, neural network and neuro-fuzzy. Two separate PID control loops were implemented. The outer loop controls MARVIN's heading, maintaining its motion along the target trajectory based on localisation information obtained by the sensor fusion algorithm. The inner loop controls MARVIN's wheel velocities in an attempt to match the velocity profiles while applying course corrections obtained by the heading control system.

- A collision avoidance system was implemented that groups perceived obstacles into three levels of threat according to their measured proximity, and takes evasive action.

- Extensive tests and calibrations were performed in the real world in order to measure and refine the system's performance.

In addition to these fulfilling all the intended objectives, the project has achieved the following:

- MARVIN's PC hardware has undergone an extensive overhaul, receiving a new ACE-828C 24V power supply, Shuttle xPC, 6025E DAQ card and ZyAIR B-220 wireless LAN module.

- A software simulation was developed that models the behaviour of MARVIN's sensors and actuators. It integrates seamlessly with the control system, allowing developers to switch between simulated and real environments by adjusting a single variable. A MATLAB GUI was created so that MARVIN's motion and perceived environment, both in simulation and in the real world, can be tracked in real time. A data logging system was created to record the relevant data for future analysis.

- The control system was successfully interfaced with the navigation system, resulting in a combined system that can navigate autonomously throughout a corridor and laboratory environment, avoiding any obstacles it encounters along the way.

7.2 Future Work

The completed control system has proven successful, but there are a number of improvements that could be made in the future:

- Additional Sensors
- Motor Driver Improvements
- Simulation Improvements
- Improved Sensor Algorithms

7.2.1 Additional Sensors

The following sensors could be added to MARVIN to improve its localisation and navigation capabilities:

- **Laser Rangefinder** – Shaun Hurd's custom laser rangefinding system [Hurd, 2001] must be interfaced to MARVIN's new PC. The device can be utilised for localisation purposes in the same manner as the infrared rangefinders. However, its capabilities are more easily exploited by the navigation system than the control system. Due to its comparatively long (10 m) measurement range, the laser rangefinder is useful as a means to map the environment dynamically. It will also allow the navigation system to detect impending obstacles in time to plot a course around them without halting the robot.

- **Compass** – A compass will be useful as an absolute heading reference if the interference issues can be overcome. Magnetic fields generated by objects inside the operating environment (or generated by MARVIN itself) will interfere with a standard magnetic compass. A compass module designed specifically for robotic applications that can compensate for magnetic interference, such as the P2UsCMP120 from ActivMedia Robotics [http://www.activrobots.com], may prove an ideal solution. Alternatively, an inertial sensor such as a gyrocompass could be used.

- **Optical Mouse** – A standard optical mouse contains a CCD camera and a DSP that measures changes in position from the shifting patterns on a moving surface. If an optical mouse is positioned close to the floor or focussed through a lens, it could provide an alternative form of dead reckoning localisation that is not susceptible to the same errors as the odometers (e.g. wheel slippage).

7.2.2 Motor Driver Improvements

If future projects require MARVIN to travel at faster velocities than the present software limits allow, extra precautions may become necessary to prevent high-speed collisions. The microcontroller software may also require modifications so that the duty cycle limit can be safely removed without risking damage to the motor driver circuits. Reducing the acceleration limit is one solution, but this would compromise the system's speed of response at all velocities. A more favourable alternative is to implement a speed-dependent acceleration limit. Since the emergency brake signal and the communication timeout bypass the acceleration limit, future developers must ascertain the safest course of action to take in the event of a collision or a PC lockup.

Further investigations are also required to determine whether MARVIN's present motor drivers should be replaced with Craig Jensen's generic motor drivers.

7.2.3 Simulation Improvements

MARVIN's hardware is now functional, so the simulation is no longer essential, but it is still very useful for the initial testing stage whenever a significant modification or addition is implemented on MARVIN's software. The simulation will allow future developers to debug the software in a safe environment where errors do not result in physical damage. However, due to the simplicity of the present simulation, many errors are not detected until the code is executed on real hardware. It might therefore be useful to expand the simulation to incorporate additional real-world properties such as motor response characteristics. As new sensors and actuators are installed on MARVIN, simulation functions should also be developed to model their behaviour.

7.2.4 Improved Sensor Algorithms

New sensors that are installed on MARVIN can be added to the sensor fusion algorithm by assigning the appropriate weights. However, as the number of sensors increases, the weighted average scheme will become less effective. At some point a more complex algorithm such as a Bayesian or Dempster-Shafer scheme or a neural network may become necessary.

One potential improvement that could be applied to the rangefinder localisation algorithm is to extrapolate MARVIN's position and orientation from measured wall positions over time, instead of (or as well as) those measured by multiple sensors at a single point in time. Figure 7.1 superimposes the measured wall positions for two separate trajectories, and shows that the data points are generally grouped into straight lines. A line representing a detected wall can be fitted to these data points using the MATLAB function **polyfit**, and compared with the line representing the expected wall position. Any transformation that is applied to the measured line in order to match it to the expected wall position could then be applied to MARVIN's position and orientation.
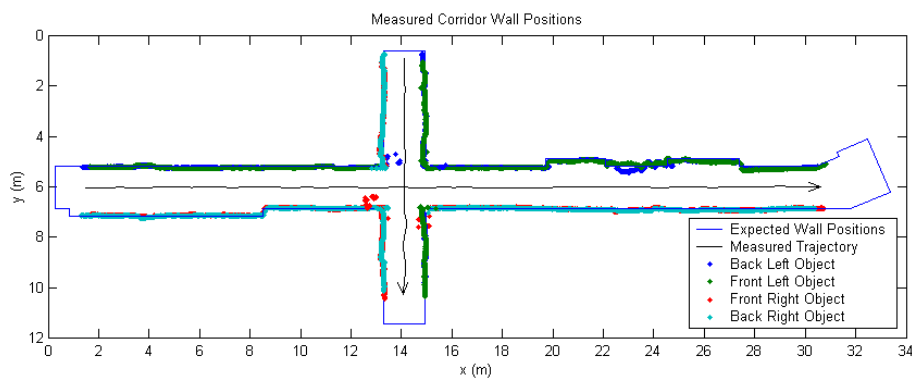
Figure 7.1: Wall Positions Measured over Time

The main advantage of this technique is that it reduces errors due to sensor noise and wall disturbances. It also allows the localisation algorithm to measure MARVIN's heading even if only a single rangefinder is facing the wall. However, a significant drawback is that it will slow the algorithm's response to changes in offset or

orientation. The rangefinders' software filters already result in distortions when a stationary turn is executed in the corridor, and any further delays would exaggerate the problem further. These advantages and disadvantages should be carefully considered before such a revision is implemented.

7.3 Summary

The result of this project is a hardware and software platform that smoothly executes a range of motion instructions in a corridor or laboratory environment at a maximum velocity of 0.6 m/s. In a corridor environment the control system guides MARVIN to its target position with 99% accuracy under normal conditions. If it is given false starting information the accuracy remains within 98%. The collision avoidance algorithm allows the control system to stop within 40 cm of a detected obstacle when travelling at its maximum speed.

Throughout the course of this project, MARVIN has been transformed from a (non-working) remotely guided vehicle into an autonomous device. The completed navigation and control system allows MARVIN to plan and execute a sequence of motion instructions that drive MARVIN to a designated location in a corridor or laboratory environment. Overall, the project can be considered a significant success, meeting (and in some cases exceeding) its objectives, and providing a robust system that can be expanded upon in future projects undertaken by the Mechatronics Group.

Appendix A: Circuit Schematics

A.1   Beacon Receiver Schematic
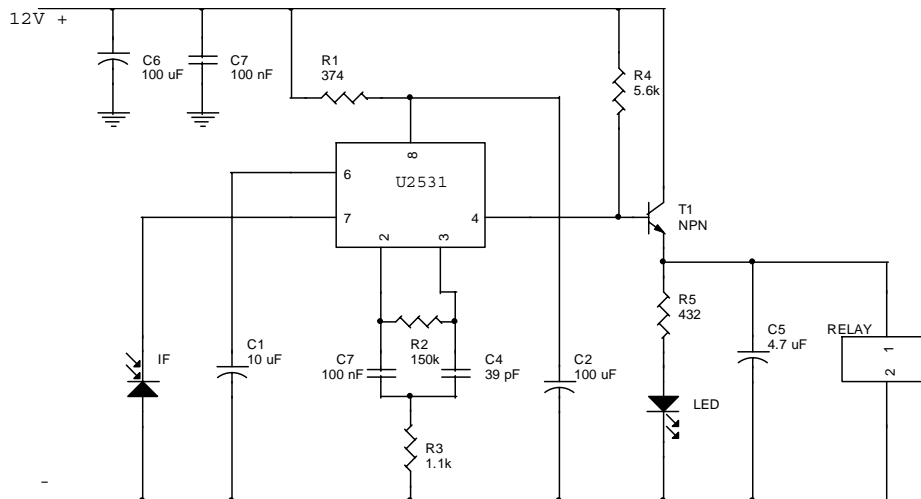
A.2   Beacon Emitter Schematic
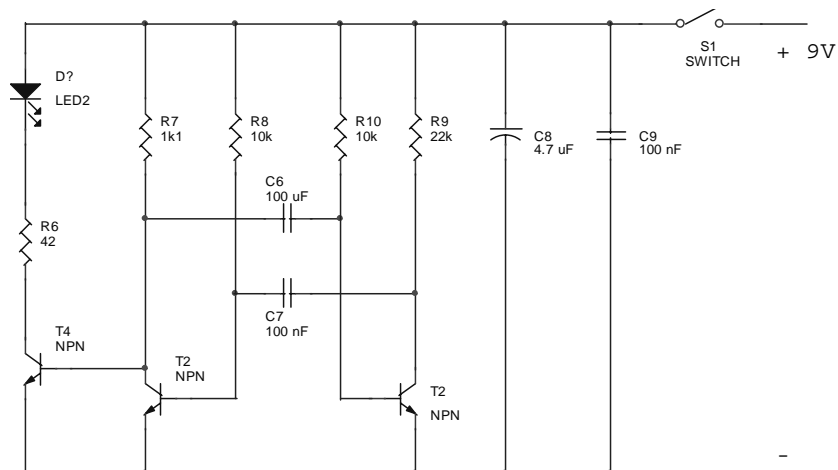
A.3   Motor Diver Schematic
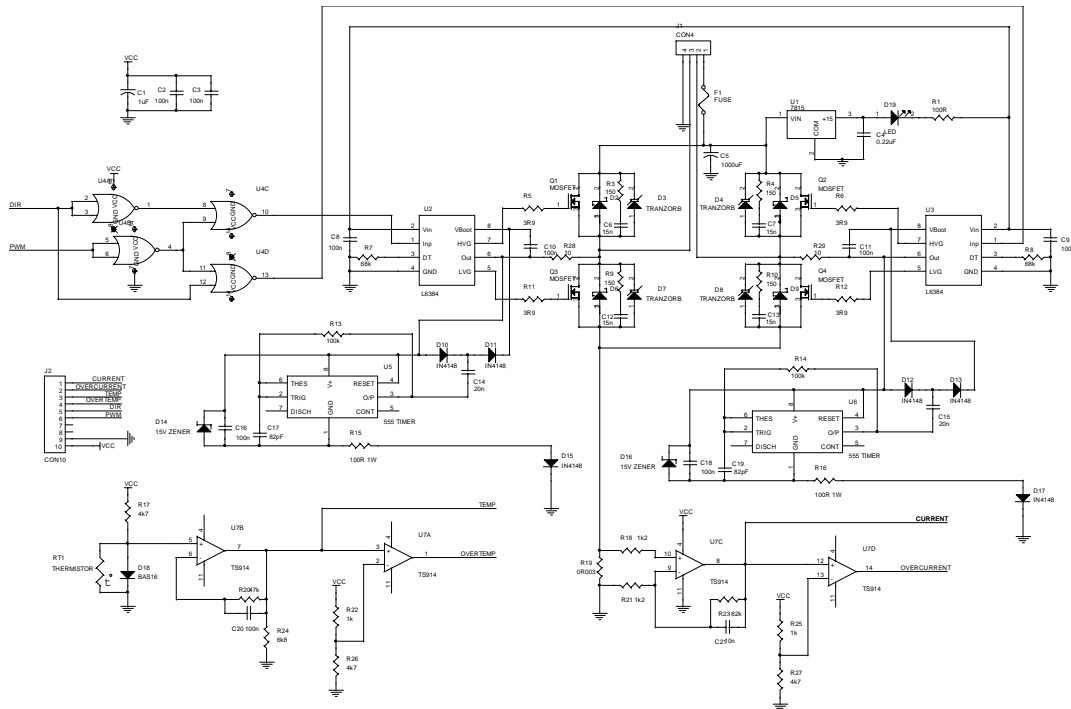
A.4   Motor Driver Schematic
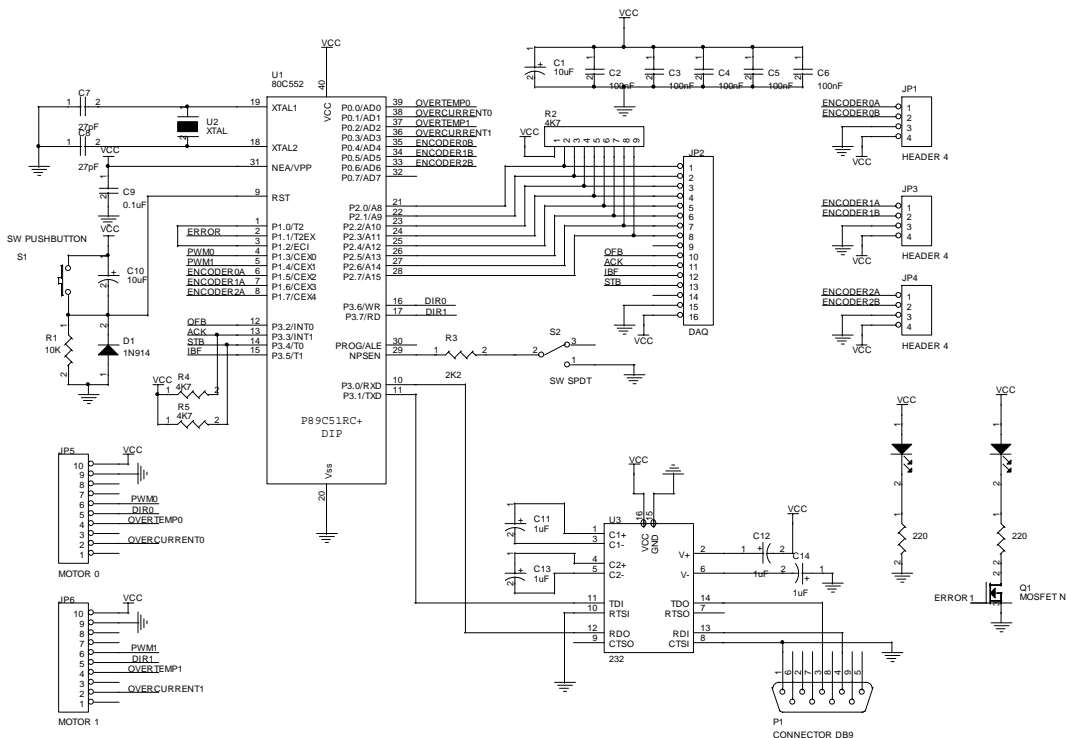
Appendix B: Source Code

B.1   gui_marvin_control.m

```
function varargout = gui_marvin_control(varargin)

% Chris Lee-Johnson
%
% GUI_MARVIN_CONTROL Application M-file for gui_marvin_control.fig
% FIG = GUI_MARVIN_CONTROL launch gui_marvin_control GUI.
% GUI_MARVIN_CONTROL('callback_name', ...) invoke the named callback.

if nargin == 0  % LAUNCH GUI

    fig = openfig(mfilename,'reuse');

    % Generate a structure of handles to pass to callbacks,
    % and store it.
    handles = guihandles(fig);
    handles.run = 0;
    handles.stop = 0;
    handles.execute = 0;
    handles.target_distance = 0;
    handles.target_angle = 0;
    handles.offset_angle = 0;
    handles.ir_weighting = 1;
    handles.corridor_angle = 0;
    handles.corridor_offset = 0;
    handles.wall_distance_1 = 0.84;
    handles.wall_distance_2 = 0.84;
    guidata(fig,handles);

    if nargout > 0
        varargout{1} = fig;
    end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK

    try
        if (nargout)
            [varargout{1:nargout}] = feval(varargin{:});
        else
            feval(varargin{:});
        end
    catch
        disp(lasterr);
    end

end


% ------------------------------------------------------------------
function varargout = run_Callback(h, eventdata, handles, varargin)

handles.run = get(h,'Value');
guidata(h,handles);

if handles.run == 1

    % Constants.
    N = 100;
    M = 1;
    X0 = 1.200;%14.1;%27;%
    Y0 = 6.025;%1.5;%
    HEADING0 = 0;
    X_WALL = [0.300,13.280,13.280,14.95,14.95,19.695,19.695,22.105,22.105,24.525,24.525,27.385,27.385,30.425, ...
        31.364,31.244,32.464,33.3588,31.7709,14.950,14.95,13.28,13.28,8.52,8.52,0.825,0.825,0.3,0.3];
    Y_WALL = [5.185,5.185,0.6,0.6,5.185,5.185,4.885,4.885,5.185,5.185,4.885,4.885,5.185,5.185,4.7865, ...
        4.6265,4.1085,6.2165,6.8650,6.865,11.45,11.45,6.865,6.865,7.165,7.165,6.865,6.865,5.185];

    % Initialise variables.
    x(1:2) = X0;
    y(1:2) = Y0;
    heading(1:2) = HEADING0;
    corridor_angle(1:2) = 0;
    w_vel_array = [0,0;0,0];
    time_plot = [0,0];
    line_count = 1;
    draw_count = 1;

    % Generate unique filename from time and date.
    fn_data = strcat(datestr(clock,30),'.txt');

    % Open file.
    file_data = fopen(strcat('c:\Project\Code\Data\',fn_data),'a');

    % Initialise marvin_control function.
    [time,tgt_x,tgt_y,tgt_heading,abs_x,abs_y,abs_heading,rel_x,rel_y,rel_heading,w_velocity, ...
        ir_y,ir_heading,ir_adj_x,ir_adj_y,ir_adj_angle,ir_obj_distance,contact_switch,beacon] ...
```

```
    = marvin_control(2,0,0,0,[0,0],handles.corridor_offset,handles.corridor_angle*pi, ...
    [-handles.wall_distance_1,handles.wall_distance_2],X0,Y0,HEADING0);

% Draw corridor walls etc.
axes(handles.position);
set(gca,'YDir','reverse');
set(gca,'XTick',[0:2:34]);
line(X_WALL,Y_WALL,'LineStyle','-');
line([0.300,31.244],[6.025,6.025],'LineStyle','--');
line([14.115,14.115],[11.450,0.600],'LineStyle','--');
axis image;
axis([0,34,0,12]);

% Draw MARVIN's path, heading and IRs.
for i = (1:N)
    traj_line(i) = line(X0,Y0,'Color','k');
end;
arrow_line = line(X0,Y0,'Color','k');
for i = 1:6
    ir_line(i) = line(X0,Y0,'Color','r');
end;

% Draw velocity profiles.
axes(handles.velocity);
for i = (1:N)
    vel_line_1(i) = line(0,0,'Color','b');
    vel_line_2(i) = line(0,0,'Color','r');
end;

% Update screen plot.
set(gcf,'DoubleBuffer','on');
drawnow;

while handles.run == 1

    % Get GUI handles.
    handles = guidata(h);

    % Set marvin_control inputs.
    if handles.stop == 1
        new_instruction = -1;
        target_distance = 0;
        target_angle = 0;
        offset_angle = 0;
    elseif handles.reset == 1
        new_instruction = 2;
        target_distance = 0;
        target_angle = 0;
        offset_angle = 0;
    else
        new_instruction = handles.execute;
        target_distance = handles.target_distance;
        target_angle = handles.target_angle*pi;
        offset_angle = handles.offset_angle*pi;
    end;
    ir_weighting = [handles.ir_weighting,handles.ir_weighting];
    corridor_offset = handles.corridor_offset;
    corridor_angle = handles.corridor_angle*pi;
    wall_y = [-handles.wall_distance_1,handles.wall_distance_2];

    % Reset button handles.
    handles.reset = 0;
    handles.execute = 0;
    handles.stop = 0;

    % Call marvin_control.
    [time,tgt_x,tgt_y,tgt_heading,abs_x,abs_y,abs_heading,rel_x,rel_y,rel_heading,w_velocity, ...
        ir_y,ir_heading,ir_adj_x,ir_adj_y,ir_adj_angle,ir_obj_distance,contact_switch,beacon] ...
        = marvin_control(new_instruction,target_distance,target_angle,offset_angle,ir_weighting, ...
        corridor_offset,corridor_angle,wall_y,X0,Y0,HEADING0);

    % Set current and last values for plotting.
    x(2) = x(1);
    y(2) = y(1);
    heading(2) = heading(1);
    x(1) = abs_x;
    y(1) = abs_y;
    heading(1) = abs_heading;
    w_vel_array(2,:) = w_vel_array(1,:);
    w_vel_array(1,:) = w_velocity;
    time_plot(2) = time_plot(1);
    time_plot(1) = time;

    % Convert integers to doubles so that fprinf can process them.
    contact_switch = double(contact_switch);
    beacon = double(beacon);

    % Save data to file.
    %
    % time:           1
    % tgt_x:          2
    % tgt_y:          3
    % tgt_heading:    4
    % abs_x:          5
    % abs_y:          6
    % abs_heading:    7
    % rel_x:          8
    % rel_y:          9
    % rel_heading     10
    % w_velocity:     11-12
    % ir_y:           13-14
    % ir_heading:     15-16
```

```
            % ir_adj_x:         17-22
            % ir_adj_y:         23-28
            % ir_adj_angle:     29-34
            % ir_obj_distance:  35-40
            % contact_switch:   41-44
            % beacon:           45-46
            %

fprintf(file_data,'%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t
%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\t%g\n',...
            [time;tgt_x;tgt_y;tgt_heading;abs_x;abs_y;abs_heading;rel_x;rel_y;rel_heading;w_velocity';ir_y'; ...
            ir_heading';ir_adj_x';ir_adj_y';ir_adj_angle';ir_obj_distance';contact_switch';beacon']);

            % Set text handles.
            set(handles.x,'String',num2str(x(1),'x = %0.3f m'));
            set(handles.y,'String',num2str(y(1),'y = %0.3f m'));
            set(handles.heading,'String',num2str(heading(1)/pi,'h = %0.3f*pi'));
            set(handles.xr,'String',num2str(rel_x,'xr = %0.3f m'));
            set(handles.yr,'String',num2str(rel_y,'yr = %0.3f m'));
            set(handles.hr,'String',num2str(rel_heading/pi,'hr = %0.3f*pi'));
            set(handles.ir_y_1,'String',num2str(ir_y(1),'IR y = %0.3f m'));
            set(handles.ir_h_1,'String',num2str(ir_heading(1)/pi,'IR h = %0.3f*pi'));
            set(handles.ir_y_2,'String',num2str(ir_y(2),'IR y = %0.3f m'));
            set(handles.ir_h_2,'String',num2str(ir_heading(2)/pi,'IR h = %0.3f*pi'));
            set(handles.ir1,'String',num2str(ir_obj_distance(1),'bl = %0.3f m'));
            set(handles.ir2,'String',num2str(ir_obj_distance(2),'fl = %0.3f m'));
            set(handles.ir3,'String',num2str(ir_obj_distance(3),'f = %0.3f m'));
            set(handles.ir4,'String',num2str(ir_obj_distance(4),'fr = %0.3f m'));
            set(handles.ir5,'String',num2str(ir_obj_distance(5),'br = %0.3f m'));
            set(handles.ir6,'String',num2str(ir_obj_distance(6),'b = %0.3f m'));
            if contact_switch(1) == 1
                set(handles.bump,'String','COLLISION, back left');
            elseif contact_switch(2) == 1
                set(handles.bump,'String','COLLISION, front left');
            elseif contact_switch(3) == 1
                set(handles.bump,'String','COLLISION, front right');
            elseif contact_switch(4) == 1
                set(handles.bump,'String','COLLISION, back right');
            end;

            % Write data to GUI.
            guidata(h,handles);

            % Get arrowhead line coordinates
            arrow_x = [x(1)-0.5*cos(adjust_angle(heading-pi/8)),x(1),x(1)-0.5*cos(adjust_angle(heading+pi/8))];
            arrow_y = [y(1)-0.5*sin(adjust_angle(heading-pi/8)),y(1),y(1)-0.5*sin(adjust_angle(heading+pi/8))];

            % Absolute positions of detected objects.
            for i = 1:6
                if ir_obj_distance(i) > 1.5
                    ir_obj_x(i) = NaN;
                    ir_obj_y(i) = NaN;
                else
                    ir_obj_x(i) = ir_adj_x(i) + ir_obj_distance(i)*cos(ir_adj_angle(i));
                    ir_obj_y(i) = ir_adj_y(i) + ir_obj_distance(i)*sin(ir_adj_angle(i));
                end;
            end;

            % Draw MARVIN's path, heading and IRs.
            axes(handles.position);
            set(traj_line(line_count),'XData',x,'YData',y);
            set(arrow_line,'XData',arrow_x,'YData',arrow_y);
            for i = 1:6
                set(ir_line(i),'XData',[ir_adj_x(i),ir_obj_x(i)],'YData',[ir_adj_y(i),ir_obj_y(i)]);
            end;

            % Draw velocity profile.
            axes(handles.velocity);
            set(vel_line_1(line_count),'XData',[time_plot(1),time_plot(2)], ...
                'YData',[w_vel_array(1,1),w_vel_array(2,1)]);
            set(vel_line_2(line_count),'XData',[time_plot(1),time_plot(2)], ...
                'YData',[w_vel_array(1,2),w_vel_array(2,2)]);

            % Update screen plot every M cycles.
            if draw_count == 1
                drawnow;
            end;

            % Increment counters.
            draw_count = mod(draw_count,M)+1;
            line_count = mod(line_count,N)+1;

        end;

        % Close data file.
        fclose(file_data);

end;
```

B.2   marvin_control.m

```
function [time,tgt_x,tgt_y,tgt_heading,abs_x,abs_y,abs_heading,rel_x,rel_y,rel_heading,w_vel,ir_y,ir_heading, ...
    ir_adj_x,ir_adj_y,ir_adj_angle,ir_obj_distance,contact_switch,beacon] = marvin_control(new_instruction, ...
    new_tgt_distance,new_tgt_angle,new_ofst_angle,ir_weighting,corridor_y,corridor_angle,wall_y,init_x, ...
    init_y,init_heading)

% Chris Lee-Johnson
%
% marvin_control.m - Main motion control system for MARVIN.
% Note: On first call, new_instruction must be set to 2.
%
% ----------------------------------------------------------
% Outputs
% ----------------------------------------------------------
% time:             Time elapsed (s)
% abs_x:            Absolute x position coordinate (m)
% abs_y:            Absolute y position coordinate (m)
% abs_heading       Absolute heading (rad)
% rel_x:            Distance along corridor (m)
% rel_y:            Offset from corridor centre axis (m)
% rel_heading:      Heading relative to centre axis (rad)
% tgt_x:            Distance on target trajectory (m)
% tgt_y:            Offset on target trajectory (m)
% tgt_heading:      Heading on target trajectory (rad)
% w_vel:            Velocity of MARVIN's wheels (m/s)
%                   [left, right]
% ir_obj_distance:  Array of IR rangefinder distances (m)
%                   [left back, left front, front,
%                   right front, right back, back]
% contact_switch:   Array of contact switch values (0,1)
%                   [left back, left front,
%                   right front, right back]
% ----------------------------------------------------------
% Inputs
% ----------------------------------------------------------
% new_instruction:  New instruction flag
%                   -1: brake
%                    0: no new instruction
%                    1: new instruction
%                    2: first instruction
% new_tgt_distance: Distance between origin & destination (m)
% new_tgt_angle:    Target angle to turn through (-pi:pi rad)
% new_ofst_angle:   Offset of target angle (-pi:pi rad)
%                   (use for small heading adjustments)
% ir_weighting:     Weightings for IR rangefinders (0:1)
%                   [left,right (facing corridor angle)]
% corridor_y:       Offset from centre axis of corridor (m)
% corridor_angle:   Absolute direction of corridor (rad)
% wall_y:           Wall offsets from centre axis (m)
%                   [left,right (facing corridor angle)]
% init_x:           Initial absolute x coordinate (m)
% init_y:           Initial absolute y coordinate (m)
% init_heading:     Initial absolute heading (rad)

% Simulation flag (0:real, 1:simulation)
SIM = 1;

% Encoder counts per metre.
COUNTS_PER_M = 28062*[1.0035,0.9965];

% Distance between MARVIN's wheels (m)
W_SEPARATION = 0.508;

% IR rangefinder origins (relative to MARVIN's origin).
% (left back, left front, front, right front, right back, back)
TILT = 0.12062;
IR_OGN_X = [-0.0745,0.0745,0.1445,0.0745,-0.0745,-0.1445];
IR_OGN_Y = [-0.1415,-0.1415,0,0.1415,0.1415,0];
IR_OGN_ANGLE = [-pi/2-TILT,-pi/2+TILT,0,pi/2-TILT,pi/2+TILT,pi];

% Persistent variables.
persistent lvserv;
persistent target_distance;
persistent target_angle;
persistent offset_angle;
persistent w_pos_prof;
persistent w_vel_prof;
persistent w_section;
persistent w_tgt_pos;
persistent w_tgt_vel;
persistent w_velocity;
persistent w_dir;
persistent velocity_limit;
persistent x;
persistent y;
persistent heading;
persistent x0;
persistent y0;
persistent heading0;
persistent abs_x0;
persistent abs_y0;
persistent abs_heading0;
persistent tgt_trj_x;
persistent tgt_trj_y;
persistent tgt_trj_heading;
persistent head_error;
```

```
persistent brake;
persistent w_sim_vel;

% If first instruction, set up LABVIEW interface and initialise
% variables.
if new_instruction >= 2

    % Initialise persistent variables.
    abs_x0 = init_x + corridor_y*sin(corridor_angle);
    abs_y0 = init_y - corridor_y*cos(corridor_angle);
    abs_heading0 = corridor_angle;
    x = 0;
    y = corridor_y;
    heading = adjust_angle(init_heading-corridor_angle);
    w_velocity = [0,0];
    w_tgt_vel = [0,0];
    head_error = [0,0,0];
    w_dir = [1,1];
    brake = 0;

    if SIM

        % Simulation - initialise "actual" wheel speeds.
        w_sim_vel = [0,0];

    else

        % Set up LabVIEW ActiveX server.
        lvserv = actxserver('LabVIEW.Application');

    end;

end;

if SIM

    % Simulation - Get wheel encoder counts.
    [w_count_diff,time_diff,time] = sim_en_count(new_instruction,w_sim_vel,COUNTS_PER_M);

    % Simulation - Read infra-red rangefinders.
    [ir_voltage] = sim_ir_voltage(new_instruction,x,y,heading,wall_y,IR_OGN_X,IR_OGN_Y,IR_OGN_ANGLE);

    % Simulation - Read contact switch.
    contact_switch = [0,0,0,0];
    beacon = [0,0];

else

    % Get wheel encoder counts from LabVIEW VI.
    [w_count_diff,time_diff,time] = acq_en_count(new_instruction,lvserv,w_velocity,w_tgt_vel);

    % Read infra-red rangefinders.
    [ir_voltage] = acq_ir_voltage(new_instruction,lvserv);

    % Read contact switch.
    [contact_switch,beacon] = acq_switch(new_instruction,lvserv);

end;

% Calculate MARVIN's wheel velocities.
[w_velocity,w_vel_filt,period] = rep_en_velocity(new_instruction,w_velocity,w_count_diff,time_diff,COUNTS_PER_M);

if new_instruction ~= 0

    % Set/reset brake flag.
    if new_instruction == -1
        brake = 1;
    else
        brake = 0;
    end;

    % Record target distance and angles.
    target_distance = new_tgt_distance;
    target_angle = new_tgt_angle;
    offset_angle = new_ofst_angle;

    % If corridor direction changes, reset distance, offset & heading.
    [abs_x0,abs_y0,abs_heading0,x,y,heading] = rel_coord(new_instruction,abs_x0,abs_y0,abs_heading0,x,y, ...
        heading,corridor_y,corridor_angle);

    % Record initial conditions.
    x0 = x;
    y0 = y;
    heading0 = heading;

    % Initialise velocity profile section variable to first section.
    w_section = [1,1];

    % Get target position of each wheel.
    [w_tgt_pos] = wheel_pos(target_distance,target_angle,W_SEPARATION);

    % Calculate velocity profile for fastest wheel.
    [w_pos_prof,w_vel_prof,w_dir,velocity_limit] = gen_vel_prof(target_distance,target_angle, ...
        w_dir,w_tgt_pos,w_velocity,velocity_limit);

    % Plot target trajectory.
    [tgt_trj_x,tgt_trj_y,tgt_trj_heading] = gen_tgt_trj(target_distance,target_angle, ...
        offset_angle,x0,y0,heading0,w_tgt_pos,wall_y);

end;

% Calculate MARVIN's cartesian coordinates and heading.
```

```
[x,y,heading] = rep_en_coord(x,y,heading,w_count_diff, ...
    COUNTS_PER_M,W_SEPARATION);

if SIM
    % Simulation - Prevent MARVIN from crossing walls.
    HALF_WIDTH = 0.292;    % Half of MARVIN's width
    if y < wall_y(1) + HALF_WIDTH
        y = wall_y(1) + HALF_WIDTH;
    elseif y > wall_y(2) - HALF_WIDTH
        y = wall_y(2) - HALF_WIDTH;
    end;
end;

% Measure IR distances.
[ir_obj_distance] = rep_ir_distance(ir_voltage);

% Measure offset and heading from IR values.
[ir_y,ir_heading] = rep_ir_coord(new_instruction,ir_obj_distance,y,heading,wall_y, ...
    IR_OGN_X,IR_OGN_Y,IR_OGN_ANGLE);

% Adjust coordinates and heading by comparing sensor data.
[x,y,heading] = sensor_fusion(new_instruction,x,y,heading,ir_y,ir_heading,ir_weighting);

% Get difference between target and actual headings.
[head_error,proportion,tgt_x,tgt_y,tgt_heading] = heading_error(head_error,target_distance,x,y,heading, ...
    tgt_trj_x,tgt_trj_y,tgt_trj_heading,ir_obj_distance,w_tgt_pos);

% Get uncorrected target wheel velocities from velocity profile.
[w_tgt_vel,w_section] = tgt_velocity(w_section,proportion,w_tgt_pos,w_pos_prof,w_vel_prof,w_vel_filt,period);

% Get PID control errors for each wheel.
[w_vel_error,w_vel_error_filt] = heading_control(new_instruction,period,w_tgt_vel,w_velocity, ...
    w_vel_filt,head_error,time_diff);

if SIM
    % Simulation - Set target velocity.
    w_tgt_vel = w_vel_error(:,1)' + w_velocity;
else
    % Apply PID control to target velocities.
    [w_tgt_vel] = velocity_control(new_instruction,w_tgt_vel,w_vel_error,w_vel_error_filt,w_velocity,w_dir, ...
        period,velocity_limit);
end;

% Stop wheels immediately in the event of an impending collision or a stop instruction.
[brake,w_tgt_vel,w_section] = stop_wheel(brake,w_tgt_vel,w_section,contact_switch, ...
    ir_obj_distance,target_distance,target_angle);

% Convert speeds into PWM values.
[w_pwm] = get_motor_power(w_tgt_vel);

if SIM
    % Simulation - Set motor power.
    [w_sim_vel] = sim_motor_power(new_instruction,w_tgt_vel);
else
    % Set motor power in LabVIEW VI.
    [error] = set_motor_power(new_instruction,brake,w_pwm,w_dir,lvserv);
end;

% Coordinates conversions.
[abs_x,abs_y,abs_heading] = coord_trans(abs_x0,abs_y0,abs_heading0,x,y,heading);
[tgt_x,tgt_y,tgt_heading] = coord_trans(abs_x0,abs_y0,abs_heading0,tgt_x,tgt_y,tgt_heading);
[ir_adj_x,ir_adj_y,ir_adj_angle] = coord_trans(abs_x,abs_y,abs_heading,IR_OGN_X,IR_OGN_Y,IR_OGN_ANGLE);

% Assign new variable names for returned persistant variables
% (since MATLAB doesn't allow uninitialised variables to be
% returned).
rel_x = x;
rel_y = y;
rel_heading = heading;
w_vel = w_velocity;
```

B.3 acq_en_count.m

```
function [w_count_diff,time_diff,time] = acq_en_count(new_instruction,lvserv,w_velocity,w_tgt_vel)

% Chris Lee-Johnson
%
% Function to obtain wheel counts from a LabVIEW VI
%
% w_count_diff:     Wheel displacements since last cycle (counts)
% time_diff:        Time since last cycle (s)
% time:             Total time elapsed (s)
% new_instruction:  New instruction flag
% lvserv:           LabVIEW ActiveX server object
% w_velocity:       Wheel velocities (m/s)
% w_tgt_vel:        Target wheel velocities (m/s)

% Persistent variables
persistent old_time;
persistent first_time;
persistent w_count;
persistent encoder_counter_vi;

% If first instruction, initialise variables, set up LABVIEW
% interface and initialise counters.
if new_instruction == 2

    % Initialise variables.
    first_time = cputime;
    old_time = 0;
    w_count = [0,0;0,0];

    % Encoder Counter VI.
    encoder_counter_vi = invoke(lvserv,'GetViReference','c:\Project\Code\LabVIEW\Encoder Counter.vi');

    % Initialise counters.
    invoke(encoder_counter_vi,'SetControlValue','iteration',num2str(0));

else
    invoke(encoder_counter_vi,'SetControlValue','iteration',num2str(1));
end;

% Get elapsed time between cycles.
time = cputime-first_time;
time_diff = time-old_time;
while time_diff < 0.08
    time = cputime-first_time;
    time_diff = time-old_time;
end;
old_time = time;

% Run Encoder Counter VI.
encoder_counter_vi.Run;

% Get current and last counter values.
w_count(2,:) = w_count(1,:);
w_count(1,1) = invoke(encoder_counter_vi,'GetControlValue','count 1');
w_count(1,2) = invoke(encoder_counter_vi,'GetControlValue','count 2');

for i = 1:2

    % Get difference between current and last counter values.
    if w_count(1,i) >= w_count(2,i)
        w_count_diff(i) = w_count(1,i) - w_count(2,i);
    else
        % If counter overflows.
        w_count_diff(i) = w_count(1,i) - w_count(2,i) + 16777216;
    end;

    % Set encoder counts according to wheels' turning directions.
    if (w_velocity(i) == 0 & w_tgt_vel(i) < 0) | (w_velocity(i) < 0 & w_tgt_vel(i) <= 0)
        w_count_diff(i) = -w_count_diff(i);
    elseif (w_velocity(i) > 0 & w_tgt_vel(i) < 0) | (w_velocity(i) < 0 & w_tgt_vel(i) > 0)
        w_count_diff(i) = 0;
    end;

end;
```

B.4   acq_ir_voltage.m

```
function [ir_voltage] = acq_ir_voltage(new_instruction,lvserv)

% Chris Lee-Johnson
%
% Function to obtain each infrared rangefinder's voltage from
% a LabVIEW VI.
%
% ir_voltage:      Array of voltages output from rangefinders
%                  [left back, left front, front,
%                  right front, right back, back]
% new_instruction: New instruction flag
% lvserv:          LabVIEW ActiveX server object

% Persistent variables
persistent ir_analogue_input_vi;
persistent ir_inst_voltage;

N = 9;

% If first instruction, set up LABVIEW interface.
if new_instruction == 2

    % IR Analogue Input VI.
    ir_analogue_input_vi = invoke(lvserv,'GetViReference','c:\Project\Code\LabVIEW\IR Analogue Input.vi');

    % Initialise variables.
    ir_inst_voltage(1:N,1:6) = 0;

end;

for i = N-1:-1:1
    ir_inst_voltage(i+1,:) = ir_inst_voltage(i,:);
end;

% Run IR Analogue Input VI.
ir_analogue_input_vi.Run;

% Get voltage values.
ir_inst_voltage(1,1) = invoke(ir_analogue_input_vi,'GetControlValue','Analogue Input 1');
ir_inst_voltage(1,2) = invoke(ir_analogue_input_vi,'GetControlValue','Analogue Input 2');
ir_inst_voltage(1,3) = invoke(ir_analogue_input_vi,'GetControlValue','Analogue Input 3');
ir_inst_voltage(1,4) = invoke(ir_analogue_input_vi,'GetControlValue','Analogue Input 4');
ir_inst_voltage(1,5) = invoke(ir_analogue_input_vi,'GetControlValue','Analogue Input 5');
ir_inst_voltage(1,6) = invoke(ir_analogue_input_vi,'GetControlValue','Analogue Input 6');

% Software filter to reduce noise.
ir_voltage = median(ir_inst_voltage);
```

B.5   acq_switch.m

```
function [contact_switch,beacon] = acq_switch(new_instruction,lvserv)

% Chris Lee-Johnson
%
% Function to obtain the states of tactile sensors and beacon
% receivers.
%
% contact_switch:  Array of switch inputs
%                  [left back, left front,
%                  right front, right back]
% beacon:          Array of beacon receiver inputs
%                  [left, right]
% new_instruction: New instruction flag
% lvserv:          LabVIEW ActiveX server object

% Persistent variables
persistent digital_switch_input_vi;
persistent iteration;

% If first instruction, set up LABVIEW interface and initialise
% variables.
if new_instruction == 2

    % Digital Switch Input VI.
    digital_switch_input_vi = invoke(lvserv,'GetViReference','c:\Project\Code\LabVIEW\Digital Switch Input.vi');

    iteration = 0;

end;

% Set iteration variable.
invoke(digital_switch_input_vi,'SetControlValue','iteration (0:initialize)',num2str(iteration));

% Run Digital Switch Input VI.
```

```
digital_switch_input_vi.Run;

% Get switch values.
contact_switch(1) = invoke(digital_switch_input_vi,'GetControlValue','switch 1');
contact_switch(2) = invoke(digital_switch_input_vi,'GetControlValue','switch 2');
contact_switch(3) = invoke(digital_switch_input_vi,'GetControlValue','switch 3');
contact_switch(4) = invoke(digital_switch_input_vi,'GetControlValue','switch 4');
beacon(1) = invoke(digital_switch_input_vi,'GetControlValue','switch 5');
beacon(2) = invoke(digital_switch_input_vi,'GetControlValue','switch 6');

iteration = iteration + 1;
```

B.6   rep_en_velocity.m

```
function [w_velocity,w_vel_filt,period] = rep_en_velocity(new_instruction, ...
    w_velocity,w_count_diff,time_diff,COUNTS_PER_M)

% Chris Lee-Johnson
%
% Function to obtain MARVIN's wheel velocities from the
% odometer data.
%
% w_velocity:       Wheel velocities (m/s)
% w_vel_filt:       Filtered wheel velocities (m/s)
% period:           Control cycle period (s)
% new_instruction:  New instruction flag
% w_count_diff:     Wheel displacements (encoder counts)
% time_diff:        Time since last cycle (s)
% COUNTS_PER_M:     Number of encoder counts in 1m

% Convert wheel counts to metres.
w_dist_diff = w_count_diff ./ COUNTS_PER_M;

% Calculate wheel velocities.
if time_diff > 0
    w_velocity = w_dist_diff / time_diff;
end;

% Persistent variables.
persistent w_vel_array;
persistent time_diff_array;

% Number of samples for averaging.
N = 10;

% Initialise FIFO arrays.
if new_instruction == 2
    w_vel_array(1:N,1) = w_velocity(1);
    w_vel_array(1:N,2) = w_velocity(2);
    time_diff_array(1:N) = time_diff;
end;

% Update FIFO arrays.
for i = N-1:-1:1
    w_vel_array(i+1,:) = w_vel_array(i,:);
    time_diff_array(i+1) = time_diff_array(i);
end;

% Filter measured velocities.
w_vel_array(1,:) = w_velocity;
weight = 0.5;
w_vel_filt = [0,0];
for i = 1:N
   w_vel_filt = w_vel_filt  + weight * w_vel_array(i,:);
   if i < N-1
       weight = 0.5*weight;
   end;
end;

% Obtain median control cycle period.
time_diff_array(1) = time_diff;
period = median(time_diff_array);
```

B.7   rep_en_coord.m

```
function [x,y,heading] = rep_en_coord(x,y,heading,w_count_diff,COUNTS_PER_M,W_SEPARATION)

% Chris Lee-Johnson
%
% Function to obtain MARVIN's cartesian coordinates and
% heading from the encoder data.
%
% x:            Distance along axis parallel to corridor (m)
% y:            Distance from corridor centre axis (m)
% heading:      Heading (radians)
% w_count_diff: Wheel displacements (encoder counts)
% COUNTS_PER_M: Number of encoder counts in 1m
% W_SEPARATION: Distance between wheels (m)

% Angle correction factor.
ANGLE_MOD = 0.97148639449454;

% Convert wheel counts to metres.
w_dist_diff = w_count_diff ./ COUNTS_PER_M;

% Convert individual wheel displacements to overall arclength and angle.
angle = ANGLE_MOD * (w_dist_diff(1) - w_dist_diff(2)) / W_SEPARATION;
arclength = 0.5 * (w_dist_diff(1) + w_dist_diff(2));

% Obtain straight line distance from arclength and angle.
if abs(angle) < 0.0001
    distance = arclength;
else
    if arclength >= 0
        distance = abs(arclength/angle) * sqrt(2*(1-cos(angle)));
    else
        distance = -abs(arclength/angle) * sqrt(2*(1-cos(angle)));
    end;
end;

% Update co-ordinates from distance and angle.
heading = adjust_angle(heading+angle);
x = x + distance * cos(heading);
y = y + distance * sin(heading);
```

B.8   rep_ir_distance.m

```
function [ir_obj_distance] = rep_ir_distance(ir_voltage)

% Chris Lee-Johnson
%
% Function to convert rangefinder voltage readings into distances.
%
% ir_obj_distance:  Array of distances measured by rangefinders (m)
%                   [left back, left front, front,
%                   right front, right back, back]
% ir_voltage:       Array of voltages output from rangefinders (V)

% Look-up table parameters for IR voltage-distance relationships.
ir_dist_curve = [0.15:0.05:1.5];
ir_volt_curve = [2.790,2.560,2.300,1.950,1.700,1.500,1.350,1.210,1.100, ...
                1.010,0.935,0.865,0.805,0.750,0.700,0.665,0.625,0.595, ...
                0.565,0.540,0.515,0.495,0.480,0.460,0.445,0.430,0.420,0.410;
                2.750,2.500,2.235,1.900,1.650,1.460,1.305,1.175,1.070, ...
                0.990,0.910,0.850,0.790,0.730,0.690,0.650,0.610,0.585, ...
                0.555,0.530,0.505,0.485,0.465,0.445,0.430,0.415,0.400,0.390;
                2.950,2.665,2.300,1.930,1.720,1.520,1.350,1.225,1.120, ...
                1.040,0.955,0.900,0.830,0.775,0.725,0.685,0.640,0.600, ...
                0.565,0.545,0.510,0.495,0.475,0.455,0.445,0.430,0.415,0.410;
                2.800,2.575,2.305,1.990,1.725,1.510,1.370,1.230,1.125, ...
                1.030,0.950,0.885,0.840,0.790,0.745,0.700,0.665,0.630, ...
                0.600,0.575,0.550,0.530,0.510,0.495,0.480,0.465,0.455,0.450;
                2.680,2.460,2.225,1.905,1.650,1.470,1.320,1.180,1.080, ...
                1.000,0.925,0.860,0.810,0.760,0.715,0.670,0.630,0.595, ...
                0.565,0.540,0.515,0.495,0.480,0.465,0.450,0.440,0.430,0.425;
                2.760,2.500,2.240,1.900,1.655,1.475,1.325,1.200,1.085, ...
                0.995,0.920,0.855,0.800,0.750,0.710,0.675,0.645,0.605, ...
                0.580,0.550,0.525,0.500,0.475,0.450,0.430,0.415,0.400,0.390];

N = length(ir_dist_curve);

for i = 1:6

    % Record the section that contains the current voltage value.
    ir_section(i) = 0;
    for j = [1:N-1]
        if (ir_voltage(i) <= ir_volt_curve(i,j+1) & ir_voltage(i) >= ir_volt_curve(i,j)) ...
            | (ir_voltage(i) >= ir_volt_curve(i,j+1) & ir_voltage(i) <= ir_volt_curve(i,j))
            % Record current section.
```

```
            ir_section(i) = j;
        end;
    end;

    % If current position value is outside curve boundaries.
    if ir_section(i) == 0

        if (ir_voltage(i) < ir_volt_curve(i,N))
            ir_obj_distance(i) = Inf;
        else
            ir_obj_distance(i) = 0;
        end;

    else

        % Calculate distance for current section.
        if ir_volt_curve(i,ir_section(i)+1) == ir_volt_curve(i,ir_section(i))
            ir_obj_distance(i) = ir_dist_curve(ir_section(i));
        else
            ir_obj_distance(i) = (ir_voltage(i) - ir_volt_curve(i,ir_section(i))) ...
                * (ir_dist_curve(ir_section(i)+1) - ir_dist_curve(ir_section(i))) ...
                / (ir_volt_curve(i,ir_section(i)+1) - ir_volt_curve(i,ir_section(i))) ...
                + ir_dist_curve(ir_section(i));
        end;

    end;

end;
```

B.9   rep_ir_coord.m

```
function [ir_y,ir_heading] = rep_ir_coord(new_instruction,ir_obj_distance, ...
    y,heading,wall_y,IR_OGN_X,IR_OGN_Y,IR_OGN_ANGLE)

% Chris Lee-Johnson
%
% Function to obtain the offset and heading from the readings
% of the IR rangefinders.
%
% new_instruction:  New instruction flag
% ir_y:             Offset measured by IR rangefinders (m)
%                   [left,right (facing corridor angle)]
% ir_heading:       Heading measured by IR rangefinders (rad)
%                   [left,right (facing corridor angle)]
% ir_obj_distance:  Array of distances measured by rangefinders (m)
%                   [left back, left front, front,
%                   right front, right back, back]
% y:                Corridor offset (m)
% heading:          Heading (rad)
% wall_y:           Left and right wall offsets (m)

persistent ir_inst_obj_dist;

N = 10;
MAX_OBJ_DIFF = 0.2;

% Maximum difference between encoder and IR values allowed before
% IRs are rejected.
MAX_Y_DIFF = 0.5;
MAX_HEAD_DIFF = 0.5*pi;

% Offset and orientation of each IR (adjusted for MARVIN's overall
% position and orientation).
[ir_adj_x,ir_adj_y,ir_adj_angle] = coord_trans(0,y,heading,IR_OGN_X,IR_OGN_Y,IR_OGN_ANGLE);

% If first instruction, set up LABVIEW interface.
if new_instruction == 2

    % Initialise variables.
    for i = 1:N
        ir_inst_obj_dist(i,:) = ir_obj_distance;
    end;

end;

% Update old instantaneous IR object distance measurements.
for i = N-1:-1:1
    ir_inst_obj_dist(i+1,:) = ir_inst_obj_dist(i,:);
end;

% Update new instantaneous measurements.
for i = 1:6
    if ir_obj_distance(i) <= 1.5
        ir_inst_obj_dist(1,i) = ir_obj_distance(i);
    else
        ir_inst_obj_dist(1,i) = 1.5;
    end;
end;

% Average of instantaneous measurements.
ir_mean_obj_dist = mean(ir_inst_obj_dist);
```

```
for i = 1:6

    % Expected object distances due to walls.
    if ir_adj_angle(i) < 0
        ir_ex_obj_dist(i) = abs( (wall_y(1)-ir_adj_y(i)) / sin(ir_adj_angle(i)) );
    elseif ir_adj_angle(i) > 0
        ir_ex_obj_dist(i) = abs( (wall_y(2)-ir_adj_y(i)) / sin(ir_adj_angle(i)) );
    else
        ir_ex_obj_dist(i) = Inf;
    end;

    % If expected distance is greater than 1.2m, or if measured
    % distance changes too quickly, don't use that IR for
    % heading/offset measurements.
    if ir_ex_obj_dist(i) <= 1.2 & abs(ir_obj_distance(i)-ir_mean_obj_dist(i)) <= MAX_OBJ_DIFF
        ir_sel_obj_dist(i) = ir_obj_distance(i);
    else
        ir_sel_obj_dist(i) = Inf;
    end;

end;

% Detected object coordinates (relative to MARVIN's origin).
for i = 1:6
    if ir_sel_obj_dist(i) <= 1.5
        obj_x(i) = IR_OGN_X(i) + ir_sel_obj_dist(i) * cos(IR_OGN_ANGLE(i));
        obj_y(i) = IR_OGN_Y(i) + ir_sel_obj_dist(i) * sin(IR_OGN_ANGLE(i));
    else
        obj_x(i) = NaN;
        obj_y(i) = NaN;
    end;
end;

ir_y_count(1:2) = 0;
ir_head_count(1:2) = 0;
ir_y_array(1:6,1:2) = 0;
ir_head_array(1:6,1:2) = 0;

for i = 1:6

    j = mod(i,6)+1;

    % Measured offset.
    ir_wall_dist = obj_x(i)*sin(heading) + obj_y(i)*cos(heading);
    if ir_adj_angle(i) < 0
        ir_y_tmp = wall_y(1) - ir_wall_dist;
        k = 1;
    else
        ir_y_tmp = wall_y(2) - ir_wall_dist;
        k = 2;
    end;

    % Select IR heading that is closest to encoder heading.
    ir_head(1) = -atan2(obj_y(j)-obj_y(i),obj_x(j)-obj_x(i));
    ir_head(2) = adjust_angle(ir_head(1)+pi);
    [head_diff,index] = min(abs(heading-ir_head));
    ir_head_tmp = ir_head(index);

    % Rejected IR values.
    if ir_sel_obj_dist(i) > 1.5
        ir_y_tmp = NaN;
        ir_head_tmp = NaN;
    elseif (ir_adj_angle(i) > 0 & ir_adj_angle(j) < 0) | (ir_adj_angle(i) < 0 & ir_adj_angle(j) > 0)
        ir_head_tmp = NaN;
    end;

    % Difference between IR and encoder coordinates.
    y_diff = abs(y-ir_y_tmp);
    head_diff = abs(heading-ir_head_tmp);

    % Reject IR offsets that deviate too far from encoder offset.
    if y_diff < MAX_Y_DIFF
        ir_y_count(k) = ir_y_count(k) + 1;
        ir_y_array(ir_y_count(k),k) = ir_y_tmp;
    end;

    % Reject IR headings that deviate too far from encoder heading.
    if head_diff < MAX_HEAD_DIFF
        ir_head_count(k) = ir_head_count(k) + 1;
        ir_head_array(ir_head_count(k),k) = ir_head_tmp;
    end;

end;

for k = 1:2

    % Final IR offset is mean average of valid IR offsets.
    if ir_y_count(k) > 0
        ir_y(k) = mean(ir_y_array(1:ir_y_count(k),k));
    else
        ir_y(k) = NaN;
    end;

    % Final IR heading is mean average of valid IR headings.
    if ir_head_count(k) > 0
        ir_heading(k) = mean(ir_head_array(1:ir_head_count(k),k));
    else
        ir_heading(k) = NaN;
    end;

end;
```

B.10 coord_trans.m

```
function [axis1_x,axis1_y,axis1_theta] = coord_trans(axis1_x0,axis1_y0,axis1_theta0,axis2_x,axis2_y,axis2_theta)

% Chris Lee-Johnson
%
% Function to apply coordinate transformations.
%
% axis1_x:         First axis x coordinate
% axis1_y:         First axis y coordinate
% axis1_theta:     First axis theta coordinate
% axis1_x0:        First axis x origin
% axis1_y0:        First axis y origin
% axis1_theta0:    First axis theta origin
% axis2_x:         Second axis x coordinate
% axis2_y:         Second axis y coordinate
% axis2_heading:   Second axis theta coordinate

axis1_x = axis1_x0 + axis2_x*cos(axis1_theta0) - axis2_y*sin(axis1_theta0);
axis1_y = axis1_y0 + axis2_x*sin(axis1_theta0) + axis2_y*cos(axis1_theta0);
axis1_theta = adjust_angle(axis2_theta+axis1_theta0);
```

B.11 rel_coord.m

```
function [abs_x0,abs_y0,abs_heading0,x,y,heading] = rel_coord(new_instruction,abs_x0,abs_y0, ...
    abs_heading0,x,y,heading,corridor_y,corridor_angle)

% Chris Lee-Johnson
%
% Function to obtain relative coordinates and absolute origin.
%
% abs_x0:          Absolute x position of origin (m)
% abs_y0:          Absolute y position of origin (m)
% abs_heading0:    Absolute heading of origin (rad)
% x:               Distance along corridor centre axis (m)
% y:               Offset from centre axis of corridor (m)
% heading:         Heading relative to corridor angle (rad)
% new_instruction: New instruction flag
% corridor_y:      Offset from centre axis of new corridor (m)
% corridor_angle:  Absolute direction of new corridor (rad)

% Persistent variables.
persistent cor_ang_array;
persistent cor_ang_diff;

% If first instruction, initialise.
if new_instruction == 2
    cor_ang_array = [corridor_angle,corridor_angle];
end;

% Update corridor angles.
cor_ang_array(1) = cor_ang_array(2);
cor_ang_array(2) = corridor_angle;
cor_ang_diff = adjust_angle(cor_ang_array(2)-cor_ang_array(1));

if cor_ang_diff ~= 0

    % Get absolute origins of relative values.
    abs_x0 = abs_x0 + x*cos(abs_heading0) - y*sin(abs_heading0) + corridor_y*sin(abs_heading0+cor_ang_diff);
    abs_y0 = abs_y0 + x*sin(abs_heading0) + y*cos(abs_heading0) - corridor_y*cos(abs_heading0+cor_ang_diff);
    abs_heading0 = adjust_angle(abs_heading0+cor_ang_diff);

    % Reset relative values.
    x = 0;
    y = corridor_y;
    heading = adjust_angle(heading-cor_ang_diff);

end;
```

B.12 sensor_fusion.m

```
function [x,y,heading] = sensor_fusion(new_instruction,x,y,heading,ir_y,ir_heading,ir_weighting)

% Chris Lee-Johnson
%
% Function to obtain the offset and heading from the readings
% of the IR rangefinders.
%
% x:                Distance along axis parallel to corridor (m)
% y:                Offset from corridor centre axis (m)
% heading:          Heading relative to corridor axis (rad)
% ir_y:             Offset measured by IR rangefinders (m)
%                   [left,right (facing corridor angle)]
% ir_heading:       Heading measured by IR rangefinders (rad)
%                   [left,right (facing corridor angle)]
% ir_weighting:     Weightings for IR rangefinders (0:1)
%                   [left,right (facing corridor angle)]

IR_Y_TOL = 0;
IR_HEAD_TOL = 0;

% During initialisation IR weightings are increased.
if new_instruction >= 2
    ir_y_factor = 0.500;
    ir_head_factor = 0.500;
else
    ir_y_factor = 0.020;
    ir_head_factor = 0.010;
end;

% Weighted average of valid IR offsets.
ir_y_valid = [abs(ir_y(1)) < Inf, abs(ir_y(2)) < Inf];
if ir_y_valid(1) & ir_y_valid(2)
    if ir_weighting(1) == 0 & ir_weighting(2) == 0
        ir_y_av = 0.5*(ir_y(1)+ir_y(2));
    else
        ir_y_av = (ir_y(1) * ir_weighting(1) + ir_y(2) * ir_weighting(2)) ...
            / (ir_weighting(1) + ir_weighting(2));
    end;
elseif ir_y_valid(1)
    ir_y_av = ir_y(1);
elseif ir_y_valid(2)
    ir_y_av = ir_y(2);
else
    ir_y_av = y;
end;

% Weighted average of valid IR headings.
ir_head_valid...
    = [abs(ir_heading(1)) < Inf, abs(ir_heading(2)) < Inf];
if ir_head_valid(1) & ir_head_valid(2)
    if ir_weighting(1) == 0 & ir_weighting(2) == 0
        ir_head_av = 0.5*(ir_heading(1)+ir_heading(2));
    else
        ir_head_av = (ir_heading(1) * ir_weighting(1) + ir_heading(2) * ir_weighting(2)) ...
            / (ir_weighting(1) + ir_weighting(2));
    end;
elseif ir_head_valid(1)
    ir_head_av = ir_heading(1);
elseif ir_head_valid(2)
    ir_head_av = ir_heading(2);
else
    ir_head_av = y;
end;

for i = 1:2

    % If valid IR offset is sufficiently different from encoder
    % offset, include it in weighted average.
    if abs(ir_y_av-y) > IR_Y_TOL & ir_y_valid(i)
        ir_y_weight(i) = ir_y_factor * ir_weighting(i);
    else
        ir_y_weight(i) = 0;
        ir_y(i) = 0;
    end;

    % If valid IR heading is sufficiently different from encoder
    % heading, include it in weighted average.
    if abs(ir_head_av-heading) > IR_HEAD_TOL & ir_head_valid(i)
        ir_head_weight(i) = ir_head_factor * ir_weighting(i);
    else
        ir_head_weight(i) = 0;
        ir_heading(i) = 0;
    end;

end;

% Apply weighted averages.
y = (1 - ir_y_weight(1) - ir_y_weight(2)) * y + ir_y_weight(1) * ir_y(1) + ir_y_weight(2) * ir_y(2);
heading = (1 - ir_head_weight(1) - ir_head_weight(2)) * heading + ir_head_weight(1) * ir_heading(1) ...
    + ir_head_weight(2) * ir_heading(2);
```

B.13 gen_tgt_trj.m

```
function [tgt_trj_x,tgt_trj_y,tgt_trj_heading] = gen_tgt_trj(target_distance,target_angle, ...
    offset_angle,x0,y0,heading0,w_tgt_pos,wall_y)

% Chris Lee-Johnson
%
% Function to plot the target trajectory for a given movement
% instruction.
%
% tgt_trj_x:       Array of x coordinates on target trajectory (m)
% tgt_trj_y:       Array of y coordinates on target trajectory (m)
% tgt_trj_heading: Array of headings on target trajectory (rad)
% target_distance: Distance between initial and target positions (m)
% target_angle:    Target angle to turn through (rad)
% offset_angle:    Offset of target angle (rad)
% x0:              Initial distance (m)
% y0:              Initial offset (m)
% heading0:        Initial heading (rad)
% wall_y:          Left and right wall offsets (m)

% Minimum allowable distance from corridor walls.
WALL_DIST_THRESH = 0.7;

% Number of points on target trajectory.
N = 100;
i = [1:N+1];

% Apply offset to initial heading.
tgt_trj_heading0 = adjust_angle(heading0+offset_angle);

% Get target coordinates and heading.
if cos(abs(target_angle)) == 1

    % Heading for each point along target line.
    tgt_trj_heading(i) = tgt_trj_heading0;

    % Cartesian coordinates for each point along target line.
    tgt_trj_x = ((i-1)/N) * target_distance * cos(tgt_trj_heading0) + x0;
    tgt_trj_y = ((i-1)/N) * target_distance * sin(tgt_trj_heading0) + y0;

else

    % Radius of MARVIN's circular trajectory.
    radius =  abs(target_distance) / sqrt(2*(1-cos(target_angle)));

    % Heading for each point along target curve.
    [tgt_trj_heading] = adjust_angle( tgt_trj_heading0 ...
        + ((i-1)/N) * target_angle );

    % Cartesian coordinates for each point along target curve.
    if (target_distance >= 0 & target_angle > 0) | (target_distance < 0 & target_angle < 0)
        tgt_trj_x = radius * ( sin(tgt_trj_heading) + sin(-tgt_trj_heading0) ) + x0;
        tgt_trj_y = radius * ( cos(-tgt_trj_heading0) - cos(tgt_trj_heading) ) + y0;
    else
        tgt_trj_x = -radius * ( sin(tgt_trj_heading) + sin(-tgt_trj_heading0) ) + x0;
        tgt_trj_y = -radius * ( cos(-tgt_trj_heading0) - cos(tgt_trj_heading) ) + y0;
    end;

end;

% If target trajectory passes too close to corridor walls,
% follow trajectory parallel to walls.
max_y(1) = wall_y(1)+WALL_DIST_THRESH;
max_y(2) = wall_y(2)-WALL_DIST_THRESH;
if abs(target_distance) > 0.005
    for j = i
        if (tgt_trj_y(j) < max_y(1) | tgt_trj_y(j) > max_y(2)) & ((w_tgt_pos(1) > 0 & w_tgt_pos(2) > 0) ...
            | (w_tgt_pos(1) < 0 & w_tgt_pos(2) < 0))
            if tgt_trj_heading(j) <= pi/2 & tgt_trj_heading(j) > -pi/2
                tgt_trj_heading(j) = 0;
            else
                tgt_trj_heading(j) = pi;
            end;
            if tgt_trj_y(j) < max_y(1)
                tgt_trj_y(j) = max_y(1);
            else
                tgt_trj_y(j) = max_y(2);
            end;
        end;
    end;
end;
```

B.14 wheel_pos.m

```
function [w_tgt_pos] = wheel_pos(target_distance,target_angle,W_SEPARATION)

% Chris Lee-Johnson
%
% Function to obtain the target position of each wheel from
% MARVIN's target distance and angle.
%
% w_position:      Wheel positions (m)
% target_distance: Distance between origin & destination (m)
% target_angle:    Target angle to turn through (-pi:pi rad)
% W_SEPARATION:    Distance between MARVIN's wheels (m)

% Obtain arclength from input distance and angle.
if cos(target_angle) ~= 1
    if target_distance < 0
        arclength = -abs( target_angle * target_distance / sqrt(2*(1-cos(target_angle))) );
    else
        arclength = abs( target_angle * target_distance / sqrt(2*(1-cos(target_angle))) );
    end;
else
    arclength = target_distance;
end;

% Target wheel positions.
w_tgt_pos(1) = arclength + 0.5 * W_SEPARATION * target_angle;
w_tgt_pos(2) = arclength - 0.5 * W_SEPARATION * target_angle;
```

B.15 gen_vel_prof.m

```
function [w_pos_prof,w_vel_prof,w_dir,velocity_limit] = gen_vel_prof(target_distance,target_angle,w_dir, ...
    w_tgt_pos,w_velocity,velocity_limit)

% Chris Lee-Johnson
%
% Function to obtain wheel velocity profiles from MARVIN's
% target position and angle.
%
% w_pos_prof:      Velocity profile position parameters (m)
% w_vel_prof:      Velocity profile velocity parameters (m/s)
% w_dir:           Wheel direction flags (0:reverse, 1:foward)
% velocity_limit:  Overall velocity limiter (m/s)
% target_distance: Distance instruction (m)
% target_angle:    Angle instruction (rad)
% w_tgt_pos:       Wheel target positions (m)
% w_velocity:      Wheel velocities (m/s)

% Number of points per acceleration/deceleration section.
N = 100;

% Acceleration and deceleration of fastest wheel (m/s^2).
ACCELERATION = 0.15;
DECELERATION = -0.15;

% Maximum allowable velocity.
if abs(target_angle) > 0
    if abs(target_distance) < 0.5
        velocity_limit = 0.1;
    else
        velocity_limit = 0.4;
    end;
else
    velocity_limit = 0.4;
end;

% Flag to prevent function from replotting the velocity profile
% for the 2nd wheel if an illegal instruction is given to the 1st.
stop_flag = 0;

for i = 1:2

    if stop_flag == 0

        % If i = 1, j = 2 and vice versa.
        j = mod(i,2)+1;

        % Slower wheel's acceleration is proportional to ratio of
        % target positions.
        if abs(w_tgt_pos(i)) >= abs(w_tgt_pos(j))
            tgt_pos_ratio = 1;
        else
            tgt_pos_ratio = abs(w_tgt_pos(i) / w_tgt_pos(j));
        end;
        accel = tgt_pos_ratio * ACCELERATION;
        decel = tgt_pos_ratio * DECELERATION;
```

```
        max_velocity = tgt_pos_ratio * velocity_limit;
        pos_overshoot = 0;

        % Get initial velocity and final position.
        % If target position is negative, invert everything.
        if w_tgt_pos(i) > 0 | (w_tgt_pos(i) == 0 & w_dir(i) == 1)
            end_pos = w_tgt_pos(i);
            start_vel = w_velocity(i);
        else
            end_pos = -w_tgt_pos(i);
            start_vel = -w_velocity(i);
        end;
        if end_pos >= pos_overshoot
            end_pos = end_pos - pos_overshoot;
        else
            end_pos = 0;
        end;

        % Limit initial velocity.
        pseudo_max_vel = sqrt(abs(2*decel*end_pos));
        if start_vel > pseudo_max_vel
            start_vel = pseudo_max_vel;
        end;
        if start_vel > max_velocity
            start_vel = max_velocity;
        end;

        % If new motion instruction oppposes current wheel direction,
        % stop both wheels.
        if start_vel < 0
            clear w_pos_prof w_vel_prof;
            w_pos_prof(1:2,1:2) = 0;
            w_vel_prof(1:2,1:2) = 0;
            stop_flag = 1;
        else

            % Point of intersection between acceleration and
            % deceleration sections.
            mid_pos = (start_vel^2 + 2*decel*end_pos) / (2*(decel-accel));
            mid_vel = sqrt(start_vel^2 + 2*accel*mid_pos);

            % If point of intersection is less than velocity limit,
            % velocity profile will be roughly triangular.
            if mid_vel <= max_velocity

                if mid_pos == 0
                    w_pos_prof(i,1:N+1) = 0;
                else
                    w_pos_prof(i,1:N+1) = [0:mid_pos/N:mid_pos];
                end;
                w_vel_prof(i,1:N+1) = sqrt(start_vel^2 + 2*accel*w_pos_prof(i,1:N+1));

                if mid_pos-end_pos == 0
                    w_pos_prof(i,N+2:2*N+2) = 0;
                else
                    w_pos_prof(i,N+2:2*N+2) = [mid_pos:(end_pos-mid_pos)/N:end_pos];
                end;
                w_vel_prof(i,N+2:2*N+2) = sqrt(mid_vel^2+2*decel*(w_pos_prof(i,N+2:2*N+2)-mid_pos));

            % If point of intersection exceeds velocity limit,
            % velocity profile will be roughly trapezoidal.
            else

                accel_pos = (max_velocity^2 - start_vel^2) / (2*accel);
                decel_pos = end_pos + (max_velocity^2) / (2*decel);

                if accel_pos == 0
                    w_pos_prof(i,1:N+1) = 0;
                else
                    w_pos_prof(i,1:N+1) = [0:accel_pos/N:accel_pos];
                end;
                w_vel_prof(i,1:N+1) = sqrt(start_vel^2 + 2*accel*w_pos_prof(i,1:N+1));

                if end_pos-decel_pos == 0
                    w_pos_prof(i,N+2:2*N+2) = 0;
                else
                    w_pos_prof(i,N+2:2*N+2) = [decel_pos:(end_pos-decel_pos)/N:end_pos];
                end;
                w_vel_prof(i,N+2:2*N+2) = sqrt(max_velocity^2 + 2*decel*(w_pos_prof(i,N+2:2*N+2)-decel_pos));

            end;
        end;

        % If target position is negative, invert everything back again.
        if w_tgt_pos(i) < 0 | (w_tgt_pos(i) == 0 & w_dir(i) ~= 1)
            w_pos_prof(i,:) = -w_pos_prof(i,:);
            w_vel_prof(i,:) = -w_vel_prof(i,:);
        end;

    end;

end;

% If target wheel position is nonzero, update wheel direction flag.
for i = 1:2
    M = length(w_pos_prof(i,:));
    if w_pos_prof(i,M) > 0
        w_dir(i) = 1;
    elseif w_pos_prof(i,M) < 0
        w_dir(i) = 0;
    end;
end;
```

B.16 tgt_velocity.m

```
function [w_tgt_vel,w_section] = tgt_velocity(w_section, ...
    proportion,w_tgt_pos,w_pos_prof,w_vel_prof,w_vel_filt,period)

% Chris Lee-Johnson
%
% Function to obtain the target velocity of the wheels for the
% current program cycle, given its current position and velocity
% profile.
%
% w_tgt_vel:    Target wheel velocities (m/s)
% w_section:    Velocity profile sections
% proportion:   Proportion of the trajectory covered so far
% w_tgt_pos:    Target wheel positions (m)
% w_pos_prof:   Velocity profile position parameters (m)
% w_vel_prof:   Velocity profile velocity parameters (m/s)
% w_vel_filt:   Filtered wheel velocities (m/s)
% period:       Control cycle period (s)

% Velocity thresholds (m/s).
RISING_VEL_THRESH = 0.05;
FALLING_VEL_THRESH = 0.03;

for i = 1:2

    % Number of velocity profile array elements.
    N = length(w_pos_prof(i,:));

    % New wheel position.
    position = proportion * w_tgt_pos(i) + w_vel_filt(i) * period;
    if w_tgt_pos(i) >= 0
        if position > w_tgt_pos(i)
            position = w_tgt_pos(i);
        elseif position < 0
            position = 0;
        end;
    else
        if position < w_tgt_pos(i)
            position = w_tgt_pos(i);
        elseif position > 0
            position = 0;
        end;
    end;

    % If wheel has finished instruction or an emergency stop command
    % has been given, stop wheel.
    if w_section(i) >= N | w_section(i) < 0

        w_tgt_vel(i) = 0;

    else

        % Record the section that contains the current position value.
        % If position value is borderline, choose the highest section.
        w_section(i) = 0;
        for j = [1:N-1]
            if (position <= w_pos_prof(i,j+1) & position >= w_pos_prof(i,j)) ...
                    | (position >= w_pos_prof(i,j+1) & position <= w_pos_prof(i,j))
                % Record current section.
                w_section(i) = j;
            end;
        end;

        % If current position value is out of bounds of velocity profile,
        % assume error has occurred.
        if w_section(i) == 0

            if (w_pos_prof(i,N) >= 0 & position > w_pos_prof(i,N)) ...
                    | (w_pos_prof(i,N) < 0 & position < w_pos_prof(i,N))
                w_section(i) = N;
            end;
            w_tgt_vel(i) = 0;

        else

            % Calculate wheel's speed for current section.
            if w_pos_prof(i,w_section(i)+1) == w_pos_prof(i,w_section(i))
                w_tgt_vel(i) = w_vel_prof(i,w_section(i));
            else
                w_tgt_vel(i) = (position - w_pos_prof(i,w_section(i))) * (w_vel_prof(i,w_section(i)+1) ...
                    - w_vel_prof(i,w_section(i))) / (w_pos_prof(i,w_section(i)+1) ...
                    - w_pos_prof(i,w_section(i))) + w_vel_prof(i,w_section(i));
            end;

            if proportion >= 1
                w_section(i) = N;
                w_tgt_vel(i) = 0;
            else
                % Set minimum velocity during acceleration (to start wheels moving)
                % and deceleration (to prevent wheels from stopping prematurely).
                if w_vel_prof(i,w_section(i)+1) >= w_vel_prof(i,w_section(i)) ...
                        & w_tgt_vel(i) < RISING_VEL_THRESH & w_tgt_pos(i) > 0
                    w_tgt_vel(i) = RISING_VEL_THRESH;
                elseif w_vel_prof(i,w_section(i)+1) <= w_vel_prof(i,w_section(i)) ...
                        & w_tgt_vel(i) > -RISING_VEL_THRESH & w_tgt_pos(i) < 0
                    w_tgt_vel(i) = -RISING_VEL_THRESH;
```

```
                        elseif w_vel_prof(i,w_section(i)+1) < w_vel_prof(i,w_section(i)) ...
                               & w_tgt_vel(i) < FALLING_VEL_THRESH & w_tgt_pos(i) > 0
                            w_tgt_vel(i) = FALLING_VEL_THRESH;
                        elseif w_vel_prof(i,w_section(i)+1) > w_vel_prof(i,w_section(i)) ...
                               & w_tgt_vel(i) > -FALLING_VEL_THRESH & w_tgt_pos(i) < 0
                            w_tgt_vel(i) = -FALLING_VEL_THRESH;
                        end;
                    end;

            end;

        end;

end;
```

B.17 heading_error.m

```
function [head_error,proportion,tgt_x,tgt_y,tgt_heading] = heading_error(head_error,target_distance, ...
    x,y,heading,tgt_trj_x,tgt_trj_y,tgt_trj_heading,ir_obj_distance,w_tgt_pos)

% Chris Lee-Johnson
%
% Determines a heading error dependant on the difference between
% MARVIN's position & orientation and the target trajectory.
%
% head_error:       Array of heading errors (rad)
% proportion:       Proportion of the trajectory covered so far
% tgt_x:            Distance on target trajectory (m)
% tgt_y:            Offset on target trajectory (m)
% tgt_heading:      Heading on target trajectory (rad)
% target_distance:  Distance between origin & destination (m)
% x:                Distance along axis parallel to corridor (m)
% y:                Offset distance (m)
% heading:          Heading angle (rad)
% tgt_trj_x:        Array of distances n target trajectory (m)
% tgt_trj_y:        Array of offsets on target trajectory (m)
% tgt_trj_heading:  Array of headings on target trajectory (rad)
% ir_obj_distance:  Array of distances measured by rangefinders (m)
%                   [left back, left front, front,
%                    right front, right back, back]
% w_tgt_pos:        Target wheel positions (m)

% Constants.
POS_ERROR_LIMIT = 0.5;
SLOW_STOP_DIST = 0.8;
SAFE_MARG_DIST = 0.1;

% Number of points in target trajectory.
N = length(tgt_trj_x)-1;

% Get indeces of the two points on the target curve closest to
% current coordinates.
if (w_tgt_pos(1) > 0 & w_tgt_pos(2) < 0) | (w_tgt_pos(1) < 0 & w_tgt_pos(2) > 0)
    heading_separation = abs(adjust_angle(tgt_trj_heading-heading));
    [temp(1),tgt_trj_index(1)] = min(heading_separation);
    heading_separation(tgt_trj_index(1)) = Inf;
    [temp(2),tgt_trj_index(2)] = min(heading_separation);
    heading_separation(tgt_trj_index(1)) = temp(1);
else
    position_separation = sqrt( (tgt_trj_x-x).^2 + (tgt_trj_y-y).^2 );
    [temp(1),tgt_trj_index(1)] = min(position_separation);
    position_separation(tgt_trj_index(1)) = Inf;
    [temp(2),tgt_trj_index(2)] = min(position_separation);
    position_separation(tgt_trj_index(1)) = temp(1);
end;

% Get the target point.
x1 = tgt_trj_x(tgt_trj_index(1));
y1 = tgt_trj_y(tgt_trj_index(1));
x2 = tgt_trj_x(tgt_trj_index(2));
y2 = tgt_trj_y(tgt_trj_index(2));
if (x2-x1) == 0
    m1 = Inf;
else
    m1 = (y2-y1) / (x2-x1);
end;
if (y2-y1) == 0
    m2 = Inf;
else
    m2 = (x1-x2) / (y2-y1);
end;
if m1 == Inf
    tgt_x = x1;
    if m2 == Inf
        tgt_y = y1;
    else
        tgt_y = y;
    end;
elseif m2 == Inf
    tgt_x = x;
    tgt_y = y1;
```

```
else
    tgt_x = ( m1*x1 - m2*x + y - y1 ) / ( m1 - m2 );
    tgt_y = ( m1*y - m2*y1 + x - x1 ) / ( m1 - m2 );
end;

% Confine point to target curve for angle/proportion
% calculations.
if (tgt_x < x1 & tgt_x < x2 & x1 <= x2) | (tgt_x > x1 & tgt_x > x2 & x1 >= x2) ...
        | (tgt_y < y1 & tgt_y < y2 & y1 <= y2) | (tgt_y > y1 & tgt_y > y2 & y1 >= y2)
    p_x = x1;
    p_y = y1;
elseif (tgt_x < x1 & tgt_x < x2 & x1 > x2) | (tgt_x > x1 & tgt_x > x2 & x1 < x2) ...
        | (tgt_y < y1 & tgt_y < y2 & y1 > y2) | (tgt_y > y1 & tgt_y > y2 & y1 < y2)
    p_x = x2;
    p_y = y2;
else
    p_x = tgt_x;
    p_y = tgt_y;
end;

% Get target angle.
a1 = tgt_trj_heading(tgt_trj_index(1));
a2 = tgt_trj_heading(tgt_trj_index(2));
if (w_tgt_pos(1) > 0 & w_tgt_pos(2) < 0) | (w_tgt_pos(1) < 0 & w_tgt_pos(2) > 0)
    a = is_inside_arc(tgt_trj_heading(1),tgt_trj_heading(round(N/2)),tgt_trj_heading(N+1),heading);
    if a == 0
        p1 = abs(adjust_angle(heading-a2));
        p2 = abs(adjust_angle(heading-a1));
    elseif a == 1
        p1 = 0;
        p2 = 1;
    else
        p1 = 1;
        p2 = 0;
    end;
else
    p1 = sqrt((p_x-x1)^2+(p_y-y1)^2);
    p2 = sqrt((p_x-x2)^2+(p_y-y2)^2);
end;
if p1+p2 == 0
    p_ratio = 0;
else
    p_ratio = p1/(p1+p2);
end;
tgt_heading = average_angle(a1,a2,p_ratio);

% If travelling in reverse, fold target heading over actual
% heading axis.
fwd_tgt_heading = tgt_heading;
if target_distance < 0
    tgt_heading = adjust_angle(heading-adjust_angle(tgt_heading-heading));
end;

% Get angle of line linking actual and target positions.
x_diff = tgt_x-x;
y_diff = tgt_y-y;
if x_diff == 0
    if y_diff >= 0
        separation_angle = pi/2;
    else
        separation_angle = -pi/2;
    end;
else
    separation_angle = atan2(y_diff,x_diff);
end;

% Difference seperation angle and real heading (sign indicates
% left or right turn to correct).
sep_ang_error = adjust_angle(separation_angle-heading);

% Difference between actual and target position.
if sep_ang_error >= 0
    position_error = sqrt( x_diff^2 + y_diff^2 );
else
    position_error = -sqrt( x_diff^2 + y_diff^2 );
end;

% Derive weighting from position error.
if abs(position_error) >= POS_ERROR_LIMIT
    weight = 1;
else
    weight = abs(position_error) / POS_ERROR_LIMIT;
end;

% Weighted average of separation angle and target heading.
avg_angle = average_angle(separation_angle,tgt_heading,weight);

% Heading error is the difference between average angle and
% real heading.
head_error(3) = head_error(2);
head_error(2) = head_error(1);
head_error(1) = real(adjust_angle(avg_angle-heading));

% Get proportion of trajectory that MARVIN has covered so far.
if min([tgt_trj_index(1),tgt_trj_index(2)]) == tgt_trj_index(1)
    proportion = (tgt_trj_index(1) + p_ratio - 1) / N;
else
    proportion = (tgt_trj_index(2) - p_ratio) / N;
end;

% If MARVIN is too far from target position or heading, or if
% object is blocking intended trajectory, slow wheels to a halt
```

```
% (as opposed to brake or rapid stop seen in stop_wheels function).
ex_distance = (1-proportion) * target_distance;
if position_error > 0.75 | abs(adjust_angle(tgt_heading-heading)) > 0.5*pi ...
        | (ir_obj_distance(3) <= SLOW_STOP_DIST & ex_distance > 0 ...
        & ir_obj_distance(3) <= ex_distance + SAFE_MARG_DIST) ...
        | (ir_obj_distance(6) <= SLOW_STOP_DIST & ex_distance < 0 ...
        & ir_obj_distance(6) <= -ex_distance + SAFE_MARG_DIST)
    proportion = 1;
end;
```

B.18 heading_control.m

```
function [w_vel_error,w_vel_error_filt] = heading_control(new_instruction, ...
    period,w_tgt_vel,w_velocity,w_vel_filt,head_error,time_diff)

% Chris Lee-Johnson
%
% Function to derive PID control errors from wheel velocities
% and heading error.
%
% w_vel_error:      PID velocity errors (m/s)
%                   (2x3 array)
% w_vel_error_filt: Filtered PID velocity errors (m/s)
%                   (3x2 array)
% new_instruction:  New instruction flag
% period:           Control cycle period (s)
% w_tgt_vel:        Target wheel velocities (m/s)
% w_velocity:       Actual wheel velocities (m/s)
% w_vel_filt:       Filtered wheel velocities (m/s)
% head_error:       Array of heading errors (rad)

% Persistent variables
persistent w_last_vel;
persistent w_vel_e;
persistent w_vel_e_f;
persistent modifier;

% Acceleration limits.
TGT_ACCEL_LIMIT = 0.3;
TGT_DECEL_LIMIT = -0.3;
RE_ACCEL_LIMIT = 0.5;
RE_DECEL_LIMIT = -0.5;

% Integral time.
HEAD_TI = 10;

% Derivative time.
HEAD_TD = 0.05;

% Proportional gain.
MIN_AVG_TGT_VEL = 0.4;
MAX_AVG_TGT_VEL = 2.0;
MIN_HEAD_K = 1.0/pi;
MAX_HEAD_K = 5.0/pi;
avg_tgt_vel = mean(w_tgt_vel);
if avg_tgt_vel <= MIN_AVG_TGT_VEL
    head_k = MAX_HEAD_K;
elseif avg_tgt_vel >= MAX_AVG_TGT_VEL
    head_k = MIN_HEAD_K;
else
    head_k = MIN_HEAD_K + ((MAX_HEAD_K-MIN_HEAD_K) / (MAX_AVG_TGT_VEL-MIN_AVG_TGT_VEL)) ...
        * (MAX_AVG_TGT_VEL-avg_tgt_vel);
end;

% If first instruction, initialise variables.
if new_instruction == 2
    w_last_vel = w_tgt_vel;
    modifier = 0;
    w_vel_e(1:2,1:3) = 0;
    w_vel_e_f(1:2,1:3) = 0;
end;

% PID control to derive modifier due to heading error.
if period > 0
    modifier = modifier + head_k * ( (1+period/HEAD_TI+HEAD_TD/period) * head_error(1) ...
        - (1+2*HEAD_TD/period) * head_error(2) + HEAD_TD/period * head_error(3) );
else
    modifier = 0;
end;

% Impose limits on modifier.
if modifier > 1
    modifier = 1;
elseif modifier < -1
    modifier = -1;
end;

% Limit maximum wheel velocity based on modifier value.
dyn_vel_lim = 2-1.8*abs(modifier);
for i = 1:2
    if w_tgt_vel(i) > dyn_vel_lim
```

```
        w_tgt_vel(i) = dyn_vel_lim;
    elseif w_tgt_vel(i) < -dyn_vel_lim
        w_tgt_vel(i) = -dyn_vel_lim;
    end;
end;

% Apply modifier to target velocity.
if (w_tgt_vel(1) > 0 & w_tgt_vel(2) > 0) ...
     | (w_tgt_vel(1) < 0 & w_tgt_vel(2) < 0)
    if modifier >= 0
        w_tgt_vel = [w_tgt_vel(1)*(1+0*modifier),w_tgt_vel(2)*(1-modifier)];
    else
        w_tgt_vel = [w_tgt_vel(1)*(1+modifier),w_tgt_vel(2)*(1-0*modifier)];
    end;
end;

% Limit wheel acceleration/deceleration.
for i = 1:2
    if w_tgt_vel(i) > w_last_vel(i) + TGT_ACCEL_LIMIT * period
        w_tgt_vel(i) = w_last_vel(i) + TGT_ACCEL_LIMIT * period;
    elseif w_tgt_vel(i) < w_last_vel(i) + TGT_DECEL_LIMIT * period
        w_tgt_vel(i) = w_last_vel(i) + TGT_DECEL_LIMIT * period;
    end;
    if w_tgt_vel(i) > w_velocity(i) + RE_ACCEL_LIMIT * period
        w_tgt_vel(i) = w_velocity(i) + RE_ACCEL_LIMIT * period;
    elseif w_tgt_vel(i) < w_velocity(i) + RE_DECEL_LIMIT * period
        w_tgt_vel(i) = w_velocity(i) + RE_DECEL_LIMIT * period;
    end;
end;

% Calculate wheel velocity errors
w_vel_e(:,3) = w_vel_e(:,2);
w_vel_e(:,2) = w_vel_e(:,1);
w_vel_e(:,1) = w_tgt_vel' - w_velocity';
w_vel_e_f(:,3) = w_vel_e_f(:,2);
w_vel_e_f(:,2) = w_vel_e_f(:,1);
w_vel_e_f(:,1) = w_tgt_vel' - w_vel_filt';
w_vel_error = w_vel_e;
w_vel_error_filt = w_vel_e_f;

% Update last target velocity.
w_last_vel = w_tgt_vel;
```

B.19 average_angle.m

```
function [angle] = average_angle(a1,a2,weight)

% Chris Lee-Johnson
%
% Function to calculate the weighted average of two angles
% within the range (-pi:pi].
%
% angle:   Weighted average angle
% a1:      First angle
% a2:      Second angle
% weight:  Weighting for 1st angle
%          (2nd angle wighting = 1-weight)

if (a1-a2) > pi
    angle = adjust_angle(weight*a1+(1-weight)*(a2+2*pi));
elseif (a1-a2) < -pi
    angle = adjust_angle(weight*a1+(1-weight)*(a2-2*pi));
else
    angle = adjust_angle(weight*a1+(1-weight)*a2);
end;
```

B.20 velocity_control.m

```
function [w_tgt_vel] = velocity_control(new_instruction,w_tgt_vel,w_vel_error, ...
    w_vel_error_filt,w_velocity,w_dir,period,velocity_limit)

% Chris Lee-Johnson
%
% Function to apply PID control to the target wheel velocity.
%
% w_tgt_vel:        Target wheel velocities (m/s)
% new_instruction:  New instruction flag
% w_vel_error:      PID velocity errors (m/s)
%                   (3x2 array)
% w_vel_error_filt: Filtered PID velocity errors (m/s)
%                   (3x2 array)
% w_velocity:       Actual wheel velocities (m/s)
% w_dir:            Wheel direction flags (0:reverse, 1:foward)
% period:           Control cycle period (s)

persistent w_last_vel;

% Real acceleration limits.
VEL_DIFF_LIMIT = 0.4;

% Control constants.
W_VEL_K = [1.0,1.0];    % Proportional gain
W_VEL_TI = [0.2,0.2];   % Integral time
W_VEL_TD = [0.01,0.01]; % Derivative time

% If first instruction, initialise variables.
if new_instruction == 2
    w_last_vel = w_tgt_vel;
end;

% PID control.
if period > 0
    prop = w_vel_error(:,1)' - w_vel_error(:,2)';
    integ = (period ./ W_VEL_TI) .* w_vel_error(:,1)';
    deriv = (W_VEL_TD/period) .* w_vel_error_filt(:,1)' - (2*W_VEL_TD/period) .* w_vel_error_filt(:,2)' ...
        + W_VEL_TD/period .* w_vel_error_filt(:,3)';
    w_new_tgt_vel = w_last_vel + W_VEL_K .* (prop + integ + deriv);
else
    w_new_tgt_vel = w_velocity;
end;

for i = 1:2

    % Do not reverse direction while wheel is moving.
    if (w_tgt_vel(i) >= 0 & w_new_tgt_vel(i) < 0) | (w_tgt_vel(i) <= 0 & w_new_tgt_vel(i) > 0) ...
            | (w_velocity(i) > 0 & w_new_tgt_vel(i) < 0) | (w_velocity(i) < 0 & w_new_tgt_vel(i) > 0) ...
            | (w_dir(i) == 1 & w_new_tgt_vel(i) < 0) | (w_dir(i) == 0 & w_new_tgt_vel(i) > 0)
        w_new_tgt_vel(i) = 0;
    end;

    % Do not exceed maximum safe velocity.
    if w_new_tgt_vel(i) > velocity_limit+0.2
        w_new_tgt_vel(i) = velocity_limit+0.2;
    elseif w_new_tgt_vel(i) < -velocity_limit-0.2
        w_new_tgt_vel(i) = -velocity_limit-0.2;
    end;

    % Limit wheel acceleration/deceleration.
    if w_new_tgt_vel(i) > w_velocity(i) + VEL_DIFF_LIMIT
        w_new_tgt_vel(i) = w_velocity(i) + VEL_DIFF_LIMIT;
    elseif w_new_tgt_vel(i) < w_velocity(i) - VEL_DIFF_LIMIT
        w_new_tgt_vel(i) = w_velocity(i) - VEL_DIFF_LIMIT;
    end;

end;

% Update target velocities.
w_tgt_vel = w_new_tgt_vel;
w_last_vel = w_tgt_vel;
```

B.21 stop_wheels.m

```
function [brake,w_tgt_vel,w_section] = stop_wheel(brake,w_tgt_vel,w_section,contact_switch, ...
    ir_obj_distance,target_distance,target_angle)

% Chris Lee-Johnson
%
% Function to stop wheels immediately in the event of an impending
% collision or a stop instruction.
%
% brake:           Brake flag
% w_tgt_vel:       Target wheel velocities (m/s)
% w_section:       Velocity profile sections
% contact_switch:  Array of switch inputs
% ir_obj_distance: Array of distances measured by rangefinders (m)
% target_distance: Distance between origin & destination (m)
% target_angle:    Target angle to turn through (-pi:pi rad)

% Fast stopping distance.
FAST_STOP_DIST = 0.4;

% Brake wheels.
if contact_switch(1) ~= 0 | contact_switch(2) ~= 0 | contact_switch(3) ~= 0 | contact_switch(4) ~= 0
    w_tgt_vel = [0,0];
    w_section = [-1,-1];
    brake = 1;
% Stop wheels.
elseif (target_distance == 0 & target_angle == 0) | w_section(1) == -1 | w_section(2) == -1 ...
        | (ir_obj_distance(3) <= FAST_STOP_DIST & target_distance > 0) ...
        | (ir_obj_distance(6) <= FAST_STOP_DIST & target_distance < 0)
    w_tgt_vel = [0,0];
    w_section = [-1,-1];
end;
```

B.22 get_motor_power.m

```
function [w_pwm] = get_motor_power(w_tgt_vel)

% Chris Lee-Johnson
%
% Function to convert target velocities into PWM values to send
% to the micro in order to drive the motors.
%
% w_pwm:       PWM setting for wheels
% w_tgt_vel:   Target wheel velocities (m/s)

% PWM - velocity relationship slopes and intercepts.
W_PWM_OVER_V_POS = [159.6455,150.9647];
W_PWM_OVER_V_NEG = [159.6455,150.9647];
W_MIN_PWM_POS = [25.1706,24.0155];
W_MIN_PWM_NEG = [-25.1706,24.0155];

% Minimum wheel velocity (m/s).
MIN_VELOCITY = 0.01;

% Convert to PWM value.
for i = 1:2
    if w_tgt_vel(i) < MIN_VELOCITY & w_tgt_vel(i) > -MIN_VELOCITY
        w_pwm(i) = 0;
    elseif w_tgt_vel(i) < 0
        w_pwm(i) = W_PWM_OVER_V_NEG(i) * w_tgt_vel(i) + W_MIN_PWM_NEG(i);
    else
        w_pwm(i) = W_PWM_OVER_V_POS(i) * w_tgt_vel(i) + W_MIN_PWM_POS(i);
    end;
    if w_pwm(i) > 255
        w_pwm(i) = 255;
    elseif w_pwm(i) < -255
        w_pwm(i) = -255;
    end;
end;
```

B.23 set_motor_power.m

```
function [error] = set_motor_power(new_instruction,brake,w_pwm,w_dir,lvserv)

% Chris Lee-Johnson
%
% Function to send pwm values etc to the microcontroller in order
% to drive the motors.
%
% error:              Array of error counts returned
% new_instruction:    New instruction flag (see marvin_control)
% brake:              Brake flag
% w_pwm:              PWM setting for wheels
% lvserv:             LabVIEW ActiveX server object

% Persistent variables.
persistent set_motor_power_vi;

% If first instruction, initialise variables and set up LABVIEW
% interface.
if new_instruction >= 2

    % Set up Wheel Controller VI.
    set_motor_power_vi = invoke(lvserv,'GetViReference','c:\Project\Code\LabVIEW\Set Motor Power.vi');

end;

% Get header bytes and PWM bytes from pwm values.
for i = 1:2
    if brake == 1
        header(i) = 0;
        pwm(i) = 0;
    elseif w_dir(i) == 1
        header(i) = 3 + 2*(i-1);
        pwm(i) = round((w_pwm(i)/2) + 128);
    else
        header(i) = 2 + 2*(i-1);
        pwm(i) = -round((w_pwm(i)/2) - 128);
    end;
end;

% Set LabVIEW control values.
invoke(set_motor_power_vi,'SetControlValue','patterns to write',num2str(header(1)));
invoke(set_motor_power_vi,'SetControlValue','patterns to write 2',num2str(pwm(1)));
invoke(set_motor_power_vi,'SetControlValue','patterns to write 3',num2str(header(2)));
invoke(set_motor_power_vi,'SetControlValue','patterns to write 4',num2str(pwm(2)));

% Run VI.
set_motor_power_vi.Run;

% Read error counts.
error(1) = double(invoke(set_motor_power_vi,'GetControlValue','error'));
error(2) = double(invoke(set_motor_power_vi,'GetControlValue','error 2'));
error(3) = double(invoke(set_motor_power_vi,'GetControlValue','error 3'));
error(4) = double(invoke(set_motor_power_vi,'GetControlValue','error 4'));
```

B.24 sim_en_count.m

```
function [w_count_diff,time_diff,time] = sim_en_count(new_instruction,w_sim_vel,COUNTS_PER_M)

% Chris Lee-Johnson
%
% Simulation function to obtain encoder counts.
%
% w_count_diff:    Wheel displacements (encoder counts)
% time_diff:       Time since last cycle (s)
% time:            Total time elapsed (s)
% new_instruction: New instruction flag
% w_sim_vel:       Simulated wheel velocities (m/s)
% COUNTS_PER_M:    Number of encoder counts in 1m

% Persistent variables
persistent old_time;
persistent first_time;

% If first instruction, initialise variables.
if new_instruction == 2

    % Initialise variables.
    first_time = cputime;
    old_time = 0;

end;

% Get elapsed time between cycles.
time = cputime-first_time;
```

```
time_diff = time-old_time;
while time_diff < 0.08
    time = cputime-first_time;
    time_diff = time-old_time;
end;
old_time = time;

% Get simulated wheel counts
ERROR = [1,1];
w_count_diff = ERROR .* w_sim_vel * time_diff .* COUNTS_PER_M;
```

B.25 sim_ir_voltage.m

```
function [ir_voltage] = sim_ir_voltage(new_instruction,x,y,heading,wall_y,IR_OGN_X,IR_OGN_Y,IR_OGN_ANGLE)

% Chris Lee-Johnson
%
% Simulation function to obtain rangefinder voltages.
%
% ir_voltage:       Array of voltages output from rangefinders (V)
%                   [left back, left front, front,
%                   right front, right back, back]
% new_instruction:  New instruction flag (see marvin_control)
% x:                Distance (m)
% y:                Offset (m)
% heading:          Heading (rad)
% wall_y:           Left and right wall offsets (m)

% Persistent variables
persistent ir_inst_voltage;

% Look-up table parameters for IR voltage-distance relationships.
ir_dist_curve = [0.15:0.05:1.5];
ir_volt_curve = [2.790,2.560,2.300,1.950,1.700,1.500,1.350,1.210,1.100, ...
                 1.010,0.935,0.865,0.805,0.750,0.700,0.665,0.625,0.595, ...
                 0.565,0.540,0.515,0.495,0.480,0.460,0.445,0.430,0.420,0.410;
                 2.750,2.500,2.235,1.900,1.650,1.460,1.305,1.175,1.070, ...
                 0.990,0.910,0.850,0.790,0.730,0.690,0.650,0.610,0.585, ...
                 0.555,0.530,0.505,0.485,0.465,0.445,0.430,0.415,0.400,0.390;
                 2.950,2.665,2.300,1.930,1.720,1.520,1.350,1.225,1.120, ...
                 1.040,0.955,0.900,0.830,0.775,0.725,0.685,0.640,0.600, ...
                 0.565,0.545,0.510,0.495,0.475,0.455,0.445,0.430,0.415,0.410;
                 2.800,2.575,2.305,1.990,1.725,1.510,1.370,1.230,1.125, ...
                 1.030,0.950,0.885,0.840,0.790,0.745,0.700,0.665,0.630, ...
                 0.600,0.575,0.550,0.530,0.510,0.495,0.480,0.465,0.455,0.450;
                 2.680,2.460,2.225,1.905,1.650,1.470,1.320,1.180,1.080, ...
                 1.000,0.925,0.860,0.810,0.760,0.715,0.670,0.630,0.595, ...
                 0.565,0.540,0.515,0.495,0.480,0.465,0.450,0.440,0.430,0.425;
                 2.760,2.500,2.240,1.900,1.655,1.475,1.325,1.200,1.085, ...
                 0.995,0.920,0.855,0.800,0.750,0.710,0.675,0.645,0.605, ...
                 0.580,0.550,0.525,0.500,0.475,0.450,0.430,0.415,0.400,0.390];

M = length(ir_dist_curve);
N = 10;

% If first instruction, initialise all voltages to first voltage.
if new_instruction == 2
    ir_inst_voltage(1:N,:) = 0.1;
end;

for i = N-1:-1:1
    ir_inst_voltage(i+1,:) = ir_inst_voltage(i,:);
end;

% Cartesian coordinates of each IR (adjusted for MARVIN's overall
% position and orientation).
[ir_adj_x,ir_adj_y,ir_adj_angle] = coord_trans(x,y,heading,IR_OGN_X,IR_OGN_Y,IR_OGN_ANGLE);

% Distance (in m) from wall.
for i = 1:6
    if sin(ir_adj_angle(i)) == 0
        d(i) = Inf;
    elseif ir_adj_angle(i) < 0
        d(i) = abs( (wall_y(1)-ir_adj_y(i)) / sin(ir_adj_angle(i)) );
    else
        d(i) = abs( (wall_y(2)-ir_adj_y(i)) / sin(ir_adj_angle(i)) );
    end;
end;

for i = 1:6

    % Record the section that contains the current distance value.
    ir_section(i) = 0;
    for j = [1:M-1]
        if (d(i) <= ir_dist_curve(j+1) & d(i) >= ir_dist_curve(j)) ...
              | (d(i) >= ir_dist_curve(j+1) & d(i) <= ir_dist_curve(j))
            % Record current section.
            ir_section(i) = j;
        end;
    end;
```

```
        % If current position value is outside curve boundaries.
        if ir_section(i) == 0

            if (d(i) < ir_dist_curve(M))
                ir_inst_voltage(1,i) = ir_volt_curve(i,M);
            else
                ir_inst_voltage(1,i) = 0.1;
            end;

        else

            % Calculate distance for current section.
            if ir_dist_curve(ir_section(i)+1) ...
                    == ir_dist_curve(ir_section(i))
                ir_inst_voltage(1,i) = ir_volt_curve(i,ir_section(i));
            else
                ir_inst_voltage(1,i) = (d(i) - ir_dist_curve(ir_section(i))) ...
                    * (ir_volt_curve(i,ir_section(i)+1) - ir_volt_curve(i,ir_section(i))) ...
                    / (ir_dist_curve(ir_section(i)+1) - ir_dist_curve(ir_section(i))) ...
                    + ir_volt_curve(i,ir_section(i));
            end;

        end;

    end;

% If first instruction, initialise all voltages to first voltage.
if new_instruction == 2
    ir_inst_voltage(2:N,1) = ir_inst_voltage(1,1);
    ir_inst_voltage(2:N,2) = ir_inst_voltage(1,2);
    ir_inst_voltage(2:N,3) = ir_inst_voltage(1,3);
    ir_inst_voltage(2:N,4) = ir_inst_voltage(1,4);
    ir_inst_voltage(2:N,5) = ir_inst_voltage(1,5);
    ir_inst_voltage(2:N,6) = ir_inst_voltage(1,6);
end;

% Software filter to reduce noise.
weight = 0.5;
ir_voltage(1:6) = 0;
for i = 1:N
    ir_voltage = ir_voltage + weight * ir_inst_voltage(i,:);
    if i < N-1
        weight = 0.5*weight;
    end;
end;
```

B.26 sim_motor_power.m

```
function [w_sim_vel] = sim_motor_power(new_instruction,w_tgt_vel)

% Chris Lee-Johnson
%
% Simulation function to apply power to motors.
%
% w_sim_vel:       Simulated wheel velocities (m/s)
% w_tgt_vel:       Target wheel velocities (m/s)
% new_instruction: New instruction flag

% Minimum wheel velocity (m/s)
MIN_VELOCITY = [0.01,0.01];

% If target speed is too low to overcome friction,
% wheel will not move.
for i = 1:2
    if abs(w_tgt_vel(i)) < MIN_VELOCITY(i) | w_tgt_vel(i) == NaN
        w_sim_vel(i) = 0;
    else
        w_sim_vel(i) = w_tgt_vel(i);
    end;
end;
```

Appendix C: CD Contents

The attached CD contains the following:

- This document
    - Microsoft Word format
    - PDF format

- Test results
    - Captured data
    - MATLAB figures
    - Video samples

- Source code
    - MATLAB functions
    - LabVIEW VIs
    - Microcontroller C code

- Datasheets
    - 6025E data acquisition card
    - HEDS-5500 optical encoder
    - GP2Y0A02YK infrared rangefinder
    - P89C51RC2HBP microcontroller

References

Borenstein, J., & Feng, L., "**Measurement and Correction of Systematic Odometry Errors in Mobile Robots**", *IEEE Transactions on Robotics and Automation*, October 1996.

Chappell, D., "**Introducing ActiveX**", David Chappell & Associates, http://www.chappellassoc.com, 1997.

Cordes, J.C., "**The Creating of an Autonomous Multi-Terrain Mechatron**", MSc Thesis, Department of Physics and Electronic Engineering, University of Waikato, 2002.

"**Dynamic Data Exchange (DDE) and NetDDE FAQ**", RHA (Minisystems) Ltd, http://www.rhaminisys.com.

Franklin, G.F., Powell, J.D., Emami-Naeini, A., "**Feedback Control of Dynamic Systems**", 4th Edition, Prentice Hall, 2002.

Franklin, G.F., Powell, J.D., Workman M., "**Digital Control of Dynamic Systems**", 3rd Edition, Addison Wesley, 1998.

Godjevac, J., "**Comparative Study of Fuzzy Control, Neural Network Control and Neuro-Fuzzy Control**", 1995.

Halici, U., "**Introduction to Neural Networks**", Chapter 1, METU Informatics Institute, Middle East Technical University, http://euclid.ii.metu.edu.tr/~ion526/demo/demochp.html.

Hurd, S.A., "**Laser Range Finding for an Autonomous Mobile Security Device**", MSc Thesis, Department of Physics and Electronic Engineering, University of Waikato, 2001.

Jenson, C.H., Carnegie, D.A., Gaynor, P., "**Universal Battery Powered Pulse Width Modulated H-Bridge Motor Control for Robotic Applications**", *Proceedings of the 10th Electronics New Zealand Conference*, Hamilton, New Zealand, September 2003.

King, J.C., "**The Development of an AUV**", MSc Thesis, Department of Physics and Electronic Engineering, University of Waikato, 2002.

Lee-Johnson, C.P., Carnegie, D.A., "**The Development of a Control System for an Autonomous Mobile Robot**", *Proceedings of the 10th Electronics New Zealand Conference*, Hamilton, New Zealand, September 2003.

Loughnane, D.J., "**Design and Construction of an Autonomous Mobile Security Device**", MSc Thesis, Department of Physics and Electronic Engineering, University of Waikato, 2001.

Payne, A.D., Carnegie, D.A., "**Design and Construction of a Pair of Tricycle Based Robots to Investigate Cooperative Robotic Interaction**", *Proceedings of the 10th Electronics New Zealand Conference*, Hamilton, New Zealand, September 2003.

Prakash, A., Carnegie, D.A., Chitty, C., "**The Humanisation of an Autonomous Mobile Robot**", *Proceedings of the 10th Electronics New Zealand Conference*, Hamilton, New Zealand, September 2003.

Sikking, L.J., Carnegie, D.A., "**The Development of an Indoor Navigation Algorithm for an Autonomous Mobile Robot**", *Proceedings of the 10th Electronics New Zealand Conference*, Hamilton, New Zealand, September 2003.

van Dam, J., Dev, A., Dorst, L., Groen, F., Hertzberger, L., van Inge, A., Krose, B., Lagerberg, J., Visser, A., Wiering, M., "**Organisation and Design of Autonomous Systems**", Chapter 9, Lecture Notes, University of Amsterdam, 1999.

van Dam, J., Kröse, B., Groen, F., "**Neural Network Applications in Sensor Fusion for an Autonomous Mobile Robot**", University of Amsterdam, 1996.

Wu, H., Seigel, M., Stiefelhagen, R., Yang, J., "**Sensor Fusion using Dempster-Shafer Theory**", *IEEE Instrumentation and Measurement Technology Conference*, Anchorage, AK, USA, May 2002.

Xue, D., "**MATLAB's External Interfacing with Others**", MATLAB Paradise, http://matlab.myrice.com, 2000.