Hybrid Control of a Segway Platform Developed in MRDS

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Master of Engineering

in Electronic and Computer Systems Engineering

at

Victoria University of Wellington

by

Douglas James Ormiston Thomson

2013

Abstract

A Segway RMP200 has been bought by Victoria University for the purpose of making an autonomous robot. The focus of this project was to create reusable services that use existing navigation algorithms to control the Segway within an indoor environment.

A SICK LMS100 laser rangefinder was added to detect obstacles and allow localization of the Segway within a known map. A hybrid navigation algorithm consisting of an A* path planner with a dynamic window is used for motion planning and obstacle avoidance.

The control system followed a Service Oriented Architecture implemented in Microsoft Robotics Studio using the C# .NET programming language.

Four services were created during the project to interface with the SICK LMS100 scanner, control the Segway RMP200, implement the hybrid navigation algorithm and provide a graphic user interface for the system.

Tests show that the Segway is able to navigate and maintain localisation within the operating environment by identifying and associating corner and door landmarks within the environment.

Acknowledgements

I would like to acknowledge the following people for their help they have given me over the course of my masters.

I would firstly like to thank Professor Dale Carnegie for supervising me during this project. His advice and expertise was invaluable during the project and for the final thesis write up.

I would also like to thank Jason Edwards and Tim Exley for their technical advice during provided during this project.

Lastly I would like to thank my family for supporting me while completing my masters.

Contents

List of Tables

List of Figures

Chapter 1 Introduction

A Segway RMP200 has been obtained by Victoria University of Wellington to be used as a platform on which to develop an autonomous robot. The Segway RMP200 platform is a two wheel differential drive system capable of dynamic stabilisation. Dynamic stabilisation is the ability to balance a payload above two wheels, similar to an inverted pendulum.

The Segway platform was purchased to extend the mobility of existing at Victoria. The current platform of the MARVIN robot is limited by its current motors and the small wheels limit the platform's operating environment (such as traversing the gap while entering certain elevators within the university). These restrictions prohibit outdoor operation. The Segway platform has greater flexibility and ability to move in an indoor and outdoor environment.

An autonomous robot can perform desired tasks in known or unknown environments without human intervention or guidance. Autonomous robots require the ability to sense and act upon information acquired while traversing an environment and to navigate while avoiding obstacles. Autonomous robots employ intelligent navigation systems that are responsible for maintaining the current position of the robot, where the robot is attempting to head and how the robot navigates to a goal.

1.1 Objective

The objective of this project is to make a Segway platform intelligently move around an indoor environment while avoiding obstacles. The operating environment will be mapped so the navigation system for the Segway can assume knowledge beforehand. The current position and destination is also known before autonomous behaviour is engaged. A map and starting position is given as this project does not attempt to solve the Simultaneous Localisation and Mapping (SLAM) problem. SLAM enables a robot to build a map of an unknown area while dynamically estimating its own pose in the growing map.

This project must consider the following:

- Selection of an appropriate development environment,

- Interfacing with the Segway RMP and control software,
- Choice of sensors to aid localization and detection of obstacles,
- Creation of a service to interface with sensors,
- Use of existing algorithms for positioning the Segway in a known environment,
- Implementing path finding and following algorithms and,
- Design of a user interface to supervise autonomous behaviour.

The software developed has been designed to be extendible and re-usable to minimize the time taken to apply the system to different robotic platforms.

Balancing algorithms for the Segway RMP and algorithms for robotic navigation are established and will be utilised for this project.

1.2 Mobile Robot Platforms

Victoria University's Mechatronic Group has several robotic platforms which have been developed by previous research projects. Two of these robots, shown in Figure 1.1 and Figure 1.2, make up part of the three tier hierarchal urban search and rescue system being developed at Victoria.

Figure 1.1 Grandmother robot

The Grandmother robot displayed in Figure 1.1 is designed to be deployed remotely and proceed autonomously into a disaster zone. The system is designed for the Grandmother robot to co-ordinate several smaller Mother robots such as displayed in Figure 1.2. The Mother robot then deploys smaller disposable Daughter robots that explore the disaster zone to find and locate surviving humans. The Grandmother robot is currently undergoing a redesign and the Daughter robots are currently being developed.

Figure 1.2 Mother robot

The control systems for these robots are not developed in conventional Robotic Development Environments, rather they have developed in Matlab for the Grandmother and in embedded software for the Mother robot. This project will help to create a standard development environment that can be used to upgrade these current systems.

Another robotic platform is a differential drive robot called MARVIN (Mobile Autonomous Robotic Vehicle for Indoor Navigation). MARVIN (seen in Figure 1.3) has been designed as an autonomous mobile security system that would patrol the corridors of the university interacting with people.

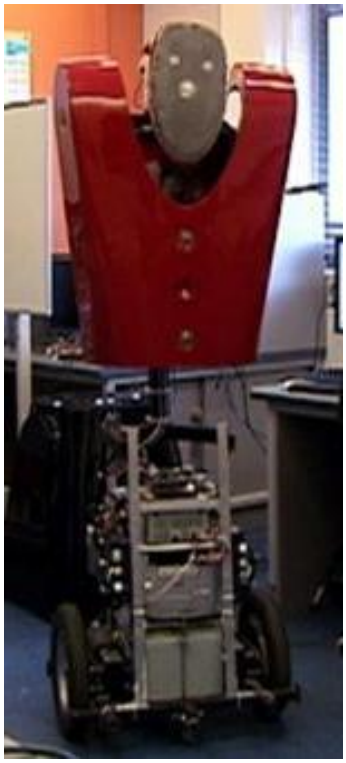

Figure 1.3 MARVIN robotic platform

1.3 Operating Environment

The Segway is intended to operate primarily in the corridors of the third floor of the Laby building at Victoria University of Wellington. This environment is used for debugging and testing the hybrid navigation system as well as the localisation algorithm. An overhead view of the floor map is given in Figure 1.4 with images of the environment given in Figure 1.5

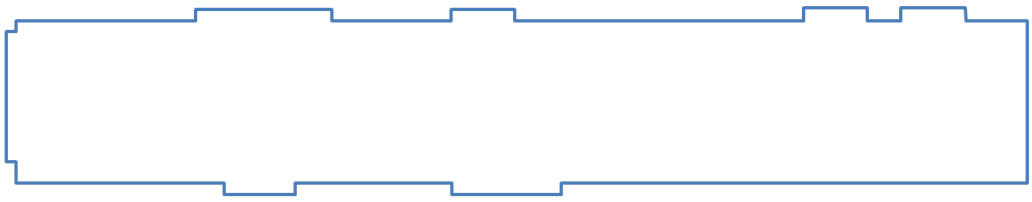

Figure 1.4 Overhead view of the operating environment

Figure 1.5 Images of the operating environment

The navigation system can be expanded to incorporate other indoor environments assuming a map of the environment has been made. The control system has been developed and tested with the expectation that the system will operate in different environments such as those shown in Figure 1.6, Figure 1.7 and Figure 1.8.

The different environments shown contain wooden and glass walls along with vinyl and carpet flooring creating a range of surfaces for sensors and wheels to operate on and which the navigation system is tested and accommodates for.

The localisation system is designed to perform in indoor corridor environments where landmarks such as corners and doors are commonly found. The control system can still operate in environments where corners and doors are sparse; however it then relies more on odometry for localisation and can succumb to cumulative errors.

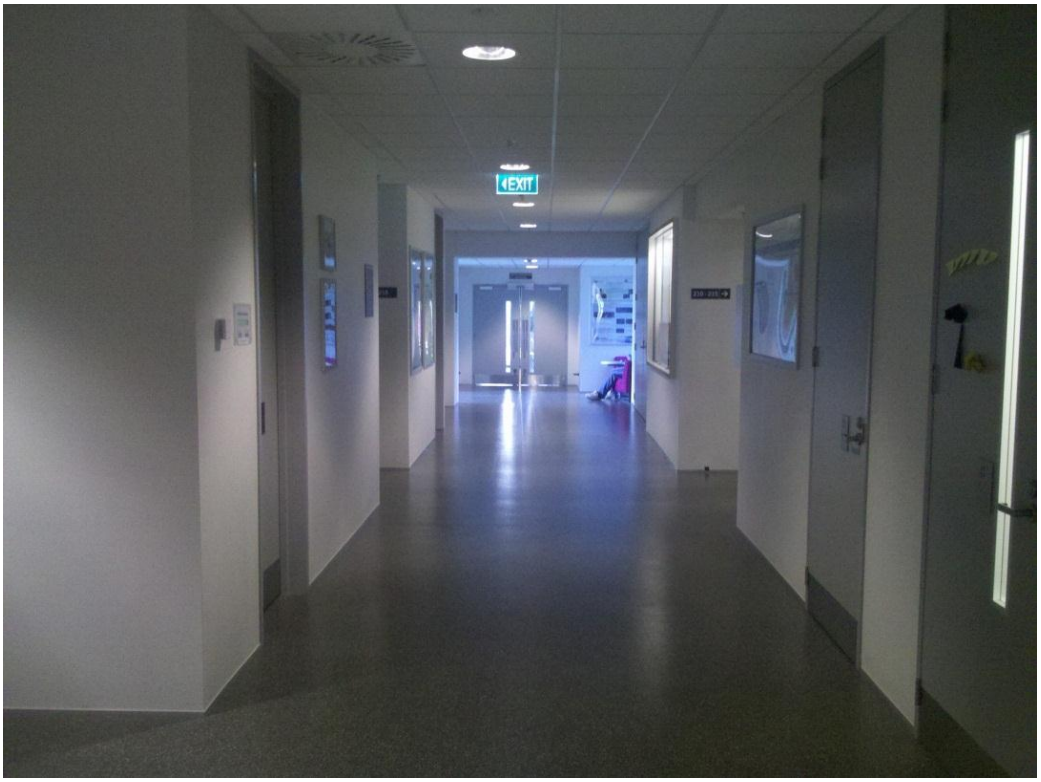

Figure 1.6 Alternative environment #1

Figure 1.7 Alternative environment #2

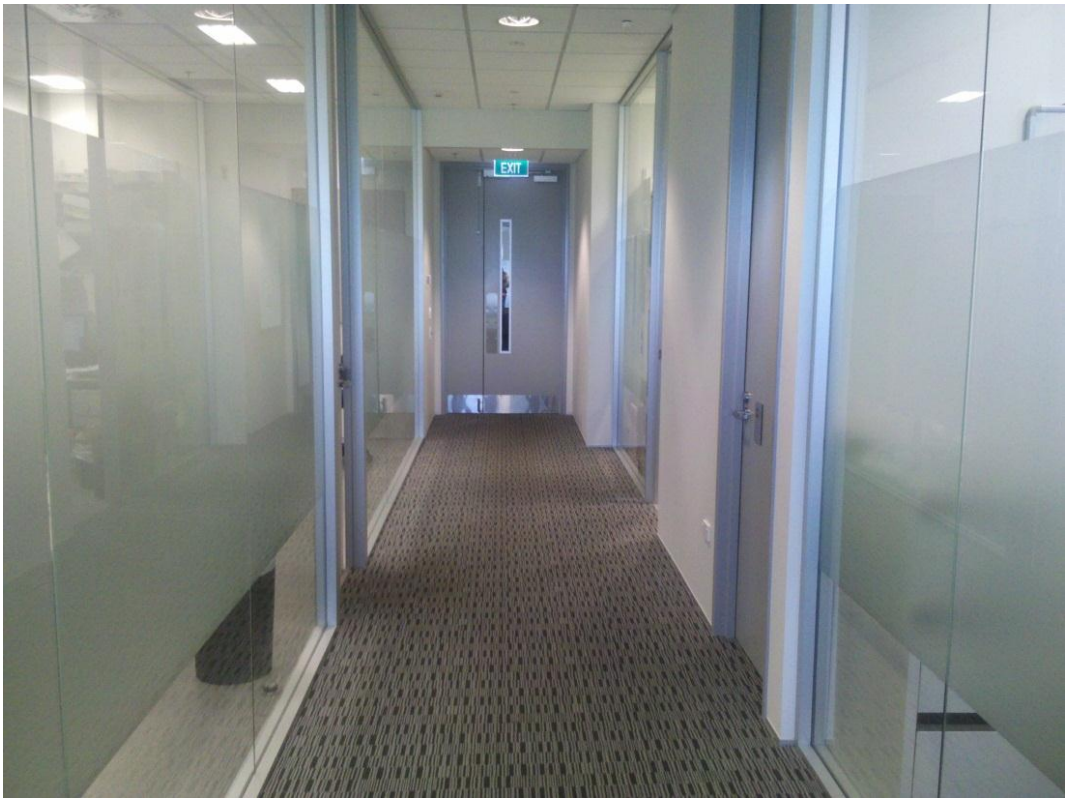

Figure 1.8 Alternative environment #3

The objective is therefore to localize the Segway in a predetermined map. This map is currently of a corridor at Victoria University but a map any environment the Segway is intended for could be created and used. Having a map also gives the ability to leave out areas where the Segway should not go such as stairways.

1.4 Chapter Summary

The thesis is organized as presented below:

Chapter 2 – Background. This chapter presents different types of control architectures for robots, followed by a review of different robotic development environments available to implement the control architecture. A review of previous robotic projects implemented using a Segway platform is also presented.

Chapter 3 – System Description. This chapter gives a detailed description of the Segway platform used in this project followed by a review of different sensors that could be used to

aid localisation and the justification of choosing the SICK LMS100 sensor. A detailed description of the SICK LMS100 sensor is then presented.

Chapter 4 – Software Interfaces. This chapter details the features available in the Robotic Development Environment, Microsoft Robotic Develop Studio (MRDS), used to interface with the Segway and the SICK LMS100 laser scanner and to develop the navigation software.

Chapter 5 – Navigation Architecture. This chapter presents the architecture of the hybrid navigation system used to control the Segway platform. The process of obtaining landmarks from sensor data and using them for localisation with odometers is also covered.

Chapter 6 – Software Description. This chapter covers the software implemented for interfacing with the hardware and the navigation architecture. A user interface designed for a human to interact with the Segway is also discussed.

Chapter 7 – Results. This chapter presents the results obtained during testing of the SICK LMS100 laser scanner and the Segway platform followed by the results of the navigation system.

Chapter 8 – Discussion. This chapter concludes the thesis by summarizing and discussing the work presented. Recommendations for future work are also discussed.

Chapter 2 Background

2.1 Introduction

This chapter begins by discussing the topics related to different robotic control architectures, namely reactive (Section 2.2), deliberative (Section 2.3) and hybrid (Section 2.4), followed by reviewing literature on previous Segway based projects. Finally, this chapter reviews five of the more common robotic development environments (RDEs), which aid designers to develop the control architectures.

A robot's control architecture provides the framework to enable functionality from different control algorithms. There are three main categories for robotic control architectures: reactive, deliberative and hybrid. Figure 2.1 shows the spectrum of deliberative and reactive robot control strategies. The left side represents methods that employ deliberative reasoning and the right represents reactive control.

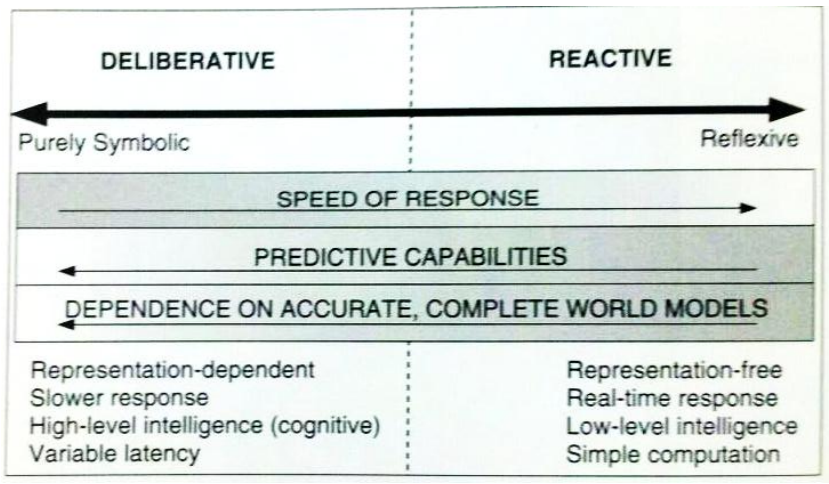

Figure 2.1 Robot control system spectrum (Arkin R. C., 1998)

2.2 Reactive Control Architectures

Reactive control architectures are characterized by a close coupling between sensing and action. Behaviour based architectures can also be classified under reactive control. Reactive controls are less dependent on a complete knowledge of the robot's environment. There are less computation requirements leading to shorter delays between perception and action

allowing reactive control systems to be faster to respond than deliberative systems. Tasks that require explicit world representations and high level intelligence can be difficult to implement in reactive systems as there is no planning component. Without this planning component, reactive architectures are unable to learn. Figure 2.2 shows the generalised makeup of a reactive control system, noting that planning is not involved.

Figure 2.2 Reactive control (Vorlesungen, 2010)

Two of the most common reactive control architectures include the subsumption architecture and the motor schema architecture.

2.2.1 Subsumption Architecture

The concept of the subsumption architecture is that each behaviour is implemented completely independently from any other behaviour. Communication between behaviours is limited to the absolute minimum. A link between a higher level behaviour and lower level behaviour is used to subsume the lower level behaviour.

Each level of competence is implemented incrementally by adding a layer of control to the existing set of levels so that the next highest level of overall control can be achieved. In an implementation of layers of control systems, a lower layer remains unaware of higher level behaviours, except for the occasional intervention by higher level behaviours to make refinements to a lower level behaviour for better performance.

The subsumption architecture shown in Figure 2.3 is one example of an approach to robot control (Brooks R. , 1985). Brooks' Subsumption Architecture avoids using a world model and instead more directly connects sensors to actuators using finite state machines to implement the appropriate actions. Behaviour-based control generalizes the augmented finite state machines into a network of behaviours that can have state and can be used to construct representations. This allows behaviour-based control to support reasoning, planning, and

learning. Figure 2.3 gives an example of a behaviour-based decomposition of a mobile robot control system. In this subsumption architecture, each item in the centre column is a behaviour.

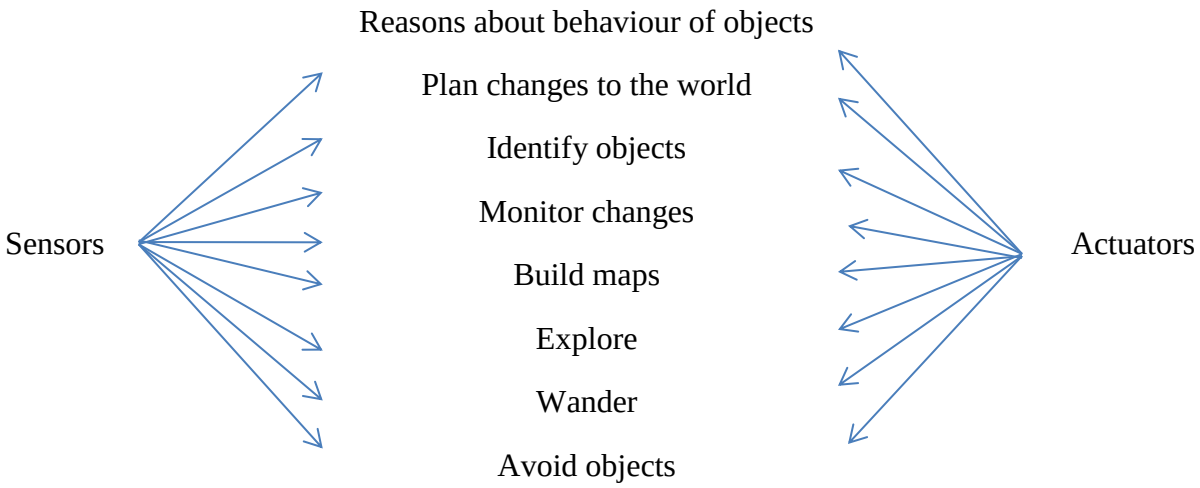

Figure 2.3 Subsumption architecture decomposition (Brooks R. , 1985)

Examples of different robots that have been constructed using the subsumption architecture include: Toto, the first map constructing subsumption-based robot (Mataric, 1992), Polly, a robotic tour guide for the MIT AI lab (Horswill, 1993) and Cog, a humanoid robot used to test human-robot interaction (Brooks & Stein, 1989).

2.2.2 Motor Schema Architecture

The motor schema architecture provides distributed and parallel behaviours that are coordinated to produce an intelligent robot (Arkin R. C., 1989). A schema is the basic unit of behaviour from which complex actions can be constructed. It consists of the knowledge of how to act or perceive as well as the process by which it is enacted. The motor schema architecture differs from other behavioural approaches in five significant ways (Arkin R. C., 1998):

- Behavioural responses are all represented in a single uniform format: vectors generated using a potential fields approach.
- Coordination is achieved through cooperative means by vector addition.

- No predefined hierarchy exists for coordination. The structure is more of a dynamically changing network than a layered architecture.
- Pure arbitration is not used; instead, each behaviour can contribute in varying degrees to the robot's overall response. The relative strengths of the behaviours determine the robot's overall response.
- Perceptual uncertainty can be reflected in the behaviour's response by allowing it to serve as an input within the behavioural computation.

Examples of different robots that have been constructed using the motor schema architecture include: George, the first robot to exhibit behaviour-based docking (Arkin & Murphy, 1990); IO, Callisto and Ganymede, three mobile robots for multi agent research (Balch, Boone, Collins, Forbes, MacKenzie, & Santamaria, 1995); and a MRV-2 mobile manipulator (Cameron, MacKenzie, Ward, Arkin, & Book, 1993).

The advantage of reactive control architectures is that the system is more efficient compared to deliberative methods (Nehmzow, 2003). There is no functional hierarchy between layers so each layer can work on different goals individually. This has the advantage that each layer can directly respond to changes in the environment as there is no central planning module which has to take account of all sub-goals. Reactive control systems are easier to design, debug and extend as the control system is built by implementing the lowest level of competence such as obstacle avoidance first, then testing before further levels are added. Reactive control systems are robust as the failure of one behaviour has only a minor influence on the performance of the whole system.

A limitation of reactive control architectures is the inability for plans to be expressed (Nehmzow, 2003). A reactive control based robot responds directly to sensory input and has no internal state memory. Therefore a reactive based control system is unable to follow externally specified sequences of actions such as: go there, pickup this, come back.

2.3 Deliberative Control Architectures

A robot employing deliberative reasoning requires relatively complete knowledge about its operating environment, commonly referred to as 'the world,' and uses this knowledge to predict the outcome of its actions (Arkin R. C., 1998). This representation enables

deliberative systems to solve certain types of problems better than reactive systems (Brooks R. , 1985). Before the development of reactive and behaviour-based architectures, deliberative reasoning methods were comprehensively used in robotic research (Arkin R. C., 1989). Deliberative control architectures are also classified as hierarchical control architectures due to their hierarchical model.

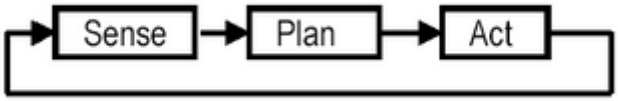

Figure 2.4 Deliberative control (Vorlesungen, 2010)

Deliberative control is a three step control method as depicted in Figure 2.4. The robot first uses data from sensors to construct a local representation of the environment, then plans and choses the directive which best achieves the current goal of the robot. Finally the robot acts to achieve the planned directive.

Deliberative architectures are hierarchical in structure with a clearly identifiable subdivision of functionality as depicted in Figure 2.5. Communication and control occurs in a predictable and predetermined manner, flowing up and down the hierarchy. Higher levels in the hierarchy provide sub goals for lower level nodes. The amount of planning decreases with lower nodes in the hierarchy as lower nodes have shorter time requirements and spatial considerations.

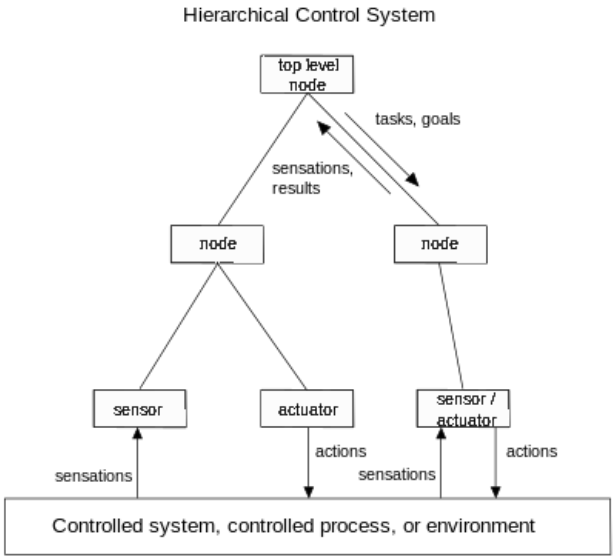

Figure 2.5 Deliberative / Hierarchical control system (Albus, 2002)

Nodes depicted in Figure 2.5 are expanded in Figure 2.6. As depicted in Figure 2.6, each node takes inputs from parent nodes and from child nodes or a sensor. A node contains four elements that interact with each other to produce the optimal performance relative to its model of the world. The four elements are sensory perception, value judgement, behaviour generation and world model. Sensory perception is responsible for receiving sensations from lower nodes as well as predicted obstacle input from the world model, then processing these into higher abstractions that update the local state. The sensory perception updates the world model to include seen obstacles and provides information to the value judgement element. The value judgement element is responsible for evaluating the updated situation and evaluating alternative plans to select the optimal solution. The behaviour generation element is responsible for executing tasks received from superior nodes as well as planning and issuing tasks for lower nodes. The world model node is the local state that provides a model for the robot and is continuously updated by higher and lower nodes.

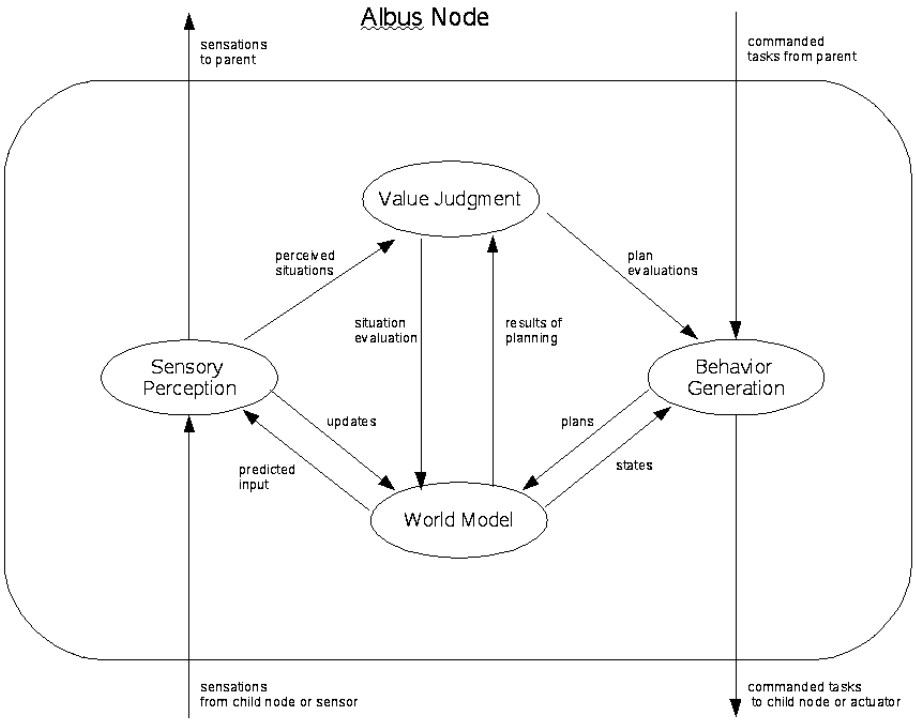

Figure 2.6 A hierarchical node for a deliberative control system (Albus, 2002)

Deliberative control methods are well suited for structured and predictable environments where a complete world model can be supplied (Albus, 2002). The disadvantage of systems relying solely on deliberative control is that they are generally too slow to cope with real world dynamic environments. World knowledge maintenance and optimal action planning have comparatively large computational efforts which are the main causes of latency.

2.4 Hybrid Control Architectures

Both deliberative control systems and purely reactive control systems have limitations when considered in isolation. Hybrid architectures combine the benefits of reactive control and deliberative control (Chand, Development of an Artificial Intelligence System for the Instruction and Control of Co-operating Mobile Robots, 2011). A high degree of flexibility is needed for successful navigation in known and unknown environments. Hybrid control architectures combine the use of high level planning and knowledge of deliberate control and the robustness, flexibility and responsiveness of reactive control. The deliberative and reactive components need to be coordinated, and different hybrid architectures decide where and how to implement this function.

2.4.1 Autonomous Robot Architecture (AuRA)

The Autonomous Robot Architecture (AuRA) (Arkin R. C., 1987) was one of the first hybrid architectures used for control of an autonomous robot. AuRA uses motor schemas for reactive control and a spatial planner for deliberative control. Figure 2.7 depicts the control components of AuRA.

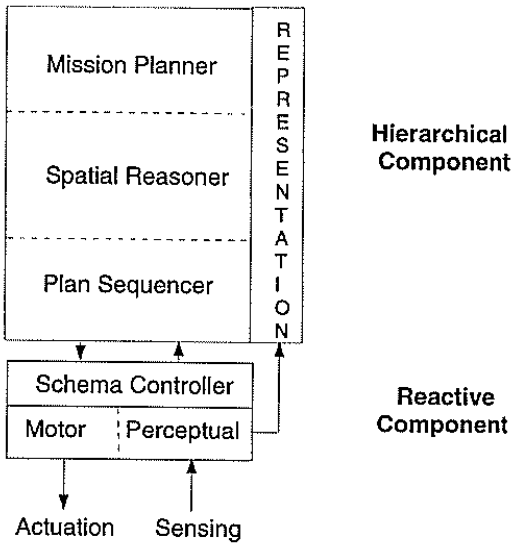

Figure 2.7 AuRA control components (Arkin R. C., 1998)

AuRA has two major planning and execution components: a hierarchical component consisting of a mission planner, spatial reasoner, and plan sequencer along with a reactive component consisting of the schema controller.

The mission planner is concerned with establishing high-level goals for the robot and the constraints within which it must operate. The spatial reasoner, or navigator system, uses knowledge about the robot's environment to construct a navigation path that the robot needs to follow to execute its mission. The path sequencer translates the navigation path into a set of motor behaviours to execute to follow the path, and then sends the collection of behaviours to the schema controller, where deliberative control ends and reactive control takes over.

The schema controller is responsible for controlling and monitoring the behavioural processes at run time. Each behaviour in the schema controller creates a response vector that gets processed and transmits the result to the low-level control system for execution.

Once reactive control begins, the deliberative component is not required unless a failure is detected in the reactive execution of the mission.

2.4.2 ATLANTIS

A three level hybrid system, ATLANTIS (A Three Layer Architecture for Navigation Through Intricate Situations), was designed by Gat at the Jet Propulsion Laboratory (Gat, 1991). The three levels are a deliberator that handles planning and world modelling, a sequencer that handles initiation and termination of low level activities and watches for reactive system failures, and a reactive controller as shown in Figure 2.8.

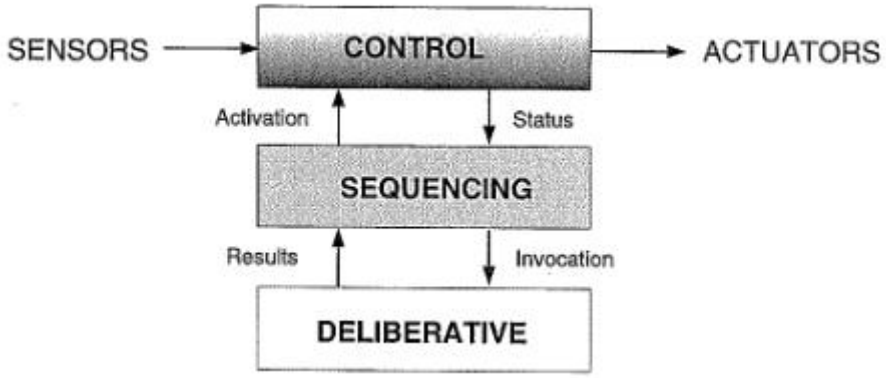

Figure 2.8 The ATLANTIS control architecture

The ATLANTIS architecture is both asynchronous and heterogeneous (Arkin R. C., 1998). The architecture is not hierarchal as none of the layers are in charge of the others; activity is spread throughout the architecture. The deliberator and sequencer acknowledge failures and adapt the reactive controller accordingly. The architecture system has been successful on both indoor and outdoor robotic systems including being tested on the Mars rover Robby (Gat, 1991), (1992).

2.4.3 Dynamic Window Obstacle Avoidance

The dynamic window obstacle avoidance method is a velocity based control system that calculates the collision free velocity for a robot to successfully reach a target goal location (Fox, Burgard, & Thrun, 1997). The kinematics of the robot are taken into account by searching the velocity space around the current position of the robot. The velocity space is all

possible sets of (v, ω) where v is all the possible velocities and ω is all the possible angular velocities for the robot during the next control cycle.

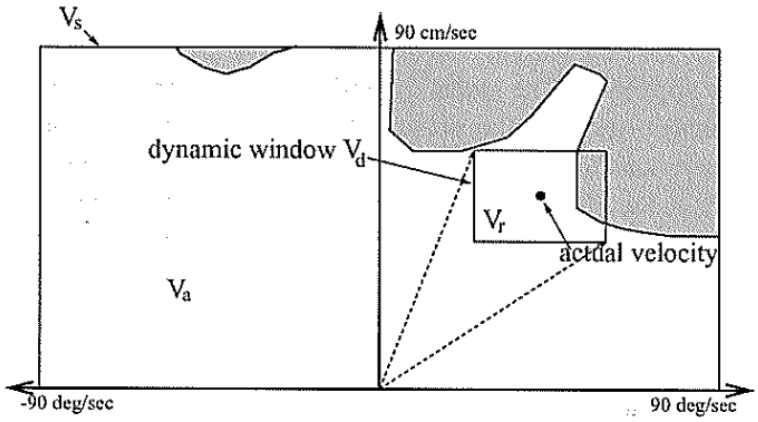

Figure 2.9 Dynamic window (Siegwart, Nourbakhsh, & Scaramuzza, 2004)

A dynamic window velocity space is visually depicted in Figure 2.9. A new motion direction is chosen by applying an objective function to all admissible velocity pairs in the dynamic window. The objective function prefers forward motion, maintenance of large distances to obstacles and alignment to the goal target (Siegwart, Nourbakhsh, & Scaramuzza, 2004).

A dynamic window hybrid navigation system has been developed by Lee-Johnson (Lee-Johnson, 2004) at the University of Waikato. Lee-Johnson's dynamic window approach supports differential drive robots and uses an A* path planning algorithm.

Chand further developed Lee-Johnson's work at Victoria University by creating a hierarchical hybrid navigation employing a dynamic window (Chand, 2011). Deliberative control was developed using a modified version of the A* path planning algorithm and a rectangular occupancy grid while reactive control was developed using a modified dynamic window approach and a polar histogram technique to avoid obstacles. The hybrid control architecture designed by Chand has been chosen as the control architecture for implementation on the Segway platform at Victoria University. The architecture has been chosen as it has been proven to be a robust navigation system (Chand, 2011) with example code available in MATLAB and C#.

2.5 Robotic Development Environments

2.5.1 Overview

Bill Gates (2007) made this statement towards standardising Robotic Development Environments:

> Robotics companies have no standard operating software that could allow popular application programs to run in a variety of devices. The standardization of robotic processors and other hardware is limited, and very little of the programming code used in one machine can be applied to another. Whenever somebody wants to build a new robot, they usually have to start from square one.

This section examines robotic control software environments. Without control software a robot is just sensors and actuators that physically arrange to create a robot but lack the capacity to interact with the real world in a useful manner.

The field of robotics faces many challenges. One of these challenges is the lack of standards both in hardware and software. This led to the need for what Kramer & Scheutz (2007) call Robotic Development Environments (RDE). Robotic development environments provide an important role for enabling the rapid advancement of the state of robotics.

Robotic development environments are intended to make creating robots easier (Kramer & Scheutz, 2007) (Pirjanian, 2005) by assisting in design, implementation, debugging and execution of a robot. An important role for an RDE is to support simulation so experimentation and debugging of new algorithms can be done without having robotic hardware available. Also RDEs should have an abstraction mechanism to make it possible to port software from one type of robot to another.

Comparisons of robotic development environments has been done several times. Kramer & Scheutz (2007) investigated nine open source RDEs while a paper by Linux Device (2008) investigates two open source and six commercial RDEs. Michal (2010) does an in depth comparison between Player/Stage/Gazebo and Microsoft Robotics Developer Studio (MRDS). Elkady & Sobh (2012) compares 17 different ‘middleware’ frameworks where

middleware was defined as “a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems.”

Comparisons of RDE’s is outside the scope of this project. Rather information from the comparisons mentioned above is summarised below and used to choose a development environment for the project.

2.5.2 Player/Stage

The first RDE summarised is Player/Stage. Player/Stage is an open source environment developed at the University of Southern California (Gerkey, Vaughan, & Howard, 2003). The Player component of Player/Stage is a robotic device hosting a server component that runs on the robot and communicates with the client application via TCP sockets. The Stage component is a 2D robot simulator that was designed to be able to simulate hundreds of robots simultaneously. A 3D simulator was later added called Gazebo. Player provides client libraries that support several programming languages including C, C++ and Python. The Player server communicates with the robot hardware itself using device specific drivers.

The Player client libraries provide generic interfaces for various robotic components that can be used to build robots. These components include features such as obstacle avoidance, vector field histogram goal-seeking, a wave front propagation path planner and adaptive Monte-Carlo localization. Player/Stage is freely available for download and is primarily used on Linux based systems. The client libraries were also specifically designed to minimize client program design constraints so that Player clients can be easily integrated with outside software .

Player refers specifically to the device and server interface. Devices are independent of one another and register with a Player server to become accessible to clients. Each client uses a separate socket connection to a server for data transfer, allowing concurrent operation of devices and ability to service multiple requests. Minimal constraints are placed on devices leaving the client the freedom of designing and implementing a control architecture.

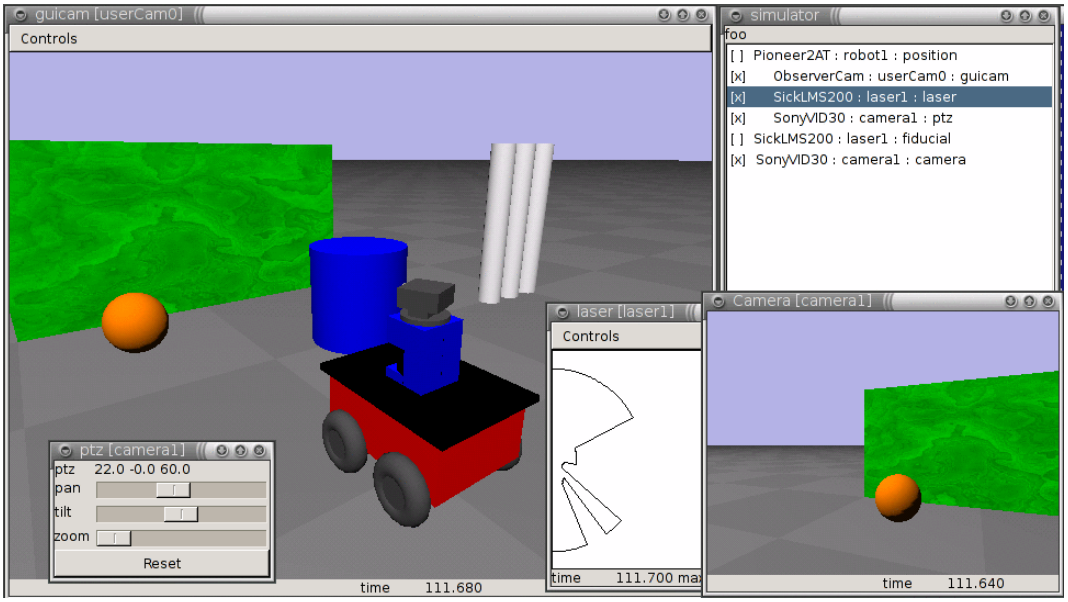

Figure 2.10 Screenshot of Player/Stage environment (Gerkey, Vaughan, & Howard, 2003)

A screenshot of the Player/Stage environment can be seen in Figure 2.10. The figure shows a Pioneer2AT robot in a simulated environment and the feedback from the attached webcam and SICKLMS200 laser scanner.

Unlike other RDEs summarised, Player/Stage is not regularly maintained (last updated 26 November 2010) and hence does not support most of the robot hardware available today (Player, 2010).

2.5.3 Robot Operating System (ROS)

Robot Operating System (ROS) (Quigley, et al., 2009) is an open source robot operating system produced and maintained by Willow Garage. ROS is not an operating system in the sense of process management and scheduling; rather, it provides a structured communications layer above the host operating system of a heterogeneous computer cluster. ROS provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes and package management.

The primary goal of the ROS project is reusability of code in robotics research and development, meaning that code written for one robot can easily be transferred and used by another robotic platform with similar capabilities.

ROS applications consist of a peer-to-peer network of processes, potentially on a number of different hosts using a loosely coupled communication infrastructure. An example of this network configuration can be seen in Figure 2.11.

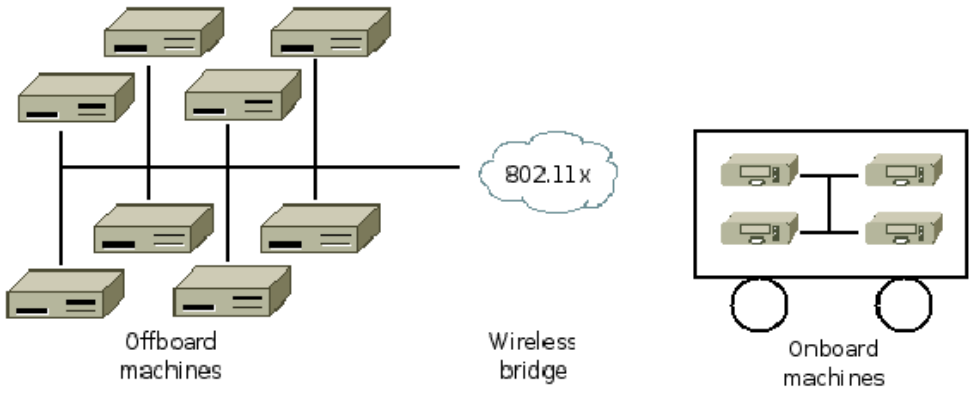

Figure 2.11 A typical ROS network configuration (Quigley, et al., 2009)

There are four main concepts for creating a ROS application: **nodes**, **messages**, **topics** and **services** (Quigley, et al., 2009).

A **node** is a process that preforms computation. A robotic system designed and implemented with ROS typically comprises multiple nodes. Nodes enable software developers to modularize ROS applications for re-use of code.

Nodes use **messages** to communicate with each other. These messages are strictly typed data structures defined within ROS.

A node sends a message by publishing it to a given **topic** which is simply a string such as "odometry" or "map". A node that is interested in a certain kind of data will subscribe to the appropriate topic. An example of this is a navigation node subscribing to the "odometry" topic for updates about the current encoder counts.

A **service** is defined by a string name and a pair of strictly typed messages, one for request and one for response messages. A service is analogous to web services, which are defined by

Uniform Resource Identifiers (URIs). Only one node can advertise a service of any particular name, just as there can only be one web service at any given URI.

"Player is a great fit for simple, non-articulated mobile platforms. It was designed to provide easy access to sensors and motors on laser-equipped Pioneers. ROS, on the other hand, is designed around complex mobile manipulation platforms, with actuated sensing". This increased functionality comes at price, "I think that it's fair to say that ROS is more powerful and flexible than Player, but, as usual, greater power and flexibility come at the cost of greater complexity" (Garage, 2012).

2.5.4 Microsoft Robotics Developer Studio (MRDS)

Microsoft Robotics Developer Studio (MRDS) was released by Microsoft in December 2006. The fundamental components of MRDS are the Concurrency and Coordination Runtime (CCR) library that allows services to be coupled together and Decentralized Software Services (DSS) which allows services to run on networked machines. CCR provides an abstraction that allows programmers to manage concurrent state updates and message processing. CCR also allows for coordination between multiple sensors and robot actuators.

MRDS defines generic contracts for robotic devices that provide an abstraction between clients and robotic hardware. MRDS client programs can also be executed in the 3D visual simulator based on the DirectX and NVidia physics engine. MRDS also provides a Visual Programming Language (VPL) that is targeted towards prototyping and novice users. VPL is integrated with Visual Studio to give the developer the ability to create a program through drag and drop blocks (activities or services) onto the design surface.

MRDS is based on the .Net framework and is primarily designed for usage with C#. Being based on the .Net framework, MRDS is only supported in the Windows operating system environment. MRDS recommends using Visual Studio as the programming environment to implement MRDS projects. MRDS is freely available for education and hobby purposes but is not open source.

2.5.5 Open Robot Control Software (OROCOS)

OROCOS (Soetens, 2010) works on a free software framework to develop a general-purpose, modular framework for advanced robot motion control (Bruyninckx, 2001). The OROCOS system contains a real-time toolkit that provides the components to be able to run on a real-time operating system.

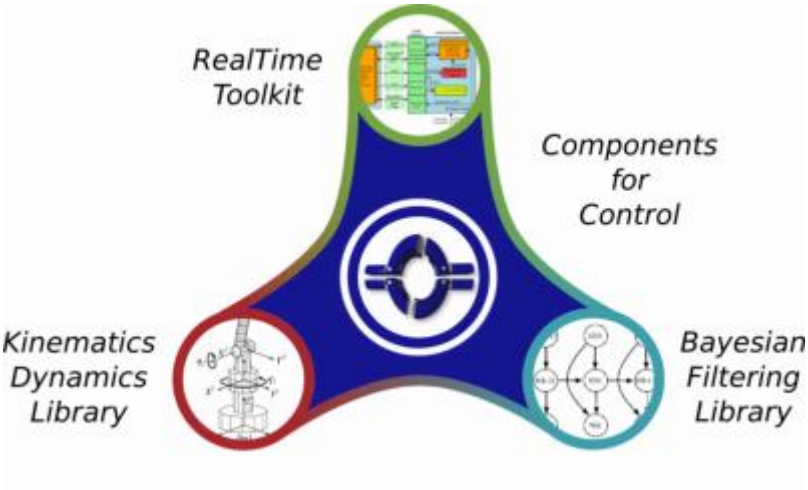

Figure 2.12 OROCOS components for controlling robots (Soetens, 2010)

OROCOS consists of the following libraries seen in Figure 2.12:

- The OROCOS Components Library (OCL) provides some ready to use control components such as the real-time toolkit. OCL also emphasises on-line interaction and component based applications.
- The OROCOS Kinematics and Dynamics Library (KDL) provides real time calculation of kinematic chains.
- The OROCOS Bayesian Filtering Library (BFL) provides an application independent framework for inference in Dynamic Bayesian Networks, such as the Kalman filter and particle filters.

The OROCOS robotic development environment does not contain a simulation environment.

OROCOS uses standards and technologies based on the Common Object Request Broker architecture (CORBA). CORBA allows inter-process and cross-platform interpretability for robot control (Henning, 2006).

A weakness in the OROCOS architecture is the lack of support for common hardware and the level of complexity in setting up the development environment.

2.5.6 Selection

As pointed out in previously completed comparisons (Michal, 2010), (Kramer & Scheutz, 2007) and (Elkady & Sobh, 2012), the real competition for a standard RDE is between MRDS and ROS. Ben Axelrod (2011) compared both MRDS and ROS and found few fundamental differences: "ROS only runs on Unix based platforms, while MRDS only runs on Windows. However, once you get past these differences, they are actually quite similar".

Elkady & Sobh (2012) tabulated attributes of different RDE's and found the only differences was that ROS was open source, while MRDS had built in security.

A previous project at Victoria University (Talwatta, 2012) was implemented using MRDS to create a standard for robotic development at Victoria. As there were few visible differences between the two RDEs, MRDS was chosen as the robotic development environment for this project to keep in line with the standard for robotic development at Victoria.

A more in depth review of how MRDS works and its available features is given in Chapter 4.

2.1 Previous Segway Platform Projects

Mobile Segway platforms have been used widely in university research projects and commercial products around the world.

The first Segway RMP platform was used to mobilise a humanoid robot called Robonaut seen in Figure 2.13 (Diftler, Ambrose, Tyree, & Goza, 2004). The Robonaut system was created at the National Aeronautics and Space Administration (NASA) in association with the Defence Advanced Research Projects Agency (DARPA) to assist human co-workers at the Johnson Space Centre with tool handling tasks. The system uses stereo vision from enabled by cameras mounted on the torso of the robot, to locate human team mates and tools, and a navigation system that uses a laser range finder alongside the vision data to follow humans while avoiding obstacles. The Robonaut platform employed a hybrid navigation system

capable of obstacle avoidance, mapping and human tracking to create a robust system capable of assisting a human by acquiring a tool from a remote location and following the human through an indoor environment with the tool for future work.

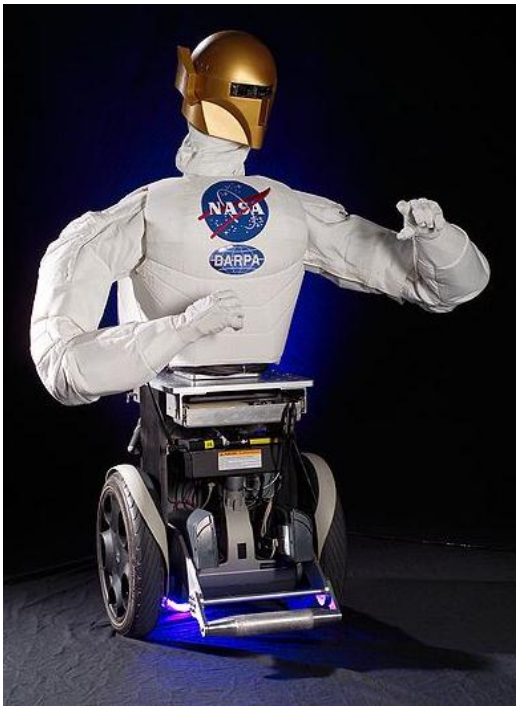

Figure 2.13 Robonaut, human assistance robot (Diftler, Ambrose, Tyree, & Goza, 2004)

Another example of a successful mobile platform using a Segway RMP is the CARDEA robot developed at MIT (seen in Figure 2.14) (Brooks, et al., 2004). This platform consists of a Segway RMP mobile base and a custom-made force controlled manipular. The mobile platform designed is capable of navigating halls, identifying and opening doors. The platform has a laptop running Linux which handles all vision processing. The platform has a sensor array made up of ten sonars, two whiskers, two cameras and a SICK LMS200 laser range finder. CARDEA uses a behavioural reactive control architecture written in Creal and runs on a custom embedded architecture called Stack.

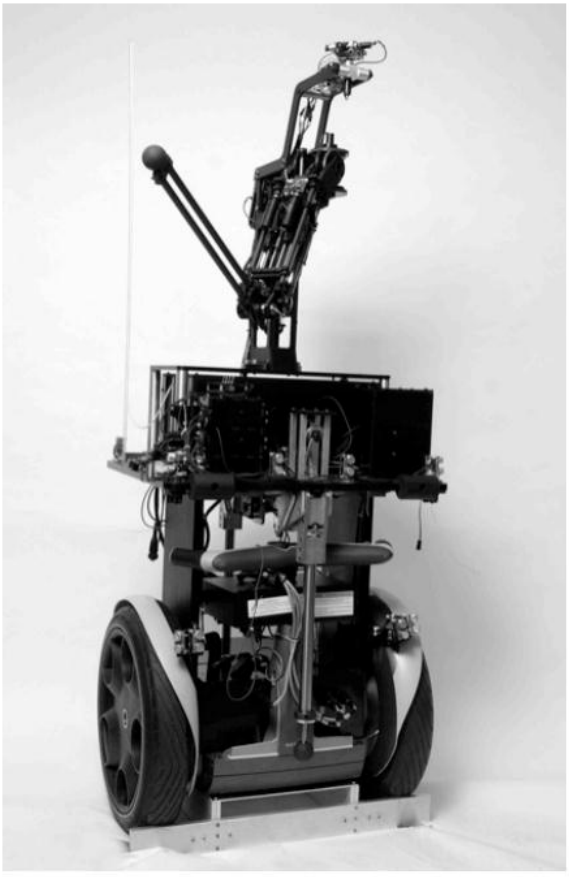

Figure 2.14 CARDEA robot system (Brooks, et al., 2004)

A Segway RMP platform was used at the Georgia Institute of Technology to mobilise a lightweight KUKA KR5 manipulator as shown in Figure 2.15 (Anderson, et al., 2008). The control algorithm used a behaviour based reactive control architecture to locate and deliver a cup of coffee. It utilizes Player/Stage as the RDE to interface with the platform hardware and a SICK LMS200 laser scanner. The system uses two laptops running Ubuntu Linux, one to control the Segway platform and the other to control the manipulator. The localisation for the system used a Markov localization method. The CARDEA robot can navigate the simple environment, a table and a human in a chair, and successfully deliver coffee from the table to a human.

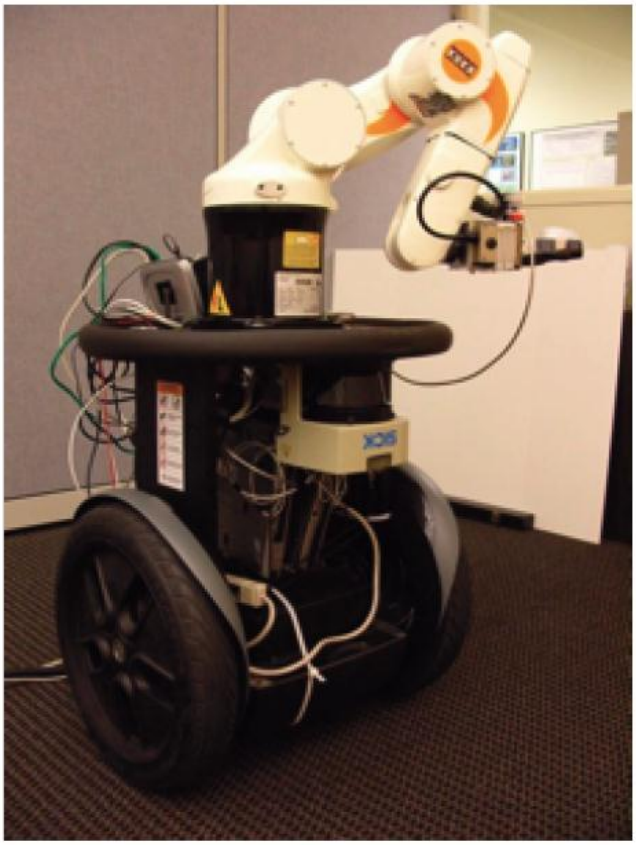

Figure 2.15 Segway project at the Georgia Institute of Technology (Mc Guire, Henriques, Nguyen, Jensen, Vinther, & Jespersen, 2009)

The Aalborg University Department of Electronic Systems acquired a Segway RMP200 platform for the purpose of making an autonomous robot shown in Figure 2.16 (Mc Guire, Henriques, Nguyen, Jensen, Vinther, & Jespersen, 2009). The project focused on trajectory planning and control for the Segway platform in an indoor environment. A SICK LMS200 laser range finder was added below the mounting plate to detect obstacles and humans, and allow localization of the robot. Localization is done with a known map using an Adaptive Monte Carlo Localization algorithm. A wavefront algorithm is used for path planning and the Nearness Diagram Plus algorithm for motion planning and obstacle avoidance. A person detector algorithm is implemented to track humans within the operating area. It uses a hybrid control architecture implemented in the Player/Stage RDE. The Segway was capable of navigating indoor human environments but had performance issues when detecting obstacles and humans.

Figure 2.16 Segway Project at the Aalborg University Department of Electronic Systems

This project differs to the above projects by employing a hierarchical hybrid navigation system using an A* path planner algorithm along with a dynamic window obstacle avoidance approach. The navigation system is built in the Microsoft Robotics Developer Studio RDE.

This project is similar to all but the Robonaut project in that the expected operating environment is an indoor controlled environment. The Segway platform does not have on board cameras like on Robonaut but employs a SICK LMS100 laser range finder, like the Aalborg University Segway platform, which has advantages for indoor navigation over the SICKLMS200 laser range finder used in the CARDEA and Georgia Institute's robots. The advantages are discussed in Section 3.2.The mounting position of the laser scanner on the Aalborg Segway unit is less than ideal as it limits the 270° angular range to around 170°. The SICK LMS100 range finder is mounted on top of the Segway for this project to allow full range use. All projects employ an autonomous navigation system with Robonaut, Georgia Institute and Aalborg University Segway projects using different hybrid navigation systems. This project employs a hierarchical hybrid navigation system using an A* path planner algorithm along with a dynamic window obstacle avoidance approach. The Georgia Institute and Aalborg University Segway projects used the Player/Stage RDE whereas the navigation

system for this project is built in the Microsoft Robotics Developer Studio RDE. A comparison between these RDE's and other common RDE's has been presented in Section 2.5.

Chapter 3 System Description

This chapter describes the system used for this project. The first section is a description of the Segway RMP200, its operating principles and main characteristics. The second section is a description of the SICK LMS100 Laser Range Finder (LRF) used as the primary distance sensor for this project. The system can be seen in Figure 3.1.

Figure 3.1 Segway System. Top left: Back view. Top right: Side view. Bottom: Front view.

3.1 Segway RMP200

3.1.1 Introduction

The Segway Personal Transporter (PT) was invented by Dean Kamen and first came to the consumer market in 2001 (Segway Inc., 2012). The Segway unit works in a similar manner to how a person walks, where the centre of gravity of the body is leaned forward and to prevent falling over, a leg is moved to stabilize the body. The Segway has two wheels instead of legs and rotates the wheels at a speed so as to prevent the operator from falling when they lean forwards or backwards. This makes the Segway TP move, and Segway Inc. calls this dynamic stabilization.

The Defence Advanced Reasearch Projects Agency (DARPA) along with the National Aeronautics and Space Administration (NASA) commissioned Segway Inc. to develop a computer controlled version of its personal mobility system capable of balancing large payloads (Diftler, Ambrose, Tyree, & Goza, 2004). In 2003 This became the Segway Robotic Mobility Platform (RMP). Segway Inc. created several robotic platforms including the Segway RMP200 (Segway Inc, 2012), which has been acquired by Victoria University of Wellington for research purposes.

The Segway is designed to be a stabilised differential drive platform that can be merged into a system to control the platform (Segway Inc, 2009). The controlling system generates velocity and steering commands to move the platform. Control commands can be sent to the RMP platform by using either the CAN bus or USB. This project controls the Segway platform using a USB interface from a control laptop.

The Segway RMP200 platform consists of a base plate, where two battery packs, engines and User Interface control box are located. The payload plate located at the top of the Segway is supported by two side panels. The Segway RMP model is depicted in Figure 3.2.

Figure 3.2 Segway RMP200

The Segway RMP200 has two different modes of operation, tractor mode and balance mode. In tractor mode the Segway platform becomes a non-stabilized differential drive system. The wheel velocities may be commanded as either a target linear velocity or target angular velocity. When tractor mode is active, another additional ground contact must be provided to prevent the platform from falling. In balance mode the Segway platform becomes a dynamically stabilized platform. Balance mode must not be used with a third point of ground contact as this interferes with balancing and causes system instability.

In this project only the balance mode will be used but both features will be available for selection in software, allowing for modular reuse for future projects.

Like most mobile robots, the Segway RMP is a nonholonomic system: "A system that is subject to constraints in velocity but not position" (Choset, et al., 2005). This means that although the Segway can reach any location, there is no singular motion command that allows it to reach all locations. An example of this is that the Segway RMP cannot move sideways without turning first.

The Segway is only suited for relatively flat terrain and has a limited range of around 19 km, making it best suited for indoor tasks. For this project the Segway platform is only expected to work in indoor environments.

The Segway has three batteries, two in the base for the Segway's control system and motors and one under the top plate for attached accessories (control laptop and laser scanner). Segway Inc. recommends the tyres be inflated to between 4 and 8 psi (27.6 kPA to 55.2 kPA). At the beginning of this project the tyres were checked and inflated to 6 psi (41.4 kPA), within the recommended pressures.

The Segway uses proprietary technology for which there is little information about the hardware within the base. Early within the project the Segway platform became inoperable and due to little information available about the hardware, debugging the issue took longer than expected. The Segway platform was required to be sent back to Segway Inc. in the USA for repair.

3.1.2 Segway Sensors

The Segway platform contains sensors that monitor the movement of the platform, enabling full control over its operation. The sensors that balance the platform are as follows: (Segway Inc, 2009)

- Five gyroscopic sensors measuring:
    - Pitch angle and pitch rate,
    - Yaw angle and yaw rate and,
    - Roll angle and roll rate.
- Two accelerometers,
- Additional tilt sensors.

With these sensors, the Segway interface also provides output information of:

- Left/right wheel speed,
- Left/right wheel shaft torque,
- Left/right wheel displacement,
- Fore/aft displacement,
- Yaw displacement and,
- Battery status.

3.1.3 RMP Interface Theory of Operation

The Segway RMP200 platform has a control architecture that consists of three distinct processors. Two processors in the base (CU_A and CU_B) are used to perform the closed loop control of the motors. These two processors perform all sensing, control and fault detection functions so that the RMP may continue to operate in the case of a fault. The third processor is a User Interface (UI) processor that manages communications to a host processor as well as providing E-stop, watchdog and programming functions for the two powerbase processors (Segway Inc., 2009).

The main processors in the powerbase of the RMP communicate with the UI processor via two CAN serial busses, CAN_A and CAN_B. The UI communicates over USB to a host processor. The control architecture can visually be seen in Figure 3.3.

The power base processors are configured with CU_A as a master and CU_B as a slave. During normal operation, CU_A computes the appropriate control command and passes the commands to CU_B on a Serial Communication Interface (SCI) communications channel inside the powerbase (Segway Inc., 2009).

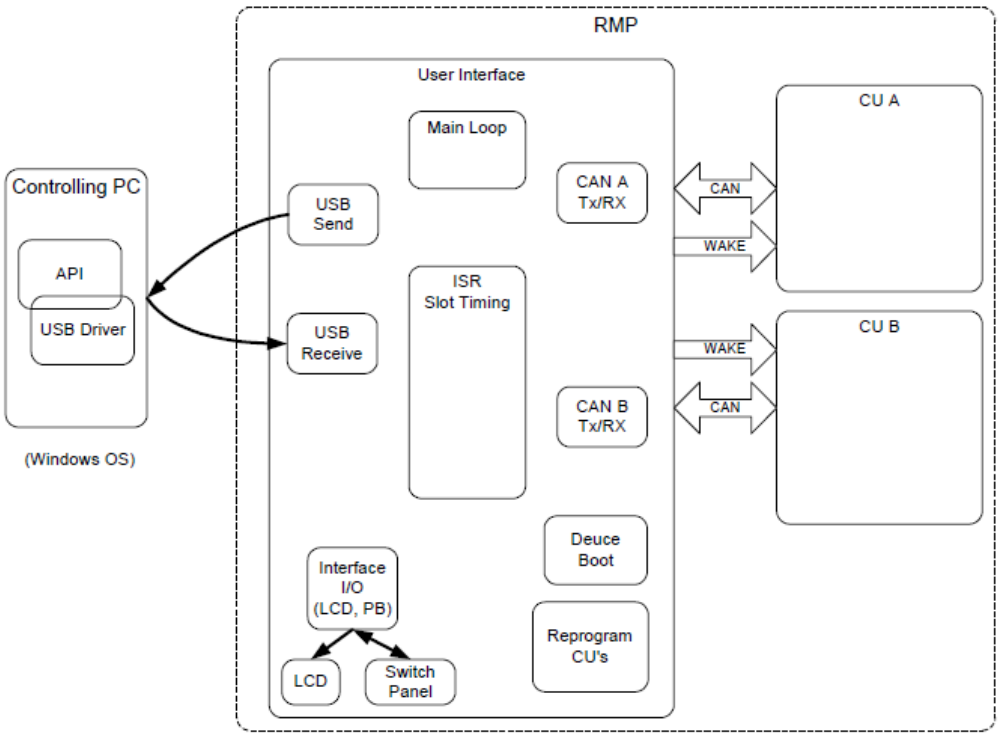

Figure 3.3 Segway RMP control architecture (Segway Inc., 2009)

Messages that control the movement of the Segway RMP may be sent by the host processor as frequently as every 10 milliseconds (100 Hz). Control messages must be sent by the host processor at a frequency no slower than 2.5 Hz or else the CU_A processor will slew the velocity command to zero. This stops the Segway in the event of a failure of the control system.

The Segway's control system starts when the green power switch as depicted in the bottom left of Figure 3.4 on the UI is pressed. The switch illuminates to indicate the UI box is powered. When the UI box is powered it is able to send and receive USB messages as well as CAN messages.

The motors are enabled when the yellow start switch as depicted in the top left of Figure 3.4 is pressed. When pressed, the WAKE line is driven high which starts the power supplies on the Control Units (CU) processor boards. This starts the wake-up procedures for CU A and CU B. When ready the CU A and CU B processors will send a CAN message to the UI to set the WAKE line low, indicating that CU A and CU B have assumed control of their own power supply. The blue tractor mode switch as depicted in the top right of Figure 3.4 will

illuminate to indicate that the Segway has successfully entered tractor mode and is ready to accept velocity commands.

For the Segway to enter balance mode, it needs to be brought into an upright position to allow engaging of the balance mode controller. Balance mode cannot be entered unless commanded by the control laptop or the blue balance mode switch is pressed as depicted in the bottom right of Figure 3.4 on the UI box. Once the balance mode button on the UI is lit, the Segway is ready to accept velocity commands.

Figure 3.4 Segway User Interface buttons

If the Segway is displaced from its desired position, it will lean against the displacement force. The harder the displacement force, the more the Segway will lean. This effect is depicted in Figure 3.5.

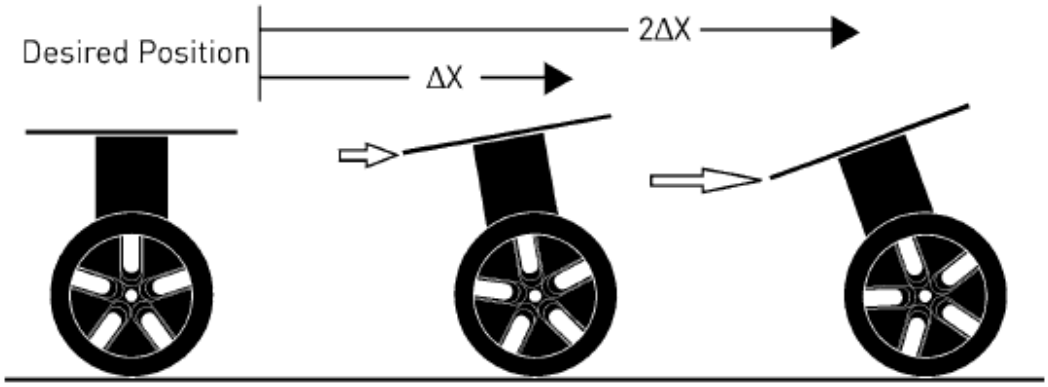

Figure 3.5 External force displacement (Segway Inc, 2009)

When the Segway is required to roll over an obstacle, the centre of gravity of the system must tilt forward over the contact point with the obstacle as depicted in Figure 3.6. Once the centre of gravity is over the contact point the Segway will roll over the obstacle provided the obstacle is small. Larger obstacles that require the Segway to tilt more than 45 degrees will cause an error within the Segway, which will cut motor power and fall over. Error conditions are explained more in Section 3.1.7.

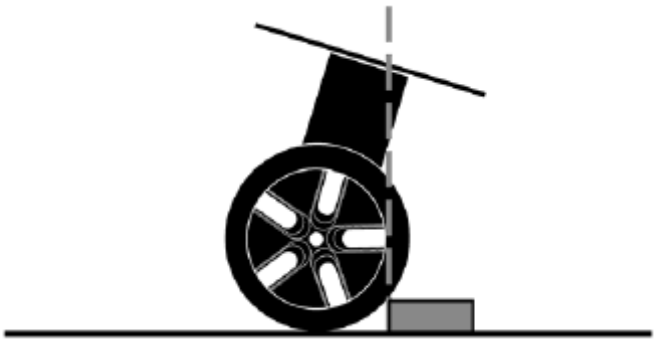

Figure 3.6 Segway traversing small obstacles (Segway Inc, 2009)

The Segway has an emergency stop switch on the UI box that causes the Segway to turn off when opened. The switch is attached to a tether as seen in Figure 3.7, that when pulled will activate the stop switch. The tether was held by the operator during initial testing to stop the Segway during an emergency.

Figure 3.7 Emergency stop switch and tether (Segway Inc, 2009)

3.1.4 USB Interface

The software included with the RMP installs the USB driver required to communicate with the Segway over serial USB. USB communication with the UI is carried out using a FTD245BM chip (Oceanchip, 2009). The installer RMPInstall.msi installs an appropriate USB driver on the controlling laptop to enable communication with the Segway platform. The RMP transmits and receives all USB communications in 18 byte packets as shown in Table 3.1.

The RMP operates internally on CAN messages. USB communications between the host computer and the RMP are essentially CAN messages with the addition of a USB header and checksum. The UI is responsible for extracting the CAN message and relaying it to CU_A. The conversion between USB and CAN is shown in Table 3.1.

Table 3.1 USB to CAN conversion

Byte	Value	Contents
0	0xF0	USB Message Header (Start Byte)
1		USB Command Identifier
	0x55	CAN Message
2		Command Type
	0x010x05	CANA_DEV USB CMD_RESET
3	0x00	Ignore on read, send as 0.
4	0x00	Ignore on read, send as 0.
5	0x00	Ignore on read, send as 0.
6		CAN Message Header (high byte)
7		CAN Message Header (low byte)
8	0x00	Ignore on read, send as 0.
9		CAN Message Byte 1
10		CAN Message Byte 2
11		CAN Message Byte 3
12		CAN Message Byte 4
13		CAN Message Byte 5
14		CAN Message Byte 6
15		CAN Message Byte 7
16		CAN Message Byte 8
17		USB Message Checksum

Byte 0 is the message header which always has the value 0xF0 indicating the start of a message. Byte 1 is the command identifier with the value of 0x55 indicating the following is a CAN message. Byte 2 is the command type where a value of 0x01 Instructs the UI to send the message contents on CAN channel A to the CU_A controller and a value of 0x05 instructs the UI processor to do a software reset. There may be more Command Types but no information is supplied on different valid commands. Bytes 3-5 are set to 0x00 when sending USB messages and ignored when received. Bytes 6 and 7 are the CAN Message Header high and low bytes. Table 3.3 contains the commands and valid parameters for configuring the Segway.

For command messages sent from the Segway, header values can be found in Table 3.5. Byte 8 is set to 0x00 when sending USB messages and ignored when received. Bytes 9-16 contain the CAN message data. For messages sent to the Segway, typical values can be found in Table 3.3.

For messages received from the Segway, typical values can be found using Table 3.4 and Table 3.5. Byte 17 is the USB message checksum. The code snippet in Figure 3.8 shows how the USB checksum is calculated.

```
unsigned short checksum;
unsigned short checksum_hi;
checksum = 0;
for(int i = 0; i < 17; i++)
{
checksum += (short)sbytes[i];
}
checksum_hi = (unsigned short)(checksum >> 8);
checksum &= 0xff;
checksum += checksum_hi;
checksum_hi = (unsigned short)(checksum >> 8);
checksum &= 0xff;
checksum += checksum_hi;
checksum = (~checksum + 1) & 0xff;
sbytes[17] = (unsigned char)checksum;
```

Figure 3.8 USB message checksum calculation

3.1.5 RMP Control Message

An RMP Control Message is used to command the RMP to move or change the platform's configuration and is 4the main means of controlling the RMP.

Table 3.2 contains the control message format, with a CAN header value of 0x0413. Bytes 1-2 contain the velocity command, with a valid range of [-1176, 1176] that maps to a velocity range of [-8 mph, 8 mph] ([-12.9 km/h, 12.9 km/h]). Bytes 3-4 contain the turning command with a valid range of [-1024 to 1024] but does not specify the angular velocity values the range corresponds with. Bytes 5-6 contain the configuration command which allows specific functions to be performed. If this command is set to zero, the configuration command and parameter, bytes 7-8, are both ignored. Table 3.3 contains the commands and valid parameters for configuring the Segway.

Table 3.2 RMP control message format

Source				Host Computer							
Destination				Command Unit Processor A							
Header – 11 bits				0x0413							
Data Length				8 bytes							
Item	Type	Size (bits)	Range	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Velocity Command	Int	16	[-1176, 1176] = [-8mph, 8mph]	X	X						
Turning Command	Int	16	[-1024, 1024]			X	X				
Configuration Command	Int	16						X	X		
Configuration Parameter	Int	16								X	X

Table 3.3 Configuration command and configuration parameter values

Command	Command Value (Int)	Parameter Range
None	0	NA
Set Max Velocity Scale Factor	10	0-16 → 0 – 1.0
Set Maximum Acceleration Scale Factor	11	0-16 → 0 – 1.0
Set Maximum Turn Rate Scale Factor	12	0-16 → 0 – 1.0
Set Current Limit Scale Factor	14	0 – 256 → 0 – 1.0
Set Gain Schedule	13	0,1,2
Set Balance Mode Lockout	15	0 = Balance Allowed 1 = Lockout Balance
Set Operational Mode	16	1 = Tractor Mode 2 = Balance Mode 3 = Power Down
Reset Integrators	50	Bitfield 1 = Right encoder 2 = Left encoder 4 = Yaw encoder 8 = Fore/Aft encoder

Scale factors are applied to the maximum velocity, maximum acceleration, maximum turn rate and to the current limit. The scale factors limit the associated value to a fraction of its full scale value. Each of these scale factors range from 0 to 1.0. Scale factors are changed by sending a control message (Table 3.2) with the associated command value (Table 3.2). Values for scale factors used in this project are discussed further in Section 6.4.2.

The acceleration scale factor allows for aggressive stopping and starting. Smaller acceleration scale factors increase the time the system takes to start moving. Larger acceleration scale factors allow for quick movement of the Segway, but could cause issues with larger payloads.

The velocity scale factor allows the controlling computer to limit the maximum speed the Segway can travel at. The scaling factor scales the [-8 mph, 8 mph] ([-12.9 km/h, 12.9 km/h]) maximum velocity linearly between 0 – 1, with a value of 0.5 limiting the maximum velocity to between [-4 mph, 4 mph] ([-6.4 km/h, 6.4 km/h]).

The turning scale factor is used to limit the maximum turning acceleration of the Segway platform. This allows the turning acceleration to be scaled down with tall payloads to prevent the Segway from tipping during turning manoeuvres.

The current limit scale factor limits the maximum motor current limit, thus limiting the amount of torque the motors can provide. Each wheel is capable of producing 122 Newton-metres of torque. Setting the current limit scale factor to 1 results in full torque capacity while

setting it to 0 results in no available torque. Dynamic balancing of the Segway requires large transient torque amounts to accelerate and decelerate. In environments with lower traction between the Segway's wheels and the ground, reduction of the maximum torque is required to prevent wheel slippage.

The Segway has three different gain schedules depending on the payload configuration. The different payload configurations are depicted in Figure 3.9. Selecting the correct gain schedule for different payloads allows the Segway to improve the handling and dynamics of the internal control loop, giving better damped responses to velocity and turning commands. Gain schedule 0 is optimised for light payloads of around 50 lb (22.7 kg) on the top plate, gain schedule 1 is optimised for tall payloads where a 25 lb (11.3 kg) payload is located at the top plate and another 25 lb payload is located 75 cm above the top plate, and gain schedule 2 is optimised for heavy payloads of around 100 lb (45.4 kg) on the top plate.

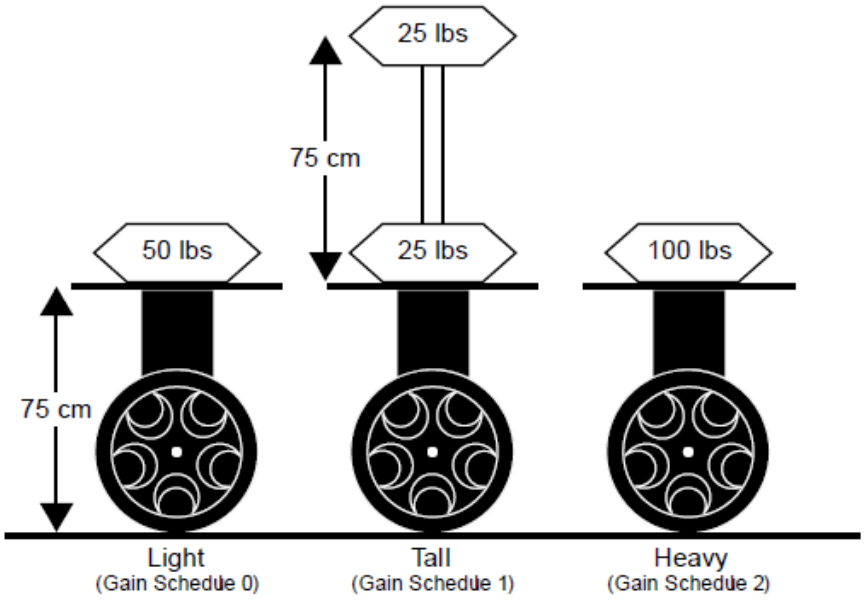

Figure 3.9 Payload configurations for the Segway (Segway Inc, 2009)

The set operational mode parameter enables the control computer software to change between tractor mode and balance mode as well as allowing the ability to turn off the Segway. Once the Segway is turned off, it needs to be manually turned on again.

The reset integrators parameter allows the encoder values to be reset. Each encoder can be individually reset while a bitwise OR function between different values in Table 3.2 can reset multiple encoders in a single command message.

3.1.6 Monitoring Messages

Monitoring messages are sent from Command Unit Processor A to the controlling computer at 100 Hz. These messages are important as they provide state estimates on the Segway to the host processor, supplying information such as current wheel speeds and encoder values. Each message contains four data slots as shown in Table 3.4. Each data slot is 16 bits long (two bytes).

Table 3.4 Monitoring messages packet format

Source			CU processor A							
Destination			Host Processor							
Header – 11 bits			See Table 3.5							
Data Length			8 bytes							
Item	Size (bits)	Range	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
Data slot 0	16	[0,0xFFFF]	X	X						
Data slot 1	16	[0,0xFFFF]			X	X				
Data slot 2	16	[0,0xFFFF]					X	X		
Data slot 3	16	[0,0xFFFF]							X	X

Seven different monitoring messages are sent from the Segway to the control computer and can be seen in Table 3.5.

- Message 1 contains the pitch angle, pitch rate, roll angle and roll rate.
- Message 2 contains the left and right wheel velocities, yaw rate and servo frame counter. The servo frame counter increments from frame to frame. A frame is the set of 8 messages the Segway sends to the control computer.
- Message 3 contains the left and right encoder values. Both are 32 bits long (four bytes) so require two data slots per message.
- Message 4 contains the fore/aft and yaw encoder values. Both are 32 bits long so require two data slots per message.
- Message 5 contains the left and right motor torque values.

- Message 6 contains the current operation mode, current controller gain schedule, the UI battery voltage level and the powerbase battery voltage level.
- Message 7 echoes the received velocity and turn commands back to the control computer and is useful for debugging.

Table 3.5 Monitoring messages and conversions

Monitoring Message	Header – 11 bits	Message Contents	Data Conversion
Monitor Data Message 1	0x0401	Data Slot 0: Pitch Angle Data Slot 1: Pitch Rate Data Slot 2: Roll Angle Data Slot 3: Roll Rate	7.8 counts / degree 7.8 counts / degree/sec 7.8 counts / degree 7.8 counts / degree/sec
Monitor Data Message 2	0x0402	Data Slot 0: LW Velocity Data Slot 1: RW Velocity Data Slot 2: Yaw rate Data Slot 3: Servo Frame counter	332 counts / m/sec 332 counts / m/sec 7.8 counts / degree/sec 0.01 sec / frame
Monitor Data Message 3	0x0403	Data Slot 0: Left Encoder (Low) Data Slot 1: Left Encoder (High) Data Slot 2: Right Encoder (Low) Data Slot 3: Right Encoder (High)	33215 counts / m 33215 counts / m
Monitor Data Message 4	0x0404	Data Slot 0: Fore/aft Encoder (Low) Data Slot 1: Fore/aft Encoder (High) Data Slot 2: Yaw Encoder (Low) Data Slot 3: Yaw Encoder (High)	33215 counts / m 112644 counts / revolution
Monitor Data Message 5	0x0405	Data Slot 0: Left Motor Torque Data Slot 1: Right Motor Torque Data Slot 2: NA Data Slot 3: NA	1094 counts / Nm 1094 counts / Nm
Monitor Data Message 6	0x0406	Data Slot 0: Operational Mode (0= disabled, 1=tractor, 2= balance) Data Slot 1: Controller Gain Schedule (0,1,2) Data Slot 2: User Interface Battery Voltage Data Slot 3: Powerbase Battery Voltage	 1.4 + counts * 0.0125 4 counts / Volt
Monitor Data Message 7	0x0407	Data Slot 0: Velocity Command (as received) Data Slot 1: Turn Command (as received) Data Slot 2: NA Data Slot 3: NA	332 counts / m/sec 332 counts / m/sec

The recommended data conversion factors are also shown in Table 3.5. The pitch angle, pitch rate, roll angle, roll rate and yaw rate are estimates that come from a pitch state estimator within the Segway's control processor. It synthesizes low frequency and high frequency sensors to arrive at estimates of angles and angular rates. Segway advise that high acceleration or rough terrain reduces the accuracy of the numbers. The conversion factors for

all encoders are based on the nominal rolling diameter of the wheels of 48 cm. As these are only approximates, more accurate conversion factors are required and shown in Section 7.2.1.

3.1.7 Error Conditions

The Segway RMP can encounter certain environmental conditions that prevent the platform from maintaining self-balance. When a fault or malfunction is sensed by the power base the system slews the velocity command to zero but keeps the motors enabled to allow system stabilisation. When a fault prevents the system from maintaining stabilisation the system will disable power to the motors, causing the Segway to fall or roll freely. When the Segway encounters these problems, the Segway disables power to the motors, thus preventing possible damage to the surrounding environment.

If the pitch angle of the Segway exceeds 45 degrees forwards or backwards, an error has occurred and the Segway will disable power. This is because the Segway controller has to travel at an excessive speed to restore balance once the Segway has tilted past this angle. An excessive roll angel of 60 degrees will also cause an error and cause the Segway to disable power.

When in balance mode, the Segway balance controller is designed to hold a stationary position based on several controller error terms, such as wheel displacement from commanded location. If the Segway moves more than 12 feet (3.66 m) from the original resting location the Segway will disable balance mode and switch to tractor mode. This error condition can occur if the wheels are slipping, an external disturbance force pushes the Segway away from equilibrium position or if a wheel is lifted off the ground.

The Segway is designed with a redundant propulsion system (Segway Inc, 2009). The system maintains electrical isolation between the frame and control electronics in order to detect the event of electrical component failure. If an electrical connection is made between two systems, the Segway performs a safety shutdown. Segway advise that the most common cause of this fault is connecting the CAN channel ground to the frame of the machine and recommend an optically isolated cable be used for any CAN based communication. This project does not require this due to using the USB communications architecture.

3.2 Range finders

Three laser range finders have been identified as being commonly used in robotic applications. They are the SICK LMS100, SICK LMS200 and the Hokuyo URG. A description of each sensor is given in the following sections along with the sensor chosen for this project.

3.2.1 SICK LMS100

The LMS-100 scanner has a maximum measurement range of 20 metres with a programmable field of view (FOV) up to 270°. The 270° FOV can be measured with an angular resolution of either 0.25° or 0.5° at a scan frequency of 25 or 50 Hz. The scanner weights 1.1 kg and consumes 350 mA at 24 V supply voltage. The SICK LMS100 dimensions are 105 x 102 x 152 mm. RS-232, CAN and Ethernet data interfaces are available. The scanner is capable of TCP/IP communication through its Ethernet port, thus the available bandwidth is sufficient to transfer 270° FOV measurements with an angular resolution of 0.5° at 50 Hz.

Figure 3.10 The SICK LMS100 laser range finder

3.2.2 SICK LMS200

The SICK LMS200 (Figure 3.11) has been frequently used in robotic applications for obstacle recognition and avoidance as discussed in a review by Mc Guire, Henriques, Nguyen, Jensen,

Vinther & Jepersen (2009). The LMS200 has a maximum measurement range of 80 m, far greater than the 20 m maximum of the LMS100 scanner. It also has a maximum 180° field of view, 90° less than the LMS100 counterpart. The angular resolution of the scanner is 0.25°, 0.5° and 1° with an 18.9 Hz, 38.5 Hz and 77 Hz scan rate respectively. The scan rate at 0.25° and 0.5° is far slower than the 50 Hz that the SICK LMS100 is capable of. The scanner weighs 4.5 kg, over four times heavier than the LMS100, and consumes 830 mA at a 24 V supply voltage, more than twice the 350 mA at 24 V for the LMS100. The LMS200 communicates with RS-232 with a maximum communication rate of 500 Kbaud/s. Cang Ye and J. Borenstein (2002) worked on a detailed characterization on the LMS-200 laser scanner. Pre-made services have been developed in MRDS (Johns & Taylor, 2008) for the SICK LMS200.

Figure 3.11 SICK LMS 200 laser range finder (SICK Inc., 2003)

3.2.3 Hokuyo URG

The Hokuyo URG (Figure 3.12) is one of the smallest laser range finders available measuring 50 x 50 x 70 mm. The Hokuyo scanner has a maximum measurement range of 4 m, much less than the 20 m for the LMS100, with a 240° FOV, slightly less than the 270° FOV for the LMS100. The angular resolution is 0.36°, comparable to the 0.25° and 0.5° options available

from the LMS100. The scan rate is 10 Hz which is much slower when compared to other measurement systems. It has RS-232 and USB data interface for communication up to 12 Mbit/s.

Figure 3.12 Hokuyo URG-04LX laser range finder

3.2.4 Chosen Sensor

The SICK LMS100 was chosen for this project. It out-performs the LMS200 in most aspects, larger FOV, faster scan rates, lighter and less power requirement, except the maximum measurement range of 80 m compared to 20 m. The increase in maximum measurement range is not required for this project as a maximum of 20 m is adequate to localise and detect obstacles within an indoor environment.

The Hokuyo laser range finder has only a slightly worse FOV when compared to the LMS100 as well as a slower scanning rate. With a small form factor, low weight and low power requirements the Hokuyo could be used as an alternative range finder device, although the 4 m measurement range could make localisation harder as less features would be extracted each scan.

3.3 SICK LMS100 Laser Scanner

3.3.1 Overview

The SICK LMS100 consists of a laser reflected by a rotating mirror. The operation of the laser is based on an infra-red pulsed laser diode, and the internal firmware includes the ability to report the intensity of reflection (SICK Inc., 2012). The device can be seen in Figure 3.10.The LMS measures its surrounding in two-dimensional polar coordinates as shown in Figure 3.13. The distance is measured by the time between emitting and receiving a laser pulse from the laser scanner, known as pulse propagation time measurement and can be seen in Figure 3.14. If a laser beam is incident on an object, the position is determined in the form of distance and direction.

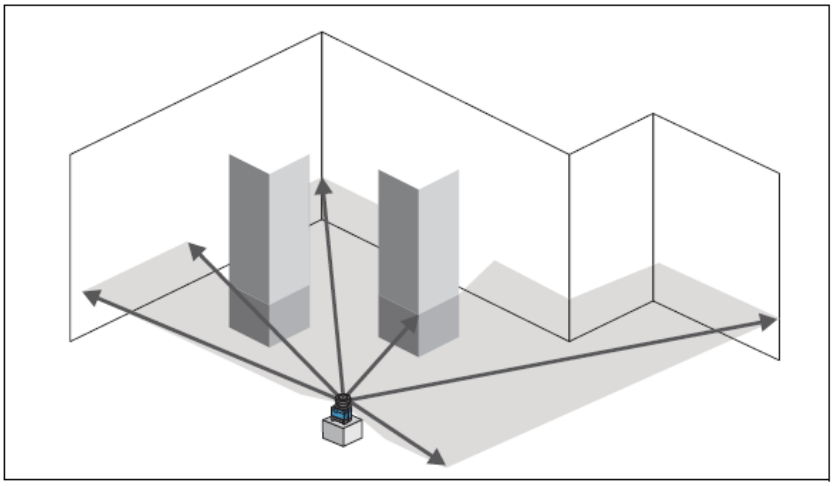

Figure 3.13 Measuring principle of the LMS

The SICK LMS 100 purchased by Victoria University does not have a pre-made service in MRDS and one needed to be developed. MRDS services are explained in greater detail in Chapter 4.

The SICK LMS 100 rangefinder is the primary sensor used in this project for localisation and obstacle avoidance for the Segway platform. It is connected to the controlling laptop using the 10/100 Mbit Ethernet (TCP/IP) interface.

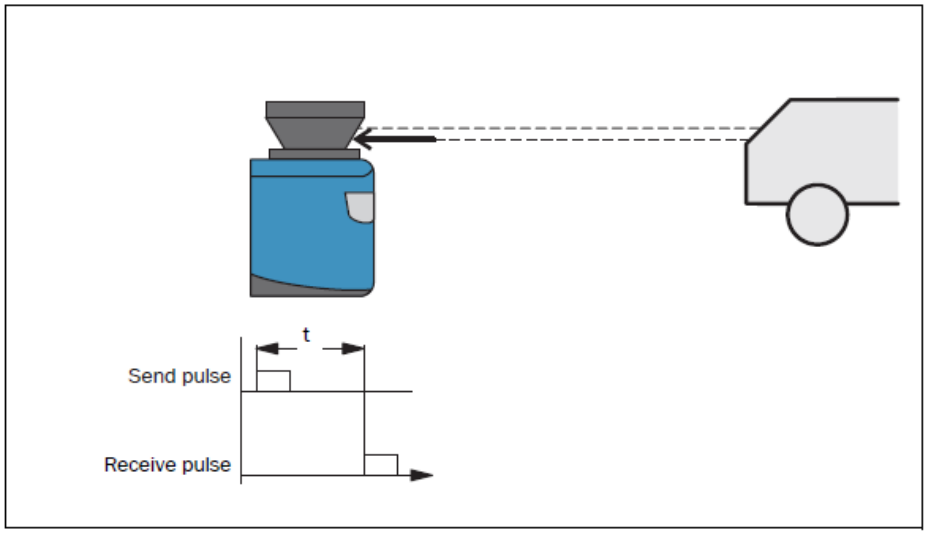

Figure 3.14 Principle of operation for pulse propagation time measurement

3.3.2 Data Communication using Telegrams

The SICK LMS100 uses "telegrams" to communicate between the unit and a host environment. Telegrams are the packet structure, or framework, used for communication between devices connected to the scanner. This project only utilises telegrams relating to:

- starting the laser scanner running,
- requesting single or continuous laser measurements, and
- stopping the laser scanner running.

Telegrams relating to configuring the SICK LMS100 (such as changing the scanning resolution) are not implemented as the laser scanner can be configured using the SOPAS Engineering Tool (Informer Technologies Inc., 2012) in an easier manner.

The LMS sends telegrams over the interfaces described above to communicate with the connected host. The following functions can be run using telegrams (SICK Inc., 2012):

1) requests for measured values by the host and subsequent output of the measured values by the LMS,
2) parameter setting by the host for the configuration of the LMS, and
3) parameters and status log querying by the host.

The IP address of the SICK LMS100 was changed to 130.195.162.58 using SOPAS so that it could network with the University computers on the 130.195.162.xxx domain.

There are two encoding options for telegrams that the laser scanner can interpret: ASCII and binary. For this project, all telegrams sent to the laser scanner (and subsequently received) use ASCII encoding. This decision was made because it is visually simpler for a human to see ASCII encoding rather than binary encoding (Figure 3.15) and that the main programming language, C#, being a higher level programming language is more suited towards ASCII support than binary. The disadvantage of using ASCII was that the start and end frame bytes did not correspond to ASCII characters recognised by the IDE used during the project. This was overcome by creating start and end frame header bytes and employing byte to string methods that converted the start and end frame characters at runtime.

ASCII	<STX>sRN{SPC}LMDscandata<ETX>
Binary	02 02 02 02 00 00 00 0F 73 52 4E 20 4C 4D 44 73 63 61 6E 64 61 74 61 05

Figure 3.15 ASCII vs binary telegram example

Figure 3.15 shows two telegrams requesting the output of measured values of one scan. The top image is an ASCII telegram while the bottom image is a binary telegram (values converted to HEX for visualisation). This gives an example of how it is easier to visually see which telegrams are being sent and received

The telegrams supported by this project are: sRN LMDscandata and sEN LMDscandata and their response messages. These message types are explained in more detail in Section 6.3.

3.4 Control Laptop

A laptop was chosen to be the main control computer to host high-level software. This is due to the requirements for running Microsoft Robotics Developer Studio. The requirements, listed below, rule out using an embedded controller for this project.

The requirements for the Microsoft Robotics Developer Studio 4 runtime environment are:

- a PC or laptop capable of running Windows 7,
- dual-core processor (2 GHz or faster recommended),
- 2 GB of memory, and
- directX 9.0c compatible graphic card (for simulation).

The Segway platform requires a USB connection for communication while the SICK LMS100 requires a TCP/IP Ethernet connection.

The specifications for the laptop used are as follows:

CPU:	Intel Core i5-2520M @ 2.50 GHz
RAM:	4.00 GB
Hard Disk:	250 GB, 5400 rpm
OS:	Windows 7 Enterprise SP1
I/O Ports:	3 x USB 2.0
Connectivity:	Intel Gigabit Ethernet
	Intel Advanced 802.11n WLAN

The chosen laptop easily meets the specifications for running the MRDS runtime environment.

A car laptop charger adapter (12 V, 90 W) was modified and connected to the 12 V battery under the top plate on the Segway platform to charge the laptop.

3.5 Complete System

An overview of the complete system can be seen in Figure 3.16. A remote PC is used to monitor and control the system and runs the UI explained in Section 6.6. The remote computer uses wireless to communicate with the control computer. The control computer runs the navigation service (Chapter 5) which controls movement of the Segway. The control computer also runs two services (explained in Sections 6.3 and 6.4) which communicate over TCP/IP and USB to control the SICK LMS100 scanner and Segway platform.

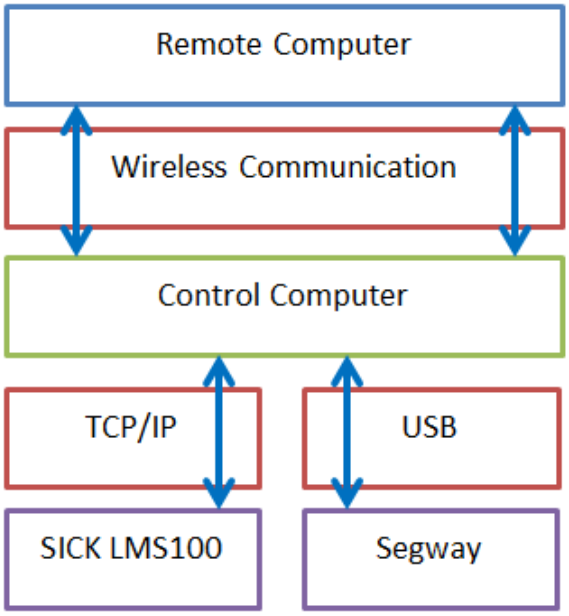

Figure 3.16 System overview

The SICK LRF scanner is mounted in the centre top of the platform. Mounting the sensor on the top of the Segway platform allows the full 270° field of view to be un-obstructed by the Segway itself during normal operation. Some previous projects using Segways and laser range finders mounted the scanner underneath the top plate, as seen in Figure 2.15 and Figure 2.16, which limited their field of view to 180° directly in front of the Segway, as distance measurements from larger fields of view returned distances to the vertical mounting plates on the Segway. Mounting the laser scanner on top of the Segway means that any distance measurements returned by the laser scanner are distances from centre of the Segway to obstacles, rather than incorporating any part of the system.

A piece of acrylic sheet was laser cut and mounted above the base of the Segway, between the two vertical plates, to create a platform for the control laptop to sit on. The lower platform was cut to be smaller than the top plate of the Segway so the footprint size would not increase. This addition can be seen in Figure 3.17.

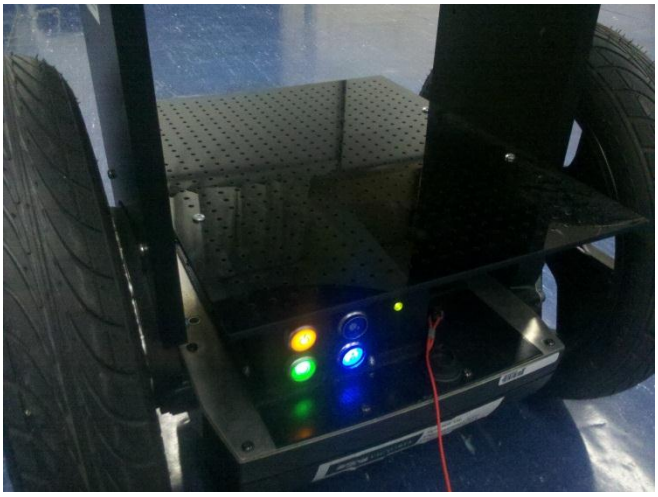

Figure 3.17 Laptop platform

One issue that is predicted to cause problems is the changing pitch angle when the Segway moves, which also changes the angle of the laser scan relative to the ground the Segway is traveling on. Figure 3.18 depicts a sketch of the Segway when tilted at a 10 degree angle and a laser range measurement pointing forward relative to the Segway.

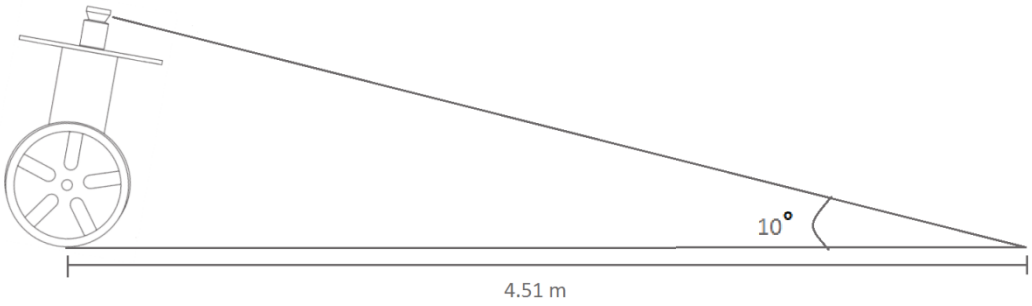

Figure 3.18 10° tilt of Segway effect on range finder

With a placement of the SICK LMS100 in a height of 80 cm, and a pitch angle of 10 degrees (not unrealistic during acceleration) the laser range will hit the floor in a distance of 4.51 m from the Segway platform. This could confuse the localization algorithm, since it will look like a wall. Possible solutions could be to mount the LMS with a motor, hang it freely to always level it, or use geometry to improve the range readings. However, during testing this issue did not affect the performance of the localization and landmark detection by a pertinent amount. As this was not a consideration, fixing the issue is not in the scope for this project.

Chapter 4 Microsoft Robotic Developer Studio

This chapter describes the software development language MRDS, the environment and tools that were used to develop the control system for the Segway platform. The Segway platform's software is written in two programming languages, mostly C# with C used to interface with the Segway platform. The navigation system has been implemented in C# running in the MRDS environment. MRDS is designed to execute on any Windows-based PC that meets the specifications laid out in Section 3.4.

4.1 MRDS

Microsoft created Microsoft Robotic Developer Studio (MRDS) (first released 2006) for the purpose of creating an industrial standard in robotics and incorporates a Service-Oriented Architecture (SOA) into embedded system development (Microsoft, 2012). SOA is characterized by loosely coupled services, open standard interface, service publication, dynamic discovery of services and dynamic composition using services discovered (Tsai, Huang, & Sun, 2008). MRDS provides a software platform and development environment that enables software written for one robot to also work with another robot with similar capabilities (Jackson, 2007).

As MRDS is designed to run in the .Net based runtime environment, MRDS applications require Windows operating systems to run them.

Following the SOA design, application modules interact as a service that subscribes to or publishes to other services, similar to Web services.

MRDS also defines a set of abstract services specifying APIs that can be used to communicate with common hardware components. These services allow MRDS to control a wide range of hardware with minimal programming effort. An example of this is the Generic Differential Drive (GDD) contract service which provides a framework for differential drive robots and allows other generic services, such as the dashboard service, to interact with them. The dashboard service shown in Figure 4.1 can be used to drive any GDD robotic platform

with a keyboard or joystick (Johns & Taylor, 2008). The dashboard service also connects to a SICK LMS200 laser range finder and displays the distance measurements. The dashboard can find GDD services on remote nodes when given a computer name and port to connect to and log also provides a logging function.

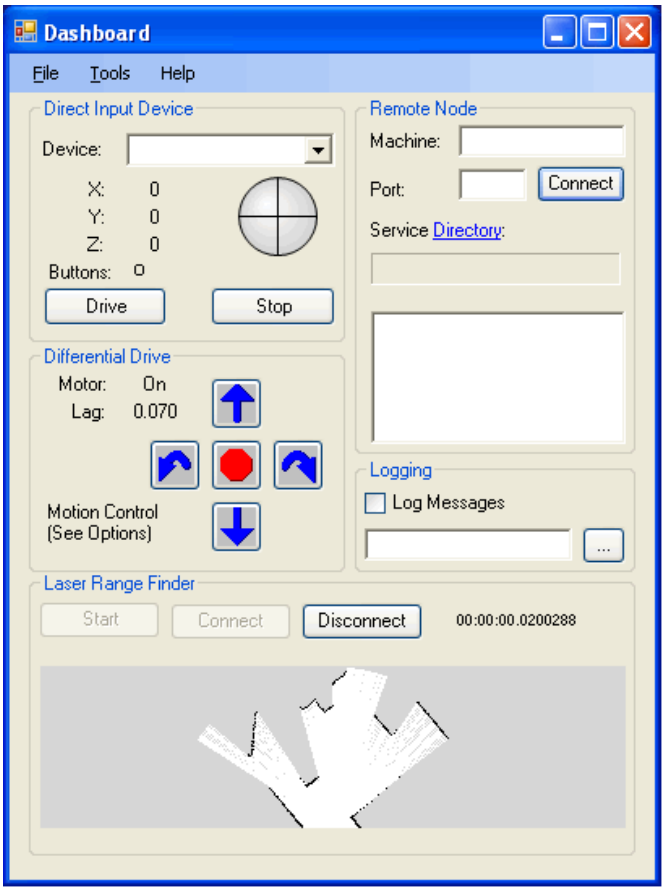

Figure 4.1 Dashboard service

The basic building block in MRDS is a *service*. Every MRDS application will contain one or more services. Services can be combined as *partners* to create robotic applications. This process is referred to as *orchestration*. Figure 4.2 shows an example of how the services might be orchestrated to control a robot. It is the job of the orchestration service to implement high-level control behaviours such as path planning and obstacle avoidance.

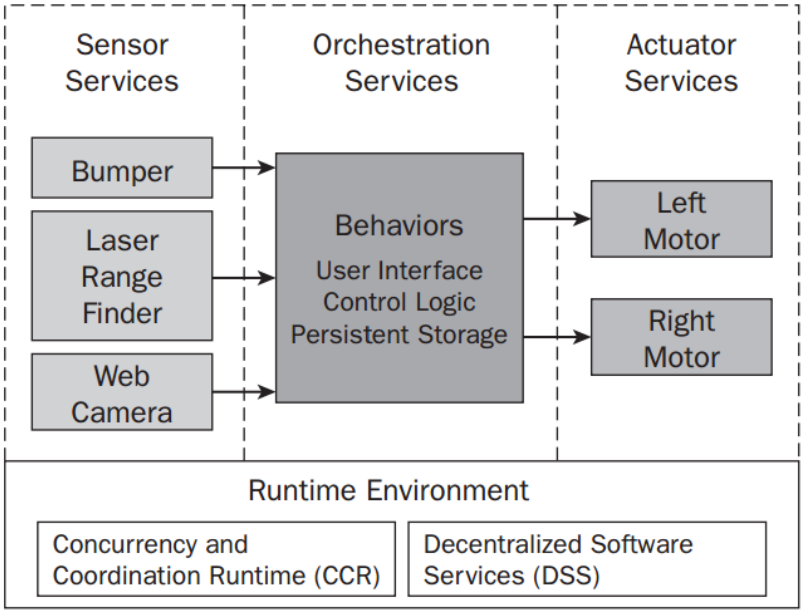

Figure 4.2 MRDS operational schema (Johns & Taylor, 2008)

The MRDS environment consists of a number of components . The Concurrency and Coordination Runtime (CCR) and Decentralized Software Services (DSS) shown in Figure 4.2 are covered in more detail in Sections 4.1.1 and 4.1.2.

MRDS also includes utility services which automatically load when a service is started. These include:

- a control panel service which provides a web interface to the end user displaying all the running services and current state or web transform for each service. A Web transform is how the service state is displayed in a web browser,
- a logging service that provides debugging and diagnosis interface, and
- a resource diagnostic service to provide additional information to assist in debugging and performance evaluation.

In addition, MRDS consists of two visual components, a 3D simulator, Visual Simulation Environment (VSE) shown in Figure 4.3, and a Visual Programming Language (VPL) shown in Figure 4.4.

Figure 4.3 MRDS 3D Visual Simulation Environment

The Visual Simulation Environment uses 3D graphics to render a virtual world and a physics engine to approximate interactions between objects within the virtual world. The VSE is designed to help prototype new algorithms and robots when actual hardware is not available. Without a simulator, prototyping new robot designs and moving from one design iteration to the next can take weeks or months due to the physical changes required. Using a simulator significantly reduces this time period. A simulator also enables easy design and debugging of software when compared to physical robots. With moving robots it is often difficult to debug errors but simulations can avoid this problem as they can be paused when required.

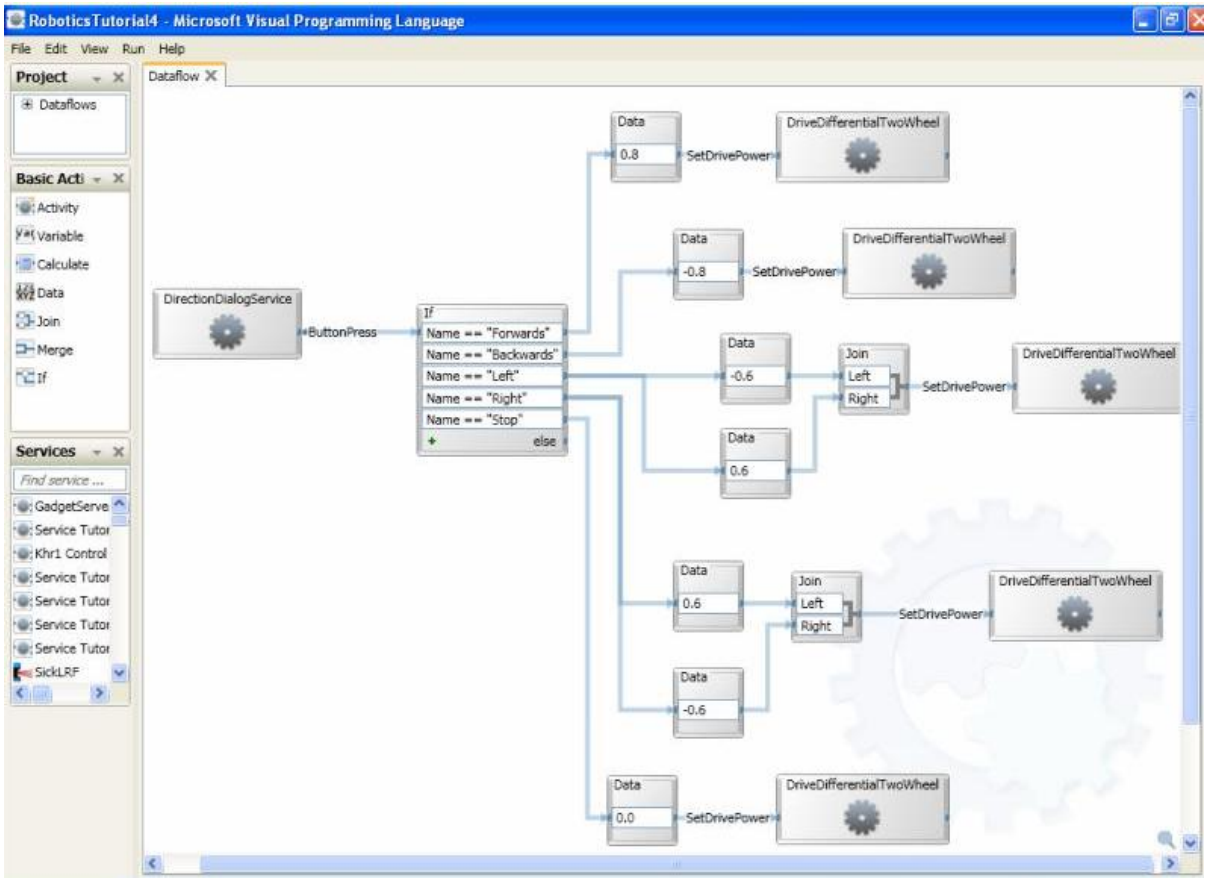

Figure 4.4 MRDS Visual Programming Language

Figure 4.4 shows a simple motor control program. The `DirectionDialogService`, on the left of the figure, sends one of five button press commands which are processed to set the motor drive power for a differential two wheel drive service on the right of the figure.

A service is run to control each individual component of a system. In the case of a robot, a service might control the motors, another service might collect range measurements from an IR sensor and another service could control the navigation system of the robot. MRDS allows these services to subscribe to other services to receive updates about the state of a service or to change the current state of another service.

4.1.1 Concurrency and Coordination Runtime (CRR)

The CCR is a managed library that provides classes and methods to help with concurrency, coordination and failure handling (Johns & Taylor, 2008). It enables the user to design

applications so that the software modules or components can be developed independently, making minimal assumptions about their runtime environment and other components. CCR allows sophisticated robots to do real-time processing such as controlling actuators (motors, arms, pumps) while being able to receive and process sensor data from multiple sensors (IR sensors, odometers, etc). The CCR eliminates the issues of two threads simultaneously attempting to update the same variable and removes the need to program using mutexes (mutual exclusions) which can lead to race conditions that intermittently cause deadlocks. CCR uses its own threading mechanism to prevent these issues which is more efficient than the Windows threading model (Johns & Taylor, 2008).

CCR implementation has three main categories of functionality:

- the *Port*, *PortSet* and *message*,
- *Receivers, Arbiters* and *Handle, and*
- the *Dispatcher*, *DispatcherQueue* and *Tasks.*

Figure 4.5 shows the relationships between each category. When a message is posted to a given Port or PortSet, the message triggers receivers that call for arbiters subscribed to the messaged port to create a task. That task is then queued and dispatched to the threading pool until assigned a thread to be run. Arbiters are used to evaluate the activation conditions that are set on receivers. Activation conditions can be set on receivers to create logical expressions. Two examples of these logical expressions are:

- Join - two messages must arrive on two ports, equivalent of a logical AND.
- Choice - a message can arrive on either of two ports, equivalent of a logical OR.

Ports can be defined as persistent or non- persistent. Persistent ports continuously listen for messages, while non-persistent ports are designed to listen for a single message then close down.

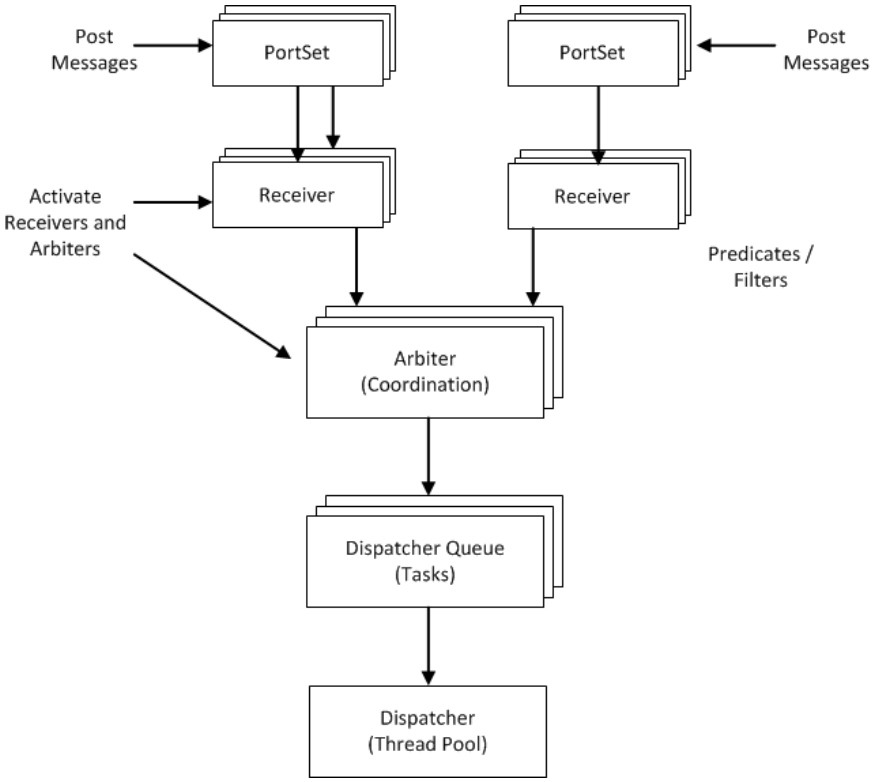

Figure 4.5 CCR architecture (Johns & Taylor, 2008)

The *Port* is the most common primitive of CCR and is used as the point of interaction to send *messages* between two components or services. *Messages* are posted to ports in an asynchronous operation and held in a First-In-First-Out (FIFO) queue (Microsoft, 2010) and remain in the port queue until it is read or de-queued by a *receiver*. *Messages* are just objects of a specified type, so classes can be created and instances of these classes can be sent as messages between services. If *messages* are never removed from the *Port*, then they just keep accumulating which poses a potential memory leak.

The advantage of *Ports* is that messages can be posted to them from any thread. Due to the nature of CCR, posting messages will always be a safe operation. The message will either be processed successfully or will return an error status indicating that it could not be processed. Also, if all the receivers are busy, the message waits until it can be processed, the sender of the message does not have to wait as posting a *message* does not create a block for the sending thread.

PortSet is a generic class that allows the grouping of multiple types of ports. Multiple messages of different types can be posted to a *Portset*. Each message type can have a different handler that executes when a message is received.

The main operations port of a service is usually a *PortSet* containing all the different ports that can receive different types of messages. Figure 4.6 shows the definition of the operations port for a generic service which contains five types of *messages*: Replace, Subscribe, Get, DsspDefaultLookup and DsspDefaultDrop. The latter three are the minimum set of message required for a MRDS service to operate (Johns & Taylor, 2008).

```
//Portset that accepts items of Replace, Get, Subscribe ,
//DsspDefaultLookup and DsspDefaultDrop

Public class GenericServiceOperations: PostSet<Replace, Subscribe,
Get, DsspDefaultLookup,DsspDefaultDrop>{}
```

Figure 4.6 A generic service's operations PortSet

When a message has been received by a port, a *task* is queued to a *dispatcher queue* and then passed onto a *dispatcher* for execution. A *task* is the name given to the thread that executes the incoming message *handler*, which runs in a fully multi-threaded environment. The *dispatcher* takes a *task* from the *dispatcher queue* and allocates a thread to run the task. When threads become available, the dispatcher is automatically queried for another task to run.

Iterators are another key tool that CCR uses to allow sequential execution of code but without blocking the execution thread when it needs to wait for a message. A service controlling a robotic arm may wait for a response message to say that a movement was successful or a fault message indicating that there was a problem. When an operation is performed that will take an unknown about of time to execute, the *iterator* effectively remembers the current location in the code and then relinquishes control until a response message is received. When the response message arrives, the code resumes execution from the point where it left off. This feature allows another thread to execute during the wait time which would have normally locked up the thread.

4.1.2 Decentralized Software Services (DSS)

The Decentralized Software Services (DSS) is responsible for starting and stopping services and managing the flow of messages between services. DSS is composed of several services that load service configurations, manage security, maintain a directory of running services, control access to local files and embedded resources, and provide user interfaces that are accessible using a web browser. DSS uses a protocol called DSS Protocol (DSSP) which is based on the Representational State Transfer (REST) model often used for web development. REST is a style of software architecture for distributed systems such as the World Wide Web and has emerged as the predominant web service design model (Fielding & Taylor, 2005).

A robotic application built with DSS consists of multiple services running independently and in parallel. DSS in combination with CCR allows these multiple services to run in a real time environment. Services built with DDS are mainly (but not limited to) hardware components such as sensors and actuators and software components such as user interfaces and aggregations referring to sensor-fusion and related tasks (Cepedia, Chaimowicz, & Soto, 2010).

DSS allows services to be operating in the same hosting environment, or DSS Node, or distributed over a network, giving flexibility for execution of computationally expensive services in distributed computers (Cepedia, Chaimowicz, & Soto, 2010).

A DSS service consists of seven main components which can be graphically seen in Figure 4.7:

- **Service URI.** The unique key for each service is the Service URI, which refers to the dynamic Universal Resource Identifier (URI) assigned to a created DSS service. The Service URI enables each service to be identified. This is most useful when multiple instances of the same service are running on the same network.
- **Contract Identifier.** The Contract Identifier is created within the code of the service for identifying it from other services, thus creating a globally unique reference. The Contract Identifier is often also the name of the service. When multiple instances of a service are running, each instance will contain the same Contract Identifier but different service URI.

- **Service State.** The Service State carries the current contents of a service. It will contain different information depending on the role of the service. The state of a service controlling a laser range finder will contain information such as distance measurements and angular resolution where as a service controlling a differential drive system will contain information such as current encoder values and current wheel speeds.
- **Service Partners.** Service Partners enable a DSS application to be created by several services providing higher level functions and create more complex applications. The Service Partner definitions connect the services that must communicate and share knowledge about their state.
- **Main Port.** A service's Main Port is a CCR Portset where all messages from external services are received. The Main Port is a private member of a service which can only receive pre-defined messages (defined at service creation) which creates a well-organized infrastructure for coupling distributed services.
- **Service Handlers.** Service Handlers receive messages that arrive on the Main Port, which can come in the form of requested information about the services state or as a notification. The Service Handlers develop specific actions in accordance to the type of message that arrives on the Main Port.
- **Event Notifications.** Event Notifications occur as the result of changes to a service's state. A service that has subscribed to another service and is currently monitoring the service will receive an update message.

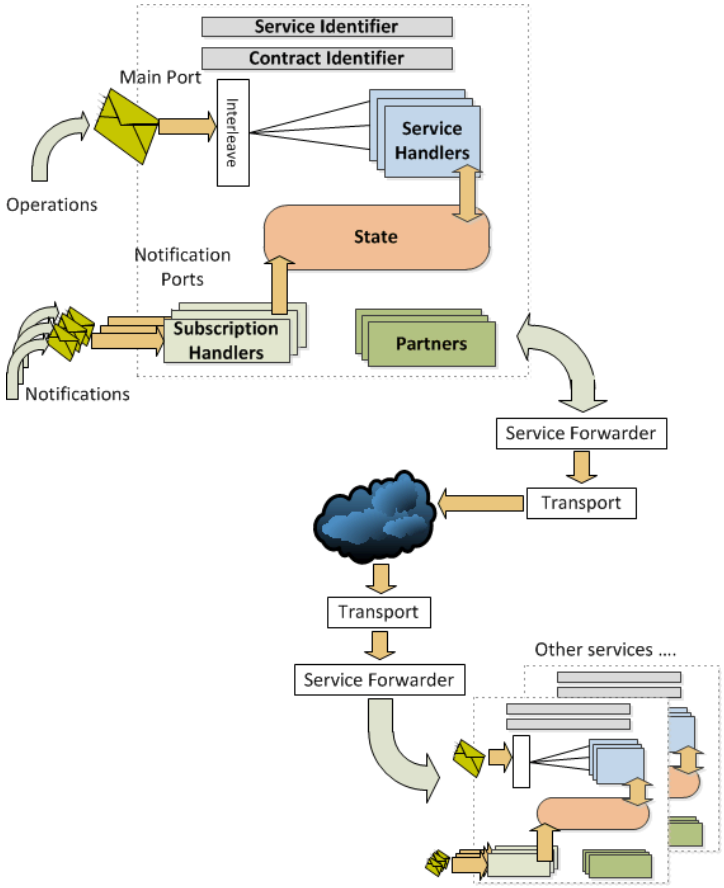

Figure 4.7 DSS architecture (Johns & Taylor, 2008)

As DSS applications can work in a distributed fashion through a network, there is a special port called Service Forwarder, which is responsible for the partnering of services running on remote nodes.

To clarify the differences between the CCR and DSS: the CCR is a programming model for handling multi-threading and inter-task synchronization, whereas DSS is used for building applications based on a coupled service model. Services can run anywhere on the network, so DSS provides a communications infrastructure that enables services to transparently run on different nodes using all of the same CCR constructs that they would use if they were running locally.

By default, MRSD's Security Manager Service does not allow services to be accessed across networked computers. When a DSS node is started with a security settings file specified, the security manager is always started. For this project the security settings were disabled so communication between the host computer and observing computer was not restricted. Figure

4.8 shows the DSS Node Security Configuration file created which this project starts with every DSS service to disable the security settings.

```
<?xml version="1.0" ?>
<SecuritySettings
xmlns="http://schemas.microsoft.com/robotics/2008/02/security.html">
<AuthenticationRequired>false</AuthenticationRequired>
<OnlySignedAssemblies>false</OnlySignedAssemblies>
<Users />
</SecuritySettings>
```

Figure 4.8 DSS node security configuration file

4.2 Programming Environment

The services for this project are built on MRDS 4 version in the .NET 4.0 framework environment. It is therefore necessary to use a .NET language. Examples of .NET languages available include C#, C++, Visual Basic, Python and MRDS's Visual Programming Language (VPL) (Johns & Taylor, 2008). It was decided to program services using C# based on a number of considerations:

- documentation and samples available with MRDS are coded in C#,
- recommended by MRDS as the preferred language for the development of DSS services,
- easy deployment in a distributed environment, and
- efficient memory and processing power requirements.

Microsoft Visual Studio 2010 has been used as the integrated design environment (IDE) to develop services for this project. Visual Studio allows applications to be designed, programmed, debugged and deployed.

Microsoft Visual Studio also allows Graphical User Interfaces (GUI) to be developed using WinForms. WinForms is a mature and simple technology for the purposes of building user interfaces quickly. WinForms will only be visible on the computer that is running the DSS node (Johns & Taylor, 2008). Because of this, a *SegwayServices* DSS service was created to

display a UI WinForm and run on a remote computer to communicate with the main navigation service.

4.3 Summary

The runtime libraries of MRDS, CCR and DSS all contribute to developing the software architecture. CCR provides the ability for segments of code to operate independently within an application. DSS extends CCR concepts by introducing functionality to develop service-oriented applications that can run across a network. Microsoft Visual Studio has been chosen as the IDE for this project and services are written using the C# programming language.

Chapter 5 Navigation Architecture

5.1 Navigation System Overview

A hybrid navigation system that employs an A* path planner and the dynamic window method was developed by a previous student Chris Lee-Johnson (Lee-Johnson, 2004). The system supported differential drive robots with a pre-generated grid map with fixed binary occupancy data being employed for path planning. The system did not have map updating capabilities. Praneel Chand (Chand, 2011) improved upon Lee-Johnson's work creating a hierarchical hybrid navigation system at Victoria University. Chand's work formed an integral part of another thesis created at Victoria University (Talwatta, 2012) which partially implemented the hierarchical hybrid navigation system on the MARVIN robotic platform (McClymont, 2011). The hybrid navigation system created by Chand has been selected as the navigation system for this project.

The localisation section of the navigation algorithm was previously designed for an IR ring that returned 12 distance measurements in a 360° field of view. This project extends Chand's work by using the increased sensor data available with the SICK LMS100 scanner to detect straight lines and distinguish corner and door landmarks within the environment. The landmarks are then compared to a database of known landmarks to update the Segways current position.

Chand's navigation system, depicted in Figure 5.1, consists of three layers:

- The deliberative layer contains the path planner and environment map components indicated by the red dashed lines in Figure 5.1.
- The reactive layer contains the path tracker and the reactive control components indicated by the blue dashed lines in Figure 5.1.
- The third layer contains localisation, information extraction and sensor fusion, and low level motion control.

The hierarchy of the modules of Figure 5.1 provides an indication of the breakdown of control. Modules on the left and right represent perception/representation and action/planning respectively. The indicated update rates have been employed in the respective algorithms on

this project but could be adjusted depending on the requirements of different robotic platforms.

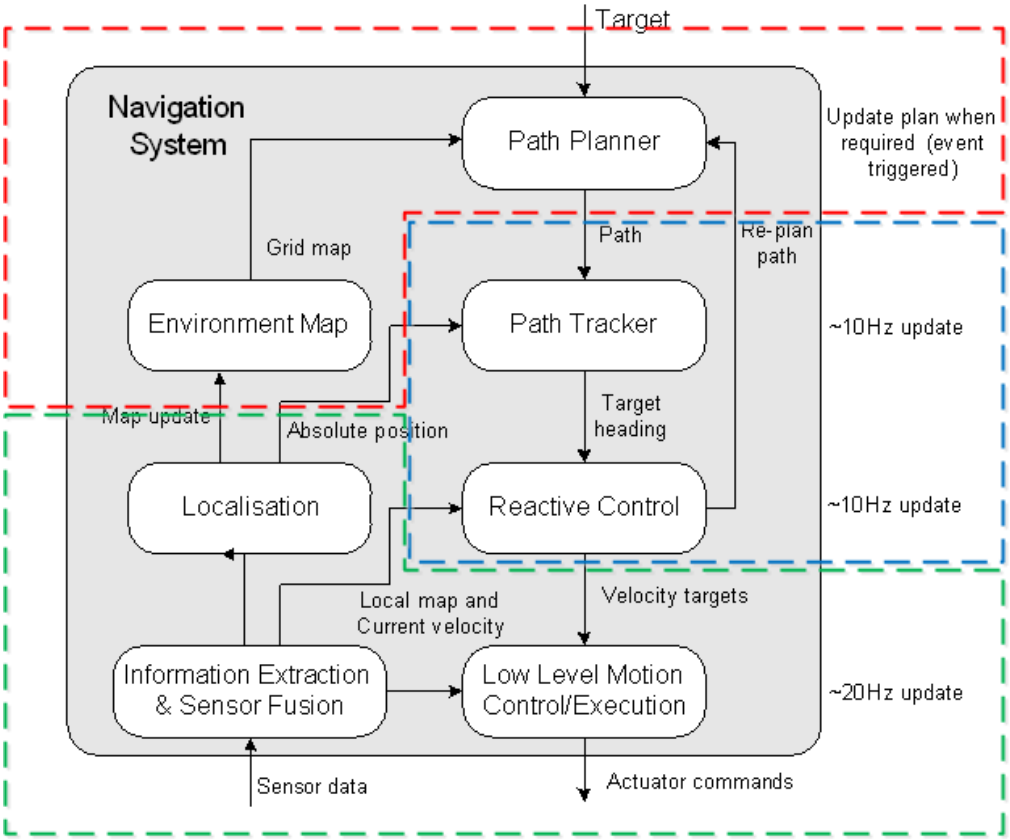

Figure 5.1 Hierarchical hybrid navigation system (Chand & Carnegie, 2011)

5.2 Deliberative Component

The deliberative component of the hierarchical hybrid navigation system bridges the gap between sensing and acting by introducing a planning step. The deliberative architecture enables a robot to perform high level tasks that would be too difficult to perform without planning (Junior, Parikh, & Junior, 2006).

This planning is based on a map of the environment in combination with the environment information acquired by the sensors. An occupancy grid has been selected for the deliberative component of the navigation system to represent the Segway's environment because of its

simplicity and usability in a range of environments. An occupancy grid map is generated by dividing the environment into discreet cells and assigning binary values indicating occupancy.

The Segway operating environment for this project will always be known. Thus, a pre-made environment map can be used by the Segway's navigation system for both navigation and path planning.

5.2.1 Environment Representation

The environment map for this project was constructed from measurements of the third floor corridor of the Laby building at Victoria University. The environment map consists of point co-ordinates and the connection between points such that a wall is represented by two (x, y) coordinates and a connection between point one and point two. The corridor measures 1.75 m × 11.4 m and contains seven doorways and two concave corners for localisation. The representative environment map is illustrated in Figure 5.2.

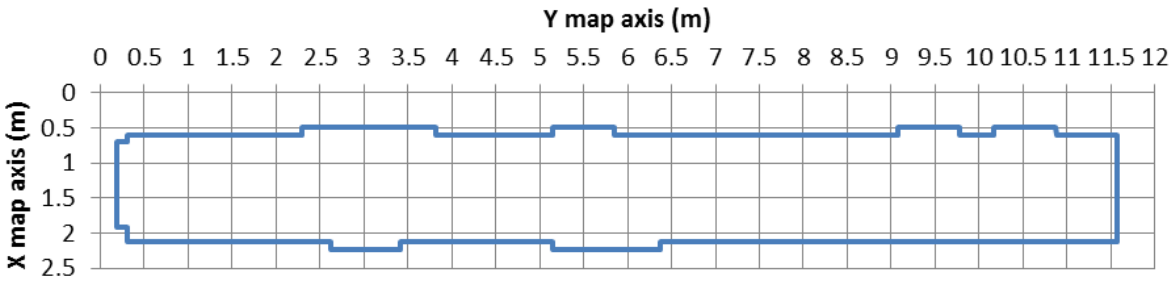

Figure 5.2 Map of Laby corridor

In order to be used as part of the navigation system, the map is converted into a two dimensional array occupancy grid with a "1" depicting a wall and a "0" representing unoccupied space. Figure 5.3 shows the map implemented after the navigation system has converted the map into an occupancy grid. The resolution of the occupancy grid is variable during the conversion from the map points to occupancy grid. For this project, the resolution is set at 0.2 m giving an occupancy grid resolution of 9 × 57 grids. This resolution is considered a good trade-off between an accurate representation of the map environment while keeping computational costs down.

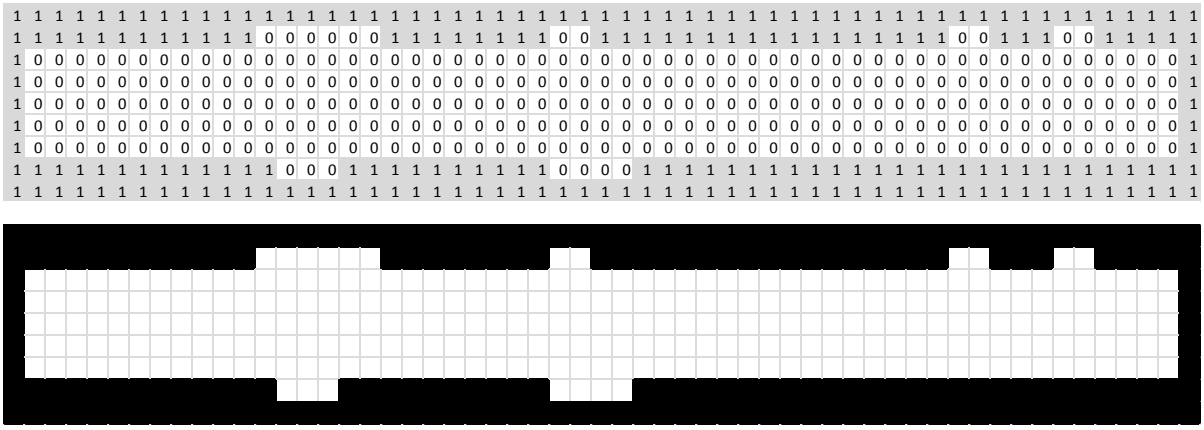

Figure 5.3 Occupancy grid of the Segway's operating environment

5.2.2 Path Planning

A single-tiered modified A* algorithm is used for planning a path through the occupancy grid. The A* algorithm is a best-first heuristic search algorithm that ranks nodes based on the cost of traveling through them (Pearl, 1984). Cost is usually represented by node distances where lower cost values denote a better path to travel. The total cost $f(x)$ of a node x is the sum of two cost values, $g(x)$ and $h(x)$. $g(x)$ represents the cost of travelling from the start node to node x while $h(x)$ is the heuristic cost of travelling from x to the goal node.

$$f(x) = g(x) + h(x) \qquad \textbf{Equation 5.1}$$

The A* algorithm considers binary occupancy values where the nodes are either traversable or non-traversable. Hence $g(x)$ is dependent on the node distance of the lowest cost path from the start node to the parent node x_{par} and the Euclidean distance between x and x_{par}. Heuristic cost $h(x)$ is an over estimate represented by the Euclidean distance from the current node x to the goal node.

If the path planner cannot find an appropriate path to the goal, path planning flags are set to stop the navigation system until an appropriate path can be found. This occurs when either the initial position or target position is located outside of the map or there is no direct path between the two locations. For an appropriate path to be found either the initial position or target position needs to be changed to a valid location.

5.3 Reactive Control Overview

An outline of various reactive control methods has been presented in Chapter 2. The reactive control algorithm selected combines a modified dynamic window (Fox, Burgard, & Thrun, 1997) with a polar histogram technique similar to the vector field histogram method presented by Ulrigh & Borenstein (1998). A simplified block diagram of the two-stage optimisation process that can track a path and avoid obstacles is illustrated in Figure 5.4. A target heading angle is determined from the path tracker which is then used as the input to the direction sensor that produces a modified target heading as an output. The modified target heading angle is then used by the dynamic window to produce linear and angular wheel velocities.

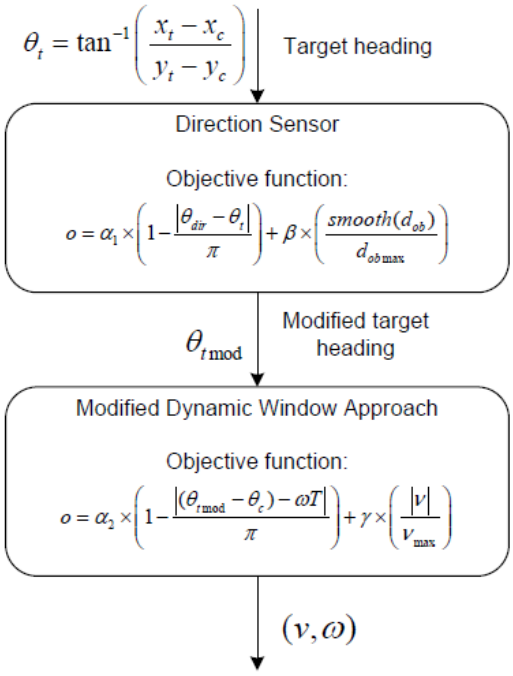

Figure 5.4 Overview of reactive control strategy (Chand, 2011)

5.3.1 Path Tracking

The path tracking algorithm checks the path planning flags to ensure that an appropriate path has been found through the map. If no path has been found, the navigation algorithm sets the angular and linear target velocities to 0 stopping the Segway from moving. The distance from the current position of the Segway (x_c, y_c, θ_c) to each node of the path planner is calculated to find the closest node position to the Segway (x_n, y_n).

When the Segway is following a planned path, (x_g, y_g, θ_g) represents the coordinates of a node that is five nodes ahead of the closest node to the Segway, otherwise, (x_g, y_g, θ_g) represents the final destination coordinates of the Segway. At a resolution of 0.2 metres for the occupancy grid, five nodes represents 1 metre along the planned path for the Segway to head towards. This gives a distance forward of the current Segway position to aim for which continuously moves forwards as the Segway moves and allows room to travel around any obstacles encountered.

The target heading is then calculated using Equation 5.2 which is the main input into the direction sensor algorithm.

$$\theta_t = \tan^{-1}\left(\frac{x_g - x_c}{y_g - y_c}\right) \qquad \textbf{Equation 5.2}$$

5.3.2 Direction Sensor

The direction sensor maximises the objective function that finds an appropriate balance between obstacle avoidance and goal directedness.

A circular shape represents the Segway as shown at the centre of Figure 5.5 with radius r_m. The current position and goal position of the Segway are defined as (x_c, y_c, θ_c) and (x_g, y_g, θ_g) respectively. The target heading angle θ_t is calculated in the path tracking stage of the algorithm (Section 5.3.1) and is used as the input to the direction sensor.

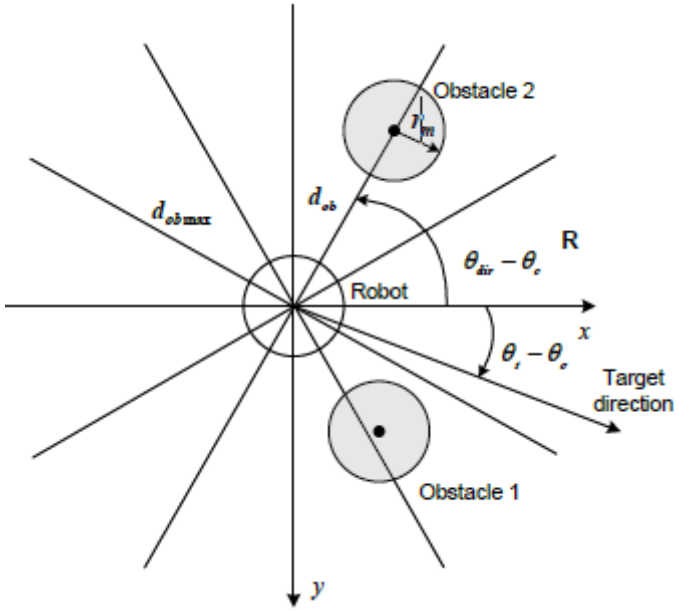

Figure 5.5 Direction sensor representation (Chand & Carnegie, 2011)

To determine the most appropriate direction of travel, the Segway is represented as a point and each obstacle is enlarged by the radius of the Segway. The region surrounding the Segway is then divided into an arbitrary number of lines to represent candidate orientations θ_{dir}. All orientation angles are converted to the Segway's reference frame R by subtracting the current absolute orientation θ_c.

An objective function is applied to each candidate orientation which maximises goal directedness $|\theta_{dir} - \theta_t|$ and distance to obstacles d_{ob}.

$$o_{ds}(\theta_{dir}) = \alpha\left(1 - \frac{|\theta_{dir} - \theta_t|}{\pi}\right) + \beta\left(\frac{d_{ob}}{d_{obmax}}\right) \qquad \textbf{Equation 5.3}$$

Equation 5.3 shows the objective function, where higher values denote a better compromise between goal direction and obstacle avoidance. The maximum obstacle distance, d_{obmax}, is set to the maximum sensing range. α and β are unit interval weighting for goal directness and obstacle clearance respectively which are calculated using trial and error to find an

appropriate balance. Smaller α and larger β values translate to large obstacle advoidance while larger α and smaller β values put preference on heading towards the goal direction over obstacle avoidance which can lead to collisions if obstacles are moving. Different values for α and β were tested for this project and can be found in Section 7.4.1.

5.3.3 Dynamic Window

In the dynamic window approach (Fox, Burgard, & Thrun, 1997) a portion of the velocity space that is achievable within the next control cycle is searched for a velocity pair (v, ω). An overview of the dynamic window method employed in this project is shown in Figure 5.6. The dynamic window approach has seven major inputs:

- A target heading θ_{tmod} is the output from the direction sensor
- The Euclidean distance to the final goal location d_{tgt}
- Current linear and angular velocity (v_c, ω_c)
- Global maximum linear velocity v_{gmax}
- Kinematic constraints
- Dynamic constraints
- Obstacle distances d_{ob}

These inputs limit the maximum and minimum linear and angular velocities used to generate velocity windows. The velocity windows, target heading and obstacles are evaluated with a modified dynamic window objective function to select an optimal velocity pair (v_c, ω_c).

The maximum linear velocity v_{max} is derived from $v_{g\,max}$ and varies depending on goal proximity d_{tgt} and obstacle distances d_{ob}. When the Segway is within deceleration and stopping distances, d_{decel} and d_{stop}, the maximum linear velocity limit v_{lmax} is varied linearly between v_{gmax} and zero (Equation 5.4).

$$V_{lmax}\begin{cases} 0 & \text{If } d_{tgt} \le d_{stop} \\ v_{g\,max} \cdot \dfrac{d_{tgt}}{d_{decel}} & \text{If } d_{stop} < d_{tgt} < d_{decel} \\ v_{g\,max} & \text{otherwise} \end{cases}$$

Equation 5.4

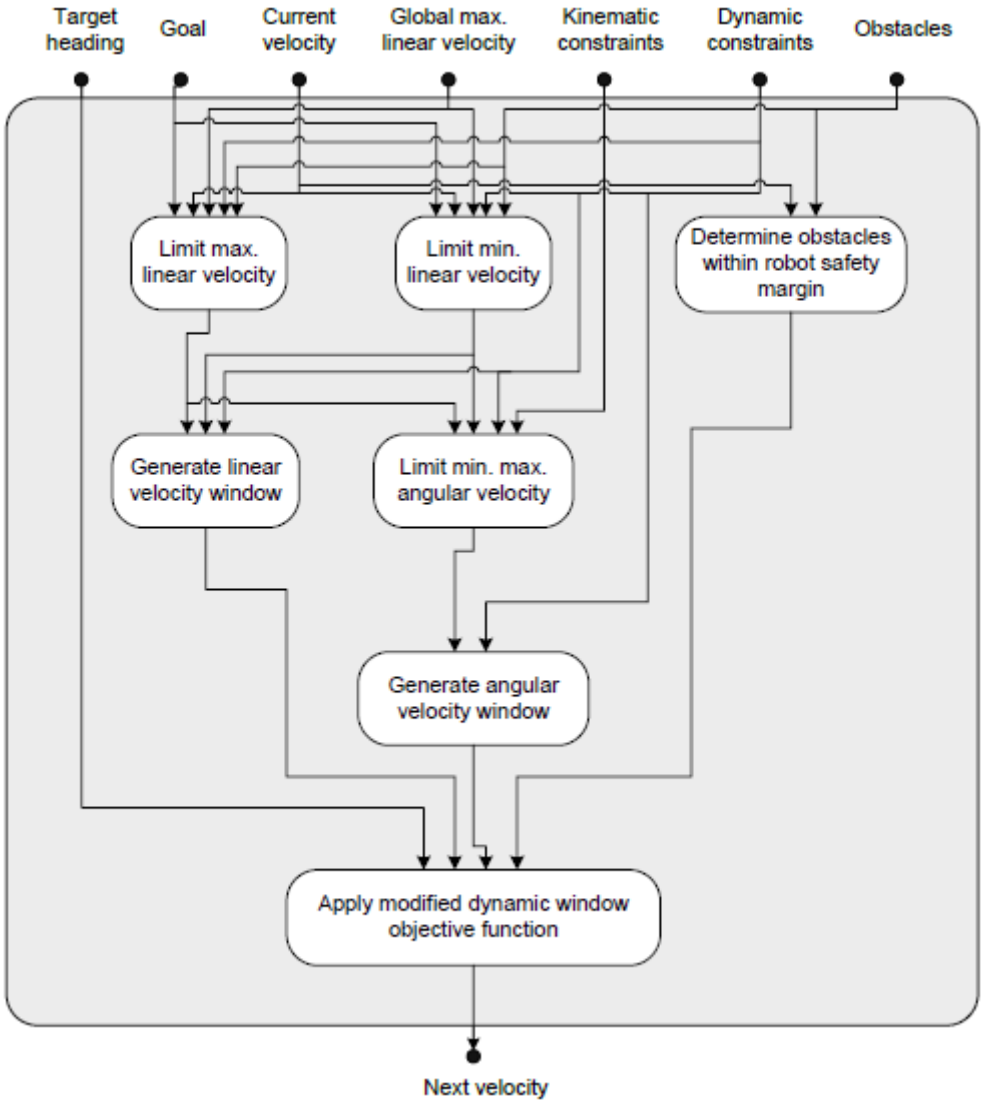

Figure 5.6 Modified dynamic window method overview (Chand & Carnegie, 2011)

The minimum linear velocity for this project is set to zero (Equation 5.5) so that the Segway stops when reaching the goal location.

$$v_{min} = 0$$ **Equation 5.5**

Linear dynamic constraints (linear acceleration l_a and deceleration l_d) and linear velocity limits (v_{min} and v_{max}) are applied to the current velocity, v_c, to produce a linear velocity window $[v_{wmin}, v_{wmax}]$ for the next control cycle. The current linear velocity window is divided into a number of divisions N_{vd} for evaluation.

Angular dynamic constraints (angular acceleration a_a and deceleration a_d) and angular velocity limits (ω_{min} and ω_{max}) are applied to the current angular velocity to produce an angular velocity window $[\omega_{wmin}, \omega_{wmax}]$ for the next control cycle. The current angular velocity window is also divided into a number of divisions N_{wd} for evaluation.

The angular velocity of the Segway has a global maximum ω_{gmax} and a global minimum $-\omega_{gmax}$ representing the Segway turning both clockwise and anti-clockwise. The maximum and minimum curvature, c_{max} and c_{min} respectively, for the next control cycle is derived from the current angular and linear velocity, and dynamic constraints of the Segway. Minimum and maximum angular velocities (Equation 5.6 and Equation 5.7) for the next control cycle are calculated from combinations of c_{min}, c_{max}, v_{min} and v_{max}.

$$\omega_{min} = \min(c_{max}v_{wmax}, c_{min}v_{wmax}, c_{max}v_{wmin}, c_{min}v_{wmin})$$ **Equation 5.6**

$$\omega_{max} = \max(c_{max}v_{wmax}, c_{min}v_{wmax}, c_{max}v_{wmin}, c_{min}v_{wmin})$$ **Equation 5.7**

A safety margin $SM$is added to the Segway's perimeter to allow it to stop before colliding with obstacles. If an obstacle distance d_{ob} is within the safety margin, the velocity window in that direction is rejected. The safety margin has a minimum value of SM_{min} which increases based on the current linear velocity of the Segway platform and a growth factor k_{SM}.

$$SM = SM_{min} + k_{SM} v_c$$ **Equation 5.8**

A flow chart showing the evaluation of each velocity pair (v_n, ω_n) to find the optimal solution is shown in

Figure 5.7. The linear and angular velocity windows are divided into velocity pairs (v_{cd}, ω_{cd}) . The candidate curvature c_{cd} for each velocity pair is calculated and needs to be within $[c_{min}, c_{max}]$ to satisfy differential drive curvature constraints (Chand & Carnegie, 2011).

After curvature constraints have been tested, the distance to collision DC_{cd} if the Segway travels at the candidate linear and angular velocities are determined. Boolean variables ST_v and ST_ω represent the ability for the Segway to successfully stop (Equation 5.9 and Equation 5.10)

$$ST_v = \begin{cases} 1 & if\ DC_{cd} > \dfrac{v_{cd}^2}{2a_a} \\ 0 & otherwise \end{cases}$$ **Equation 5.9**

$$ST_\omega = \begin{cases} 1 & if\ DC_{cd} > \dfrac{\omega_{cd}^2}{2a_{min}} \\ 0 & otherwise \end{cases}$$ **Equation 5.10**

Two objective functions are used depending on whether the Segway can avoid a collision. A primary objective function is calculated (Equation 5.11) if the Segway could avoid a collision. A secondary objective function is calculated (Equation 5.12) for ω_{cd} if the Segway could not stop in time. The secondary objective endeavours to steer the Segway away from the collision target and simultaneously slows forward velocity.

$$obj_{primary}(v_{cd}, \omega_{cd}) = \begin{cases} a_2\left(\frac{|(\theta_{tmod} - \theta_c) - \omega_{cd}|}{\pi}\right) + \gamma\left(\frac{|v_{cd}|}{v_{max}}\right) & \text{If } ST_v \text{ and } DC_{cd} > DC_{min} \\ a_3\left(a_2\left(\frac{|(\theta_{tmod} - \theta_c) - \omega_{cd}|}{\pi}\right) + \gamma\left(\frac{|v_{cd}|}{v_{max}}\right)\right) & \text{If } ST_v \text{ and } DC_{cd} \leq DC_{min} \\ -1 & \text{Otherwise} \end{cases}$$

Equation 5.11

$$obj_{secondary}(v_{cd}, \omega_{cd}) = \begin{cases} a_2\left(\frac{|(\theta_{tmod} - \theta_c) - \omega_{cd}|}{\pi}\right) & \text{If } \overline{ST_v} \text{ and } (\overline{v_c = 0}).(v_{cd} = 0) \\ a_2\left(\frac{|(\theta_{tmod} - \theta_c) - \omega_{cd}|}{\pi}\right) & \text{If } \overline{ST_v} \text{ and } ST_w \text{ and } (v_c = 0).(v_{cd} = 0) \\ -1 & \text{otherwise} \end{cases}$$

Equation 5.12

The values α_2 and γ are weightings for goal directedness and velocity respectively. Smaller values of α_2 result in the Segway only having relatively small changes in direction, which may not be optimal for goal achievement, while large values of α_2 may compromise obstacle avoidance as the platform may not deviate from the target direction sufficiently to avoid the obstacle. Smaller values of γ ensure the Segway moves relatively slowly, while larger values of γ may compromise obstacle avoidance due to traveling at too high a velocity. A small a_3 reduces the objective function output when the collision distance is below an allowed threshold.

All of the velocity pairs are checked against the objective function for the velocity pair with the maximum primary objective value. When a valid angular and linear velocity pair is found, they are set as the target angular and linear velocity for the next control cycle. If a valid velocity pair is not found then the linear velocity target that opposes the current linear velocity is chosen to avoid collisions in the current direction of movement.

When both the primary and secondary objective functions return invalid results, the angular and linear velocity targets are set to oppose the current motion to stop the Segway. This is a rare case that could occur when dynamic obstacles, such as moving people, crowd the Segway and no valid direction allows a valid solution. The Segway's control algorithm would keep the linear and angular velocities at zero until a valid direction and velocity is found.

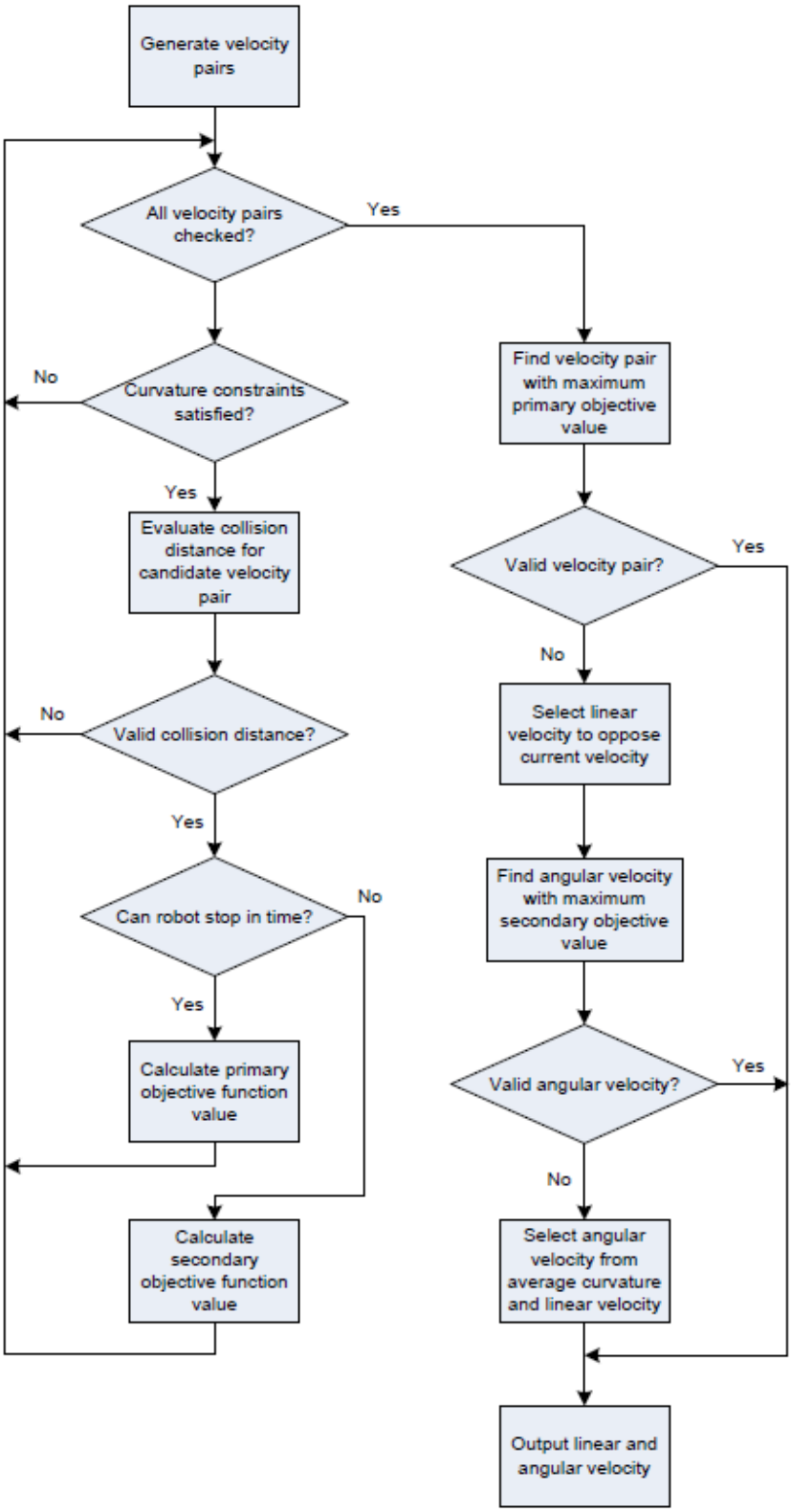

Figure 5.7 Optimal velocity pair selection flowchart (Chand & Carnegie, 2011)

5.4 Internal Representation

For data from range finders and odometers to be combined effectively they first need to be converted into an internal representation that is shared by all sensors. The position of the Segway is given as Cartesian coordinates (x, y) in metres, while its heading is defined as an angle θ in radians.

5.4.1 Odometers

The Segway monitoring messages (Table 3.1) sent from the Segway unit to the host computer contain four odometer counts: integrated left wheel position, integrated right wheel position, integrated fore/aft position and integrated turn position. The Segway's position can be calculated using the measurements from the encoders. Using odometry alone becomes challenging due to the accumulating errors that are inherent in odometry measurements. The accuracy of odometry measurements decreases over time due to limiting factors such as wheel slippage, missed encoder counts and transmission slop (Victorino, Rives, & Borrelly, 2000). Overtime these factors cause an increase in the difference between the actual distance the Segway has travelled and the distance readings from the Segway's odometry.

The Segway interface guide (Segway Inc., 2009) contains a data conversion table for data items contained in monitoring messages (Table 3.5). The table contains estimates which are based on the nominal rolling diameter of the wheel, 48 cm, and deviations can occur with changes in tyre pressure, tyre wear and payload. Based on the table, the expected conversion factors for three of the odometers (integrated left wheel position, integrated right wheel position and integrated fore/aft position) are 33215 counts per metre and 112644 counts per revolution for the integrated turn position. As these were only approximates, more accurate conversion factors were found and are discussed further in Section 7.2.1. The experiment yielded results of 34337 and 116711 for left, right and fore/aft positions and turn position respectively, mentioned here for reference.

The integrated fore/aft encoder count and turn position encoder count are not from physical encoders but rather are calculated by the control software within the Segway. The control

software assumes a wheel diameter of 48 cm and wheel separation of 53 cm (Segway Inc, 2012).

5.4.2 Position and Orientation

The Segway is a two wheel differential drive system. Assuming minimal wheel slippage, each wheel movement results in a change in the Segway's position and/or heading. If both wheels rotate the same distance at the same velocity, the Segway travels in a straight line. If both wheels rotate the same distance but in opposite velocity, a zero radius turn of the Segway occurs. Any combination of these two motions will result in a moving turn.

Integrated left and right wheel position counts can be used to calculate the arc length travelled (in metres) of the left and right wheels respectively using Equation 5.13 and Equation 5.14. The integrated fore/aft position encoder count is used to calculate the arc length travelled (l_c in metres) by the centre of the Segway using Equation 5.15. The integrated turn position encoder count can be used to calculate the angle the Segway's centre has travelled through (φ in radians) using Equation 5.16.

$$left\ \ arc\ length = \frac{left\ encoder\ count}{conversion\ factor} \qquad \textbf{Equation 5.13}$$

$$right\ \ arc\ length = \frac{right\ encoder\ count}{conversion\ factor} \qquad \textbf{Equation 5.14}$$

$$l_c = \frac{fore/aft\ encoder\ count}{conversion\ factor} \qquad \textbf{Equation 5.15}$$

$$\varphi = \frac{turn\ encoder\ count}{conversion\ factor} * 2\pi \qquad \textbf{Equation 5.16}$$

where the $conversion\ factor$ equates to the relative counts per metre/revolution mentioned in Section 5.4.1.

Using results from Equation 5.15 and Equation 5.16, the linear distance travelled by the Segway's centre (D_m in metres) is calculated using Equation 5.17.

$$D_m = \frac{l_c\sqrt{2(1-\cos\varphi}}{\varphi} \qquad \textbf{Equation 5.17}$$

Finally, the calculated distance travelled D_m and angle turned φ are converted into a set of Cartesian co-ordinates representing the change in position $(\Delta x, \Delta y, \varphi)$ which are added to the current position and orientation (x_c, y_c, θ_c) of the Segway (Equation 5.18, 5.19 and 5.20).

$$\Delta\theta = \varphi \qquad \textbf{Equation 5.18}$$

$$\Delta x = D_m \cos\theta \qquad \textbf{Equation 5.19}$$

$$\Delta y = D_m \sin\theta \qquad \textbf{Equation 5.20}$$

In the coordinate system, X position represents lateral motion with positive values to the right and negative values to the left. Y position represents forward motion as positive values and reverse motion as negative values. The heading θ represents the heading of the Segway in radians, where zero change results in movement in a straight line, positive values in a clockwise rotation and negative values in an anti-clockwise direction.

5.5 Localisation

This section addresses the methods used in this project to discover landmarks. The SICK LMS100 range finder data is used along with odometry data for localisation of the Segway. Lines are first extracted from the rangefinder dataset. Relationships between the extracted lines are used to discover the Cartesian coordinate location of landmarks. Discovered

landmarks are then compared to landmarks extracted from the map of the environment in Section 5.2.1.

Landmarks are geometric objects that can be recognized each time they are encountered again. Some specific properties of landmarks are important: they should be re-observable, distinguishable from each other and stationary. Furthermore a critical number of landmarks is required for localisation. (Riisgaard, 2005)

5.5.1 Line Extraction

The SICK LMS100 laser range finder produces a 2D representation of the environment. Points from a range scan are specified in polar coordinates (r, θ) whose origin is the current position of the Segway (x_c, y_c, θ_c). The polar representation of the scan is converted to Cartesian coordinates (x_{laser}, y_{laser}) using Equation 5.21 and Equation 5.22. A visual representation of this relationship is depicted in Figure 5.8.

$$x_{laser} = x_c + r\cos(\theta + \theta_c) \qquad \textbf{Equation 5.21}$$

$$y_{laser} = y_c + r\sin(\theta + \theta_c) \qquad \textbf{Equation 5.22}$$

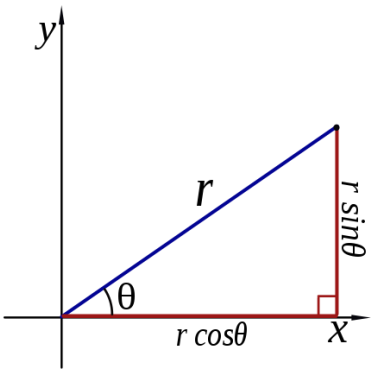

Figure 5.8 Relationship between Polar and Cartesian Coordinates

There are three main problems in line extraction in indoor environments (Forsyth & Ponce, 2002) . They are:

- How many lines are there?
- Which points belong to which line?
- Given the points that belong to a line, how to estimate the line model parameters?

Two line extraction methods inspired by Nguygen, Martinelli, Tomastis & Siegwart (2005) were investigated for the purpose of finding landmarks for localization. The two line extraction methods were 'Split and Merge' (Castellanos & Tadoos, 1996) (Borges & Aldon, 2000) and Random Sample Consensus (RANSAC) (Fischler & Bolles, 1981) (Riisgaard & Blas, 2005). These two methods were chosen based on their performance and popularity in mobile robotics, particularly for their feature extraction capabilities.

Split and Merge

The Split and Merge, also known as the Ramer-Douglas-Peuker algorithm (Liu, Jin, Cui, & Wang, 2001), is the first algorithm investigated. Pseudo code for the Split and Merge algorithm is shown in Figure 5.9.

Algorithm 1: *Split-and-Merge*

1 Initial: set s_1 consists of N points. Put s_1 in a list $\mathcal{L}$
2 Fit a line to the next set s_i in $\mathcal{L}$
3 Detect point P with maximum distance d_P to the line
4 If d_P is less than a threshold, continue (go to 2)
5 Otherwise, split s_i at P into s_{i1} and s_{i2}, replace s_i in $\mathcal{L}$ by s_{i1} and s_{i2}, continue (go to 2)
6 When all sets (segments) in $\mathcal{L}$ have been checked, merge collinear segments.

Figure 5.9 Split and Merge pseudo code (Nguyen, Martinelli, Tomatis, & Siegwart, 2005)

An example of the Split and Merge algorithm is shown in Figure 5.10. The original line consists of 10 points marked P_0 to P_9. The first (P_0) and last (P_9) points are connected with a straight line and the point with the greatest perpendicular distance to the line is found (P_5). If the selected point (P_5) is greater than the allowed distance from the line, the original line is split into two lines with P_5 being the splitting point as shown in Example C in Figure 5.10. The process is recursively repeated until the greatest perpendicular distance is less than the allowed distance to the line. P_0, P_5, P_8 and P_9 are chained to produce a simplified line as shown in Example D in Figure 5.10.

The Split and Merge algorithm has a complexity of $O(n^2)$ and is less complex relative to other line extraction methods (Nguyen, Martinelli, Tomatis, & Siegwart, 2005).

A slight adjustment to the algorithm was made during testing to account for noisy data. The adjustment required at least two points to be further than the allowed distance to the line before the data was split. This allowed longer lines to be found when a single data point was an outlier to a line. Thresholds are set so that at least 7 points are required in order to be considered as a line and each point must be within 5 cm of the found line. The minimum line length is 20 cm to avoid many short or false positive lines being found that would not associate to landmarks. Landmarks are described in Section 5.5.2.

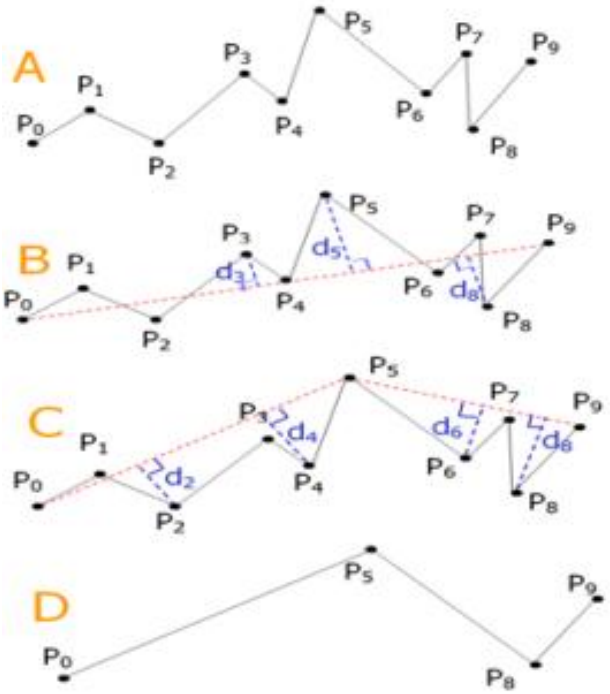

Figure 5.10 Split and Merge algorithm

RANSAC

RANSAC or Random Sample Consensus is another algorithm which can be used to extract lines from the SICK LMS100 laser scan. RANSAC finds lines by randomly taking a sample of the laser readings and then uses a least squares approximation to find the best fit line that runs through the selected readings. Once this is done, RANSAC checks how many laser

readings lie close to the best fit line. If the number of close points is above a pre-determined threshold then a line has been found.

The RANSAC algorithm presented by Riisgaard & Blas (2005) has been selected for testing. The algorithm assumes that the laser data readings are converted to Cartesian coordinates. Pseudo code for the algorithm is shown in Figure 5.11.

```
While

   • there are still un-associated laser readings,
   • and the number of readings is larger than the consensus,
   • and we have done less than N trials.

Do

   ➢ Select a random laser data reading.
   ➢ Randomly sample S data readings
   ➢ Using these S samples and the original reading, calculate a
     least squares best fit line.
   ➢ Determine how many laser data readings lie within X
     centimetres of this best fit line.
   ➢ If the number of laser data reading on the line is above
     some consensus C, do the following:
        o Calculate the least squares best fit line based on all
          the laser readings determined to lie on the old best
          fit line
        o Add this best fit line to the lines we have extracted
        o Remove the number of readings lying on the line from
          the total set of un-associated readings.

N – Max number of times to attempt to find lines.

S – Number of samples to compute initial line.

X – Max distance a reading may be from line to get associated to
line.

C – Number of points that must lie on a line for it to be taken
as a line.
```

Figure 5.11 Pseudo code for RANSAC algorithm (Riisgaard & Blas, 2005)

For this project the RANSAC parameters were set as follows:

N - 1000

S - 10

X - 5 cm

C - 7

The parameters were chosen based on experimental tuning so that the best performance was obtained. The RANSAC algorithm has a complexity of $S \times N \times N.Trials$ where S is the number of line segments extracted, N is the number of points in the scan and $N.Trials$ is the number of trials for RANSAC.

The two algorithms were tested and compared for speed, correctness and precision. Both algorithms were able to correctly identify the major lines within the scanned dataset and had few false positives once the specific parameters were tuned for the corridor environment. The major difference between the two algorithms was the completion speed. The time taken between starting and ending each algorithm was calculated and used to determine the maximum frequency that each algorithm could be continuously run at. Split and Merge performed faster than RANSAC with an average continuous running frequency of approximately 2000 Hz compared to an average continuous running frequency of approximately 150 Hz. The performance difference is mainly because RANSAC is based on non-deterministic methods whereas Split and Merge makes use of sequencing characteristics of the raw data points.

Split and Merge was chosen as the line extraction method for this project as its performance speed was faster while the correctness and precision was comparable to RANSAC. The increased complexity of RANSAC did not warrant its use for the desired environment.

When a line is found, the equation of the line between the two end points (in the format of $y = mx + c$) is calculated using Equation 5.23 and Equation 5.24. The gradient of the line is used in the landmark detection algorithm.

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$ **Equation 5.23**

$$c = y_1 - mx_1$$ **Equation 5.24**

If $x_2 - x_1 = 0$, then m is set to 100,000.

After finding all the lines within the laser data, landmarks are located and associated to known landmarks to localise the Segway.

5.5.2 Landmark Detection and Association

As mentioned, landmarks are used for updating the position of the Segway and correcting for any errors that occur over time in the odometry (Bailey, Beckler, Hoglund, & Saxton, 2008). The landmark detection algorithm locates three different types of landmarks. These landmarks are door frames, concave corners and convex corners as shown in Figure 5.12. The algorithm takes an input of an array of lines from the line extraction method and outputs an array of landmarks. Found landmarks contain the (x, y) coordinate position of the extracted landmark, the two lines which make up the landmark and two Boolean values. The first Boolean denotes whether the landmark is a door or a corner while the second Boolean denotes if the corner is convex or concave. The second Boolean is ignored if a door is found.

Corners are found by looking for perpendicular lines with nearby end points as shown in Figure 5.13. For the purposes of landmark extraction, perpendicular lines are defined by two lines which gradients differ by 90 ± 10 degrees (Equation 5.24). The ± 10 degrees allows for inaccuracies for long lines in the line extraction process. To be a corner, the two end points are required to be within 15 cm of eachother. The landmark coordinates (x, y) is the intersection of the two lines which make up the corner.

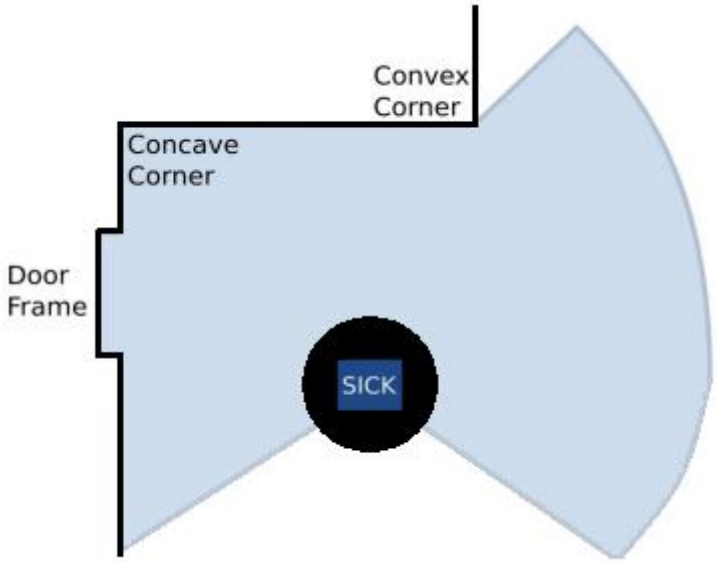

Figure 5.12 Landmarks found in indoor environments

$$\theta_{diff} = \tan^{-1}\left|\frac{m_1 - m_2}{1 + m_1 m_2}\right| \qquad \textbf{Equation 5.25}$$

Checks are done beforehad so that if $m_1 m_2 = -1$, θ_{diff} = 90 °.

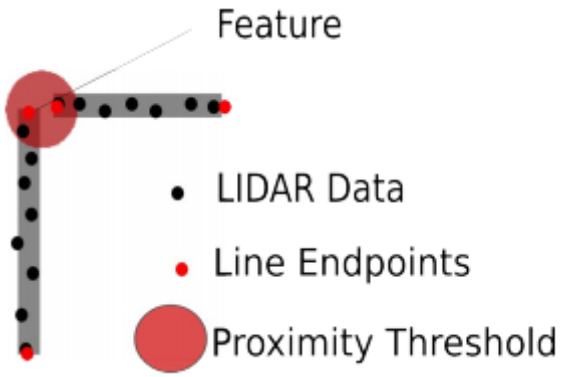

Figure 5.13 Corner landmark

When a corner feature is found, more analysis is needed to know if the corner is convex or concave. This is done by calculating three distances: the distances between the landmark's corner point and the Segway, and distances between the Segway and each of the endpoints of

the two lines that make the landmark. If the distance between the Segway and the corner point is less than the distance to the line ends then the landmark is a concave corner. If the distance between the Segway and the corner point is greater than the distance to the line endpoints then the landmark is a convex corner. Figure 5.14 shows the Segway detecting a corner landmark.

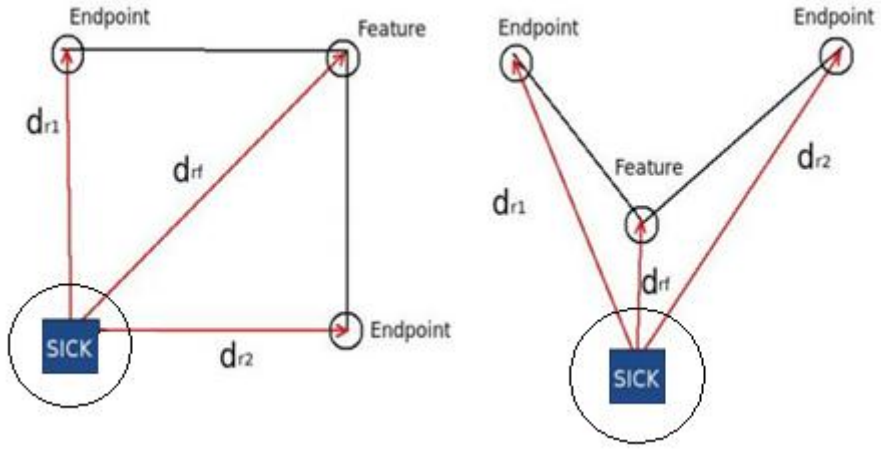

Figure 5.14 Left: convex corner. Right: concave corner.

Door features are found by looking for parallel lines that have nearby endpoints as shown in Figure 5.15. Parallel lines are defined by two lines whose gradients differ by ± 10 degrees (Equation 5.24). The ± 10 degrees allows for inaccuracies in the line extraction. To be a door frame, the two line endpoints are required to be separated by more than 7.5 cm but less than 20 cm. The landmark coordinates (x, y) is the centre point between the two line endpoints.

The landmark detection algorithm is run a single time after the environment map is loaded. The algorithm is run using the map lines described in Section 5.2.1 to create a database of known landmarks within the environment. It is assumed that the Segway is positioned in the centre of the map for convex/concave corner evaluation.

The line extraction and landmark extraction algorithms are run on a 20 Hz sensor timer and scanned landmarks are compared to the database of known landmarks for association.

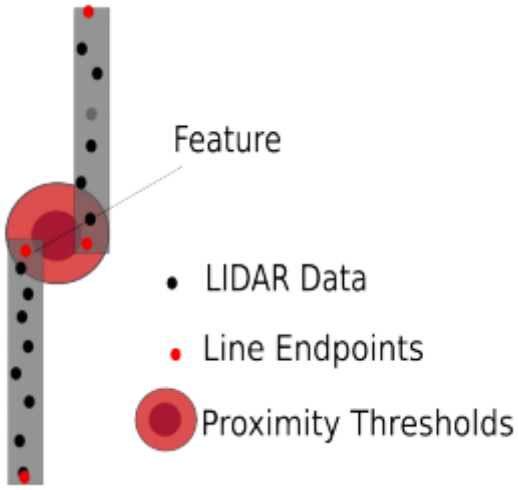

Figure 5.15 Door landmark

The technique used for association is called the nearest neighbour approach as a scanned landmark is associated with the nearest landmark in the database. The simplest way to calculate the nearest landmark is to determine the Euclidean distance. Another method that could have been used is the Mahalanobis distance (Blanco, Gonzalez, & Fernandez, 2012) which is superior but more complicated. The Mahalanobis distance differs from the Euclidean distance in that it takes the correlations of the dataset into account during calculations. The Euclidean distance was preferred as the landmarks for this project are far enough apart to make using the Mahalanobis distance an unnecessary complication.

The distance between each scanned landmark from the SICK LMS100 laser scanner and the database of landmarks is calculated and the closest landmark in the database is found. If the distance between the closest landmarks in the database is less than 20 cm, the landmarks are considered to be associated. If a scanned landmark cannot be associated to a landmark in the database, it is removed from the list of scanned landmarks.

5.5.3 Landmark Position Error

The error in position (x_c, y_c, θ) of the Segway can be calculated once all scanned landmarks have been associated to a landmark in the database. The error in position for each scanned landmark is calculated by comparing the (x, y) position of the found landmark to the expected (x, y) position of the landmark in the database as seen in Figure 5.16. This is achieved by

using Equation 5.26 and Equation 5.27. Averaging the error in position of each scanned landmark yields a single average error in the position of the Segway $(x_{avg\ err}, y_{avg\ err})$. This error is combined with odometry and used to update the position of the Segway.

$$x_{err} = x_{expected} - x_{found} \qquad \textbf{Equation 5.26}$$

$$y_{err} = y_{expected} - y_{found} \qquad \textbf{Equation 5.27}$$

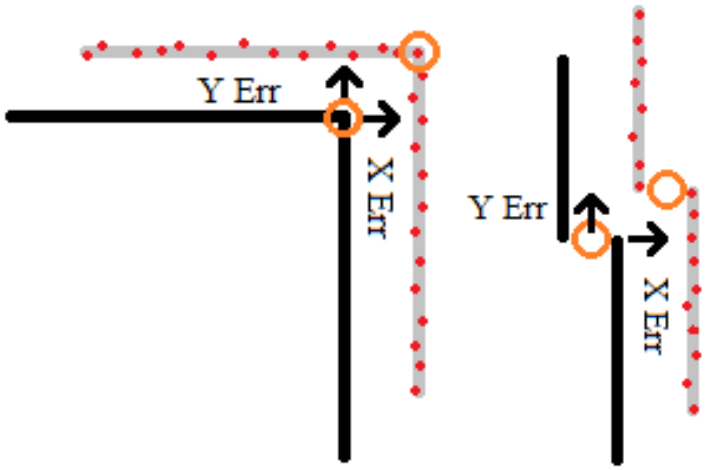

Figure 5.16 Position error example. Left: corner. Right: doorway.

Figure 5.16 shows an example of an error in the (x, y) position of the database landmark (black lines) and the (x, y) position of the scanned landmark (grey lines) for both corner landmarks (left) and door landmarks (right).

Error in heading of the Segway is determined by calculating the angle between the two lines that make up the landmark (Equation 5.24). Each landmark in both the map database and scanned list has two associated lines as seen in Figure 5.17 with two heading errors associated to them. Averaging the heading errors over all scanned landmarks gives an average heading error $\theta_{avg\ err}$. This error is combined with odometry and used to update the position of the Segway.

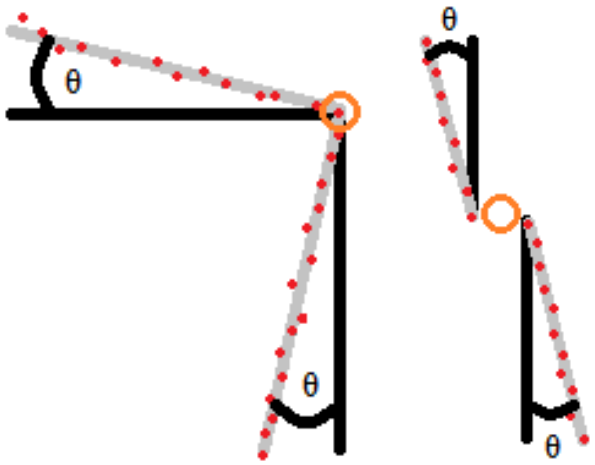

Figure 5.17 Heading error example. Left: corner. Right: doorway.

5.6 Sensor Fusion

The Segway's various odometers and sensors and the SICK LMS100 laser range finder provide useful data, but their individual importance varies with circumstance. For example, odometers are relatively accurate over short distances, but cumulative errors which are generated over distance limit their long-term usefulness. Rangefinders can be less accurate but their error is constant over time. The navigation algorithm for the Segway minimises these problems by utilising sensor redundancy. This allows multiple sensors to provide the same information (current position) but with different degrees of accuracy and precision. These two pieces of overlapping information are fused to take advantage of each sensor's strengths and reduce their weaknesses.

Although this Section concentrates on the fusion of overlapping data from different sensors, the term *sensor fusion* has a broader meaning that encompasses non-redundant sensor signals and multiple samples from a single sensor (Sauer, Brugger, Hofer, & Tibken, 2001).

Due to the small number of sensors on the Segway and the simple corridor operating environment, a Dynamic Weighted Average algorithm (Kapach, Giorini, & Mylopoulos, 2007) was chosen for sensor fusion. Other sensor fusion algorithms investigated for this project include Bayesian inference (Williams, Wilson, & Hancock, 1997), Dempster-Shafter Inference (Wu, Seigel, Stiefelhagen, & Yang, 2002), Fuzzy Logic (Godjevac, 1995) and Neural Network (van Dam, Krose, & Groen, 1996) algorithms. These algorithms would not

provide improvement enough to justify the complexity of their implementation and increase of CPU consumption.

The Dynamic Weighted Average algorithm allows each of the sensors to make a contribution towards the estimation of the current position of the Segway. The Segway's odometer weights would be much higher than the SICK LMS100 range finder, given the higher accuracy over short distances. Lower weightings are given to the range finder measurements to correct odometer errors over time.

5.7 Summary

This chapter gives a detailed explanation of the hybrid navigation system used for autonomous indoor navigation for the Segway platform. The hybrid navigation system is composed of three layers: a deliberative layer, a reactive layer and a third layer containing localisation, information extraction and sensor fusion. The deliberative component of the navigation system comprises the environment map and an A* path planner. The reactive component of the navigation system comprises a path tracker to follow the planned path, a direction sensor to avoid obstacles not represented in the environment map and a dynamic window algorithm to select the angular and linear velocity to travel for the next control cycle. Localisation of the system uses odometry from the Segway and landmark features extracted from the SICK LMS100 laser range finder. Landmark features extracted include concave and convex corners as well as doorways which are commonly found within the operating environment. Lines are used to make up landmarks and are extracted from the laser range finder data using a split and marge algorithm. Fusion of the odometry and landmark information is done using a dynamic weighted average algorithm.

Chapter 6 Software

This chapter covers the software used to implement the hybrid navigation system described in Chapter 5. An overview of the software architecture is presented as well as the functional model. The implemented software services and the interaction between each service are then described. Each service in this section is designed to be modular with reusability a goal for the software.

6.1 Segway Software Architecture

MRDS uses a SOA as the software framework to implement services. SOA is an event driven programming approach that is mostly applied to web based applications (Chen Y. , 2008).

SOA have been perceived to be less efficient than the typical Object-Oriented Computing (OOC) methods because of the extra layer of standard interface which allows SOA applications to be language and platform independent while still allowing communication (Chen Y. , 2008). SOA applications are not limited to being implemented over the Web and remote services can run on any suitable locally networked machine. SOA have benefits in robotic applications particularly for the following reasons (Chen Y. , 2006):

- Robotic systems can have limited memory capacity to carry programs for all situations, the SOA allows complex services to run on remote nodes.
- Faults can occur and on-site repair is not always available.
- Users can stop and modify individual services without stopping the whole system.
- SOA applications are independent of devices that the application communicates with allowing the same application to be applied to different robotic devices.

The services implemented to control the Segway platform using the hybrid navigation architecture are developed using the SOA model. The hybrid navigation framework consists of a three tiered system shown in Figure 6.1. The bottom tier is the hardware interfaces which consists of the `SickLRF_Scanner` service (Section 6.3) and the `SegwayBase` service (Section 6.4). These services send and receive control messages to and from the SICK

LMS100 and the Segway platform respectively. The middle tier is the application layer which consists of the `SegwayNavigation` service (Section 6.5) which implements the hybrid navigation algorithm discussed in Chapter 5. The top tier is the user interface layer which consists of the `SegwayServices` UI service (Section 6.6) which allows user control of the system from a remote computer.

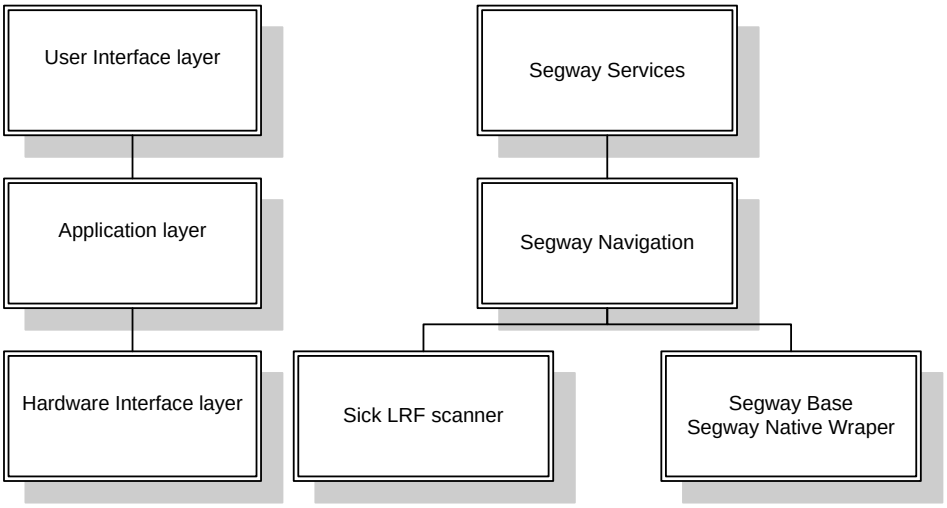

Figure 6.1 Overview of the software architecture

6.2 Operating Mode

The Segway software has two operating modes: a manual mode where the Segway responds to inputs from a keyboard or joystick and an autonomous mode where the Segway moves from one location to another while avoiding obstacles. The Segway software starts in manual mode and changes to the operating mode can be selected using the user interface.

6.2.1 Manual

Manual mode allows a user to directly control the Segway platform's movements. The navigation system starts in manual mode until commanded to go autonomous via the user interface.

In manual mode, the Segway platform has three abilities to move. The first ability is the drive distance command that is implemented using the generic differential drive contract to drive a specified distance in a straight line. The second ability is the rotate angle command, also implemented using the differential drive contract, to rotate the Segway platform by a specified angle in degrees. The third ability to control the Segway is through the use of a joystick (or any controller that conforms to the Game Controller contract in MRDS).

Manual mode allows the user to set the motor drive power and thus the speed that the Segway moves when using the three methods mentioned above. The Segway platform can also be commanded to change between tractor and balance mode or be turned off from the manual control options.

6.2.2 Autonomous

Autonomous mode allows the start of the hybrid navigation algorithm. Following the environment map upload, the user interface sets the initial and target position coordinates. If a path can be found from the initial position to the target position, the autonomous mode can start and move along the planned path. Details of the autonomous mode are further covered in Sections 6.5 and 6.6.

6.3 SickLRF_Scanner Service

The `SickLRF_Scanner` service is a DSS node service that communicates, controls and obtains range finder data from the SICK LMS100 laser scanner. This service falls under the 'Hardware Interface Layer' in Figure 6.1. There was no existing generic driver service on MRDS that provided the functionality to interface with the SICK scanner but there was a generic driver for the similar SICK LMS200 laser scanner. The SICK LMS200 service communicates over RS-232 while this project communicates with the SICK LMS100 over a TCP/IP Ethernet connection so communication with the scanner had to be designed from the ground up.

The main tasks of this service are receiving scan data packets from the SICK LMS100 scanner and notifying any subscribing services when its state has been updated. The `SickLRF_Scanner` service's state updates every time a data packet has been received.

A flow chart of tasks the `SickLRF_Scanner` service carries out is in Figure 6.2. After the service is started it connects to the SICK LRF100 using a TCP/IP Ethernet connection. If the connection is successful, receivers are activated to listen for packets from the scanner as well as commands from any subscribers. There are three commands subscribers can issue to the `SickLRF_Scanner` service. They are start continuous read, stop continuous read and start single read. These three commands send requests to the scanner to send a single measurement reading, continuously send measurements or stop continuously sending measurements. When data packets are received, the appropriate state variables are updated and a notification sent to subscribers.

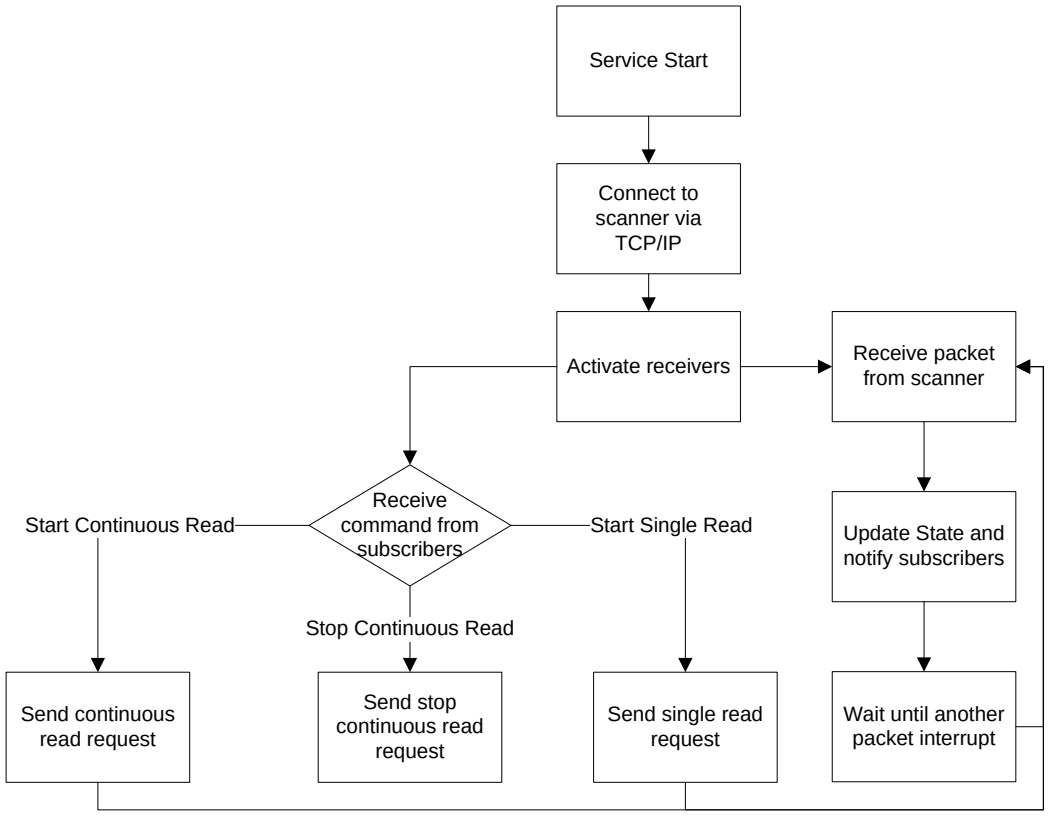

Figure 6.2 Flowchart for SickLRF_Scanner service

The `SickLRF_Scanner` scanner service has been designed to be a generic service that can be re-used by any robot using the MRDS runtime with a SICK LMS100 laser scanner. The service runs on the local host and found at port 50000.

The SickLRF_scanner service consists of three classes: the `SickLRF_ScannerService` class, the `TCPIOManager` class and the `Packet` class.

6.3.1 SickLRF_Scanner Service Class

`SickLRF_Scanner` service controls the SICK LMS100 scanner. When a service is created, the `Start()` method (Figure 6.3) is automatically called.

```
protected override void Start()
{
       _state = new SickLRF_ScannerState();
       _state.IPAddress = "130.195.162.58";
       _state.port = 2111;
       StartLRF( _state.IPAddress, _state.port);

       Activate(Arbiter.Interleave(
                new          TeardownReceiverGroup(Arbiter.Receive<DsspDefaultDrop>
(false,_mainPort,DropHandler)),
                new ExclusiveReceiverGroup(
                    Arbiter.Receive<ReceivedPacket>
(true,_internalPort,PacketHandler)),
                new ConcurrentReceiverGroup(
                    Arbiter.Receive<StartContinousRead>
(true,_mainPort,StartContinousReadHandler),
                    Arbiter.Receive<StopContinousRead>
(true,_mainPort,StopContinousReadHandler),
                    Arbiter.Receive<StartSingleRead>
(true,_mainPort,StartSingleReadHandler),
                    Arbiter.Receive<Get>(true, _mainPort, HttpGetHandler),
                    Arbiter.ReceiveWithIterator<Subscribe>
(true,_mainPort,SubscribeHandler))
       ));
}
```

Figure 6.3 Start method for the SickLRF_Scanner service

This class is responsible for creating a new `SickLRF_ScannerState.` The `SickLRF_ScannerState` contains important information about the service such as current distant measurements, angular resolution and angular range. Next the `StartLRF()` method

creates a new `TCPIOManager` which is responsible for communicating with the SICK LMS100. The `SickLRF_ScannerService` class starts the `TCPIOManager` with an IP address of 130.195.162.58 on Port 2111 (required to find and connect with the SICK LMS100). The `TCPIOManager` is explained in greater detail in Section 6.3.2.

Finally the class sets up seven message handlers using `Aribiter.Receive<>()` and adds them to the main threading interleave which control the flow of information throughout the class. The seven messages the handlers receive are:

- `StartContinousRead`
- `StopContinousRead`
- `StartSingleRead`
- `ReceivedPacket`
- `Get`
- `Subscribe`
- `DsspDefaultDrop`.

The Arbiter.Receive method format is as follows:

Arbiter.Receive<"*Message Type*">(*Persistent Receiver Boolean, Port to receive message on, Handler method to call on message arrival*).

The `DsspDefaultDrop` message handler is created under the `TeardownReceiverGroup` which classifies messages that close down the service. This message is the only non-persistant handler as it is declared with a false Boolean during handler setup.

The `ReceivedPacket` message is sent internally from the `TCPIOManager` and is created under the `ExclusiveReceiverGroup` while the other six messages, which are sent externally from subscribing services, are created under the `ConcurrentReceiverGroup`, indicating these messages can be handled concurrently with other messages.

The `StartContinousRead` and `StartSingleRead` handlers instruct the `TCPIOManager` to send telegrams to the SICK LMS100 scanner to start a continuous or single scan of the environment while the `StopContinousRead` handler instructs it to send a telegram to stop continuous scans.

The `ReceivedPacket` handler receives messages from the `TCPIOManager` when scanned data telegrams arrive. A snippet of the packet handler can be seen in Figure 6.4. The handler updates the current state with the new distance measurements received and then posts a message to subscribed services notifying them of the new distance measurements.

```
void PacketHandler(ReceivedPacket packet)
        {
            switch (packet.CommandType){

                case "sSN":
                    _state.TimeStamp = DateTime.Now;
                    _state.NumberOfMeasurements = packet.length;
                    _state.DistanceMeasurements = packet.Data;
                    _state.AngularRange = 270;
                    _state.AngularResolution = 0.5;
                    _subMgrPort.Post(new submgr.Submit(_state,
DsspActions.ReplaceRequest));
                     return;
                 case "sRA":
              ...
```

Figure 6.4 Received packet handler method

The `Get` handler receives requests, from either another service or an http website request, for an update on the current state. When the `Get` is requested from another service, a return message is sent containing the entire current state of the Sick Scanner service. When the `Get` is requested from an http website, a JPEG image representation of the environment is returned to be viewed in a web browser. An example of the returned image is shown in Figure 6.5.

The `Subscribe` handler receives messages from services requesting to get distance measurement updates from the `SickLRF_Scanner` service. The handler adds the subscribing service to the list of current subscribers and posts a success message to the subscriber indicating a successful subscription.

The final handler, `DsspDefaultDrop`, is called when the service is shutdown. The handler instructs the `TCPIOManager` to close communication with the SICK scanner and then closes the `SickLRF_Scanner` service.

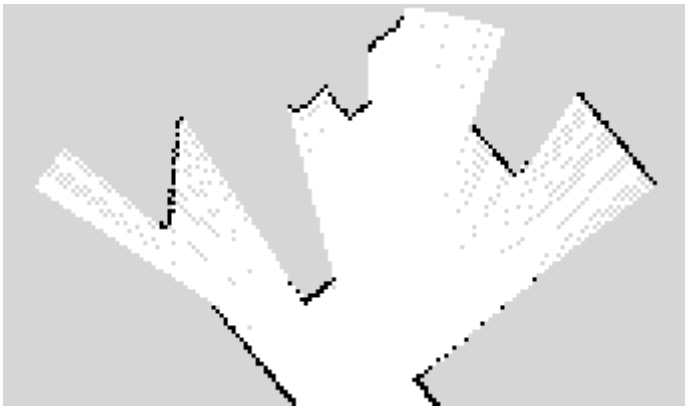

Figure 6.5 Returned image example from a HTTP Get request message

6.3.2 TCPIOManager Class and Packet Class

The `TCPIOManager` (TCP input output manager) class is the communication class responsible for connecting to the SICK LMS100 using TCP/IP, disconnecting the TCP/IP connection when the service closes, sending telegrams to the scanner and receiving telegrams from the scanner.

A `TCPIOManager` is created by the `SickLRF_ScannerService` class to manage communication with the SICK LMS100. When created, the `TCPIOManager` attempts to connect to and open a `NetworkStream` with the SICK scanner. The `Connect()` method can be seen in Figure 6.6. If the connection is unsuccessful (scanner is unplugged) the `TCPIOManager` will respond with an error message, informing subscribers that the SICK LMS100 is unavailable. Once connected, the `StartRead()` method is started which generates an interrupt when a telegram is available to be read.

```
public void Connect(String server, Int32 port)
        {
            try
            {
                // Create a TcpClient.
                client = new TcpClient(server, port);
                // Get a client stream for reading and writing.
                stream = client.GetStream();
                //start reading packets
                _internalPort.Post(new StartRead());
            }
            catch (ArgumentNullException e)
            {
                _internalPort.Post(new Error("ArgumentNullException: {0}", e);
            }
            catch (SocketException e)
            {
                _internalPort.Post(new Error("SocketException: {0}", e);
            }
        }
```

Figure 6.6 Connect method within the TCPIO Manager class

As mentioned in Section 3.3.2, telegrams are the packet structure used for communicating to and from the SICK LMS100 laser scanner. The `TCPIOManager` class supports sending three types of telegrams to the laser scanner (Table 6.1) and receiving five types of telegrams from the laser scanner (Table 6.1). Each telegram is framed with a start of frame character (STX) and end of frame character (EXT) as shown in Table 6.3.

Table 6.1 Supported telegrams sent to scanner

Telegram Message	Description
sRN LMDscandata	Start single read
sEN LMDscandata 1	Start continuous read
sEN LMDscandata 0	Stop continuous read

Table 6.2 Supported telegrams received from scanner

Telegram Message	Description
sRS LMDscandata	Confirm message to start single read
sEA LMDscandata 1	Confirm message to start continuous read
sEA LMDscandata 0	Confirm message to stop continuous read
sRA LMDscandata	Single scan data packet
sSN LMDscandata	Continuous scan data packet

Table 6.3 Telegram frame

	Frame	Telegram	Frame
Code	STX	Data	ETX
Length (byte)	1	≤ 30 kB	1
Description	Start of text character	ASCII coded. The length is dependent on the previous send telegram.	End of text character

As the SOPAS Engineering Tool software can be used to configure the scanner, telegrams relating to setting the scan rate, resolution and range are not implemented in this project's control software. This is because once the scanner is configured, the project is not required to change any settings during normal operation. SOPAS was used to configure the scan rate at 50Hz, angular range to 270° and angular resolution to 0.5°.

The sRN telegram requests a single data scan back from the SICK LMS100. Figure 6.7 shows the sRN telegram structure as well as the ASCII telegram packet with framing that is sent to the range finder. The range finder responds with a sRS LMDscandata telegram to confirm receiving the request, then the range finder sends a sRA LMDscandata telegram containing the single scan data. Figure 6.8 shows an example of a single scan request and response from the scanner.

Telegram	Description	Variable	Length	Values ASCII
Command Type	Sopas by name	String	3	sRN
Command	Only one Telegram	String	11	LMDscandata

ASCII | <STX>sRN{SPC}LMDscandata<ETX>

Figure 6.7 sRN LMDscandata telegram structure

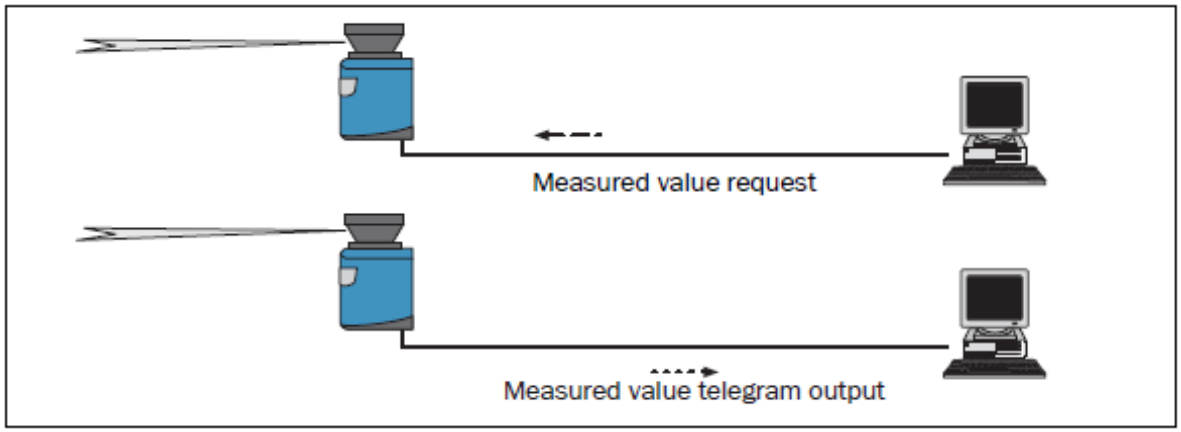

Figure 6.8 Single scan request example

The next implemented telegram is sEN LMDscandata. The sEN telegram requests the scanner to continuously scan and send back data until instructed to stop. To start continuous scanning, the control computer sends an ASCII telegram as shown in Figure 6.9 with the value of 1. To stop continuous scanning, another ASCII telegram is sent with a value of 0. The scanner responds with a sEA LMDscandata telegram with a value of 1 to confirm starting and a value of 0 to confirm stopping. After confirming the start of a continuous scan, the laser scanner will send sSN LMDscandata telegrams containing the distance measurements. Figure 6.10 shows an example of a continuous scan request and response from scanner until requested to stop.

Telegram	Description	Variable	Length	Values ASCII
Command Type	Sopas by name	String	3	sEN
Command	Data Telegram	String	11	LMDscandata
Measurement	Start/Stop	Enum_8	1	0 Stop 1 Start

ASCII | <STX>sEN{SPC}LMDscandata{SPC}1<ETX>

Figure 6.9 sEN LMDscandata telegram structure

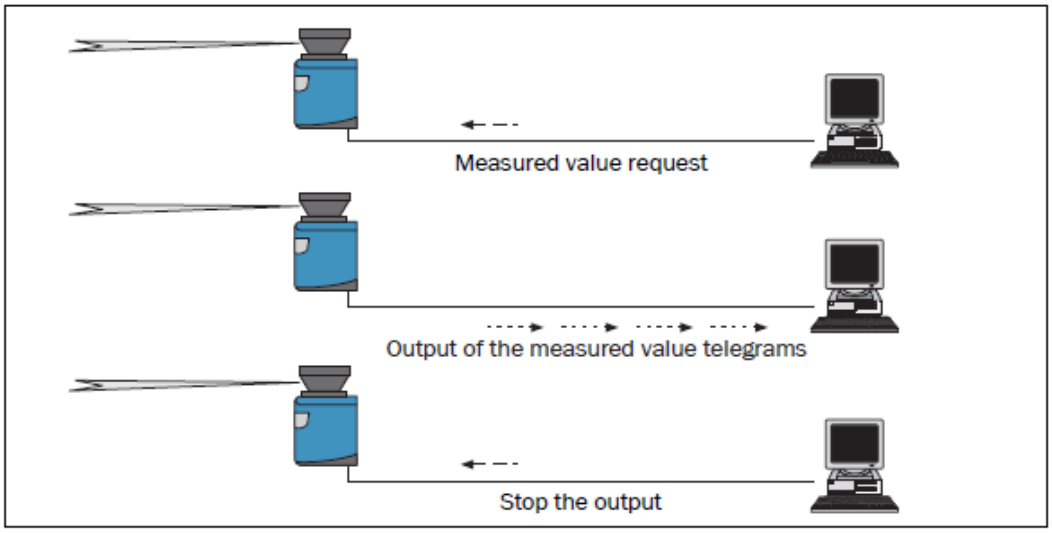

Figure 6.10 Continuous scan request example

The sRA LMDscandata and sSN LMDscandata telegrams contain components separated by space characters. When one of these telegrams arrive they are sent to the Packet class for processing.

The Packet class is responsible for splitting the received packet data into separate components. Components include the command packet name, packet number, packet length, angular resolution, angular range and the distance data.

Once the telegrams have been processed the packet is posted to the SickLRF_scanner class where the service state is updated with the latest distance measurements and subscribers notified.

6.4 Segway Base Service

The Segway interface consists of two services working together to allow the controlling computer to interface with the Segway RMP200 via USB. The first service is a low level service, '`SegwayNativeWrapper`', that handles USB communication to the Segway. The second service, '`SegwayBase`', is built on top of MRDS's Generic Differential Drive (GDD) service contract that provides a common specification for differential drive mobility platforms. The service runs on the local host and found at port 50001.

6.4.1 Segway Native Wrapper

Segway Native Wrapper service is written in C++ and is based on the example code provided by Segway Inc. and modified to allow the functionality required in MRDS. Segway Native Wrapper service is an interface library to the ftd2xx.dll which opens up USB communication, reads and writes command packets to and from the Segway platform, and closes the connection when required. The service is made up of two files, 'usb_int.cpp' and 'SegwayNativeWrapper.cpp'

On service start, the Segway Native Wrapper service loads the Segway's USB drivers and connects to the first Segway platform found. Once connected, the service can send control messages to the Segway and receive monitoring messages from the Segway. Segway control messages were discussed in Section 3.1.5 and monitoring messages were discussed in Section 3.1.6.

The `SegwayNativeWrapper.cpp` defines seven `structs` for the seven messages sent from the Segway. They are labelled `MessageData1-7`. The `SegwayNativeWrapper` is responsible for reading the received data buffer and putting the received values into the appropriate `MessageData` fields. When all seven fields have been updated, an interrupt is set for the `SegwayBase` service to read and update its state variables.

The important methods used in the USB_int.cpp file are summarised in Table 6.4. and the important methods used in the `SegwayNativeWrapper.cpp` file are summarised in Table 6.5.

Table 6.4 USB_int.cpp important methods and summary

Method	Summary
Usb_Init()	Loads the Segway USB driver and connects to the first Segway device found.
Usb_Active()	Returns true if there is an active USB link to a Segway.
Usb_LoadDLL()	Load the DLL and setup the library calls.
Usb_CloseDLL()	Close and unload the DLL.
Usb_Write(Tx buffer)	Write a buffer to the USB interface.
Usb_Read()	Read into a buffer from the USB interface.
Usb_Close()	Closes the connection to the Segway.
Usb_message_format	Format a message for the USB and calculate the checksum for the message being sent. Buffer is expected to be exactly 18 bytes in length.
Usb_can_send(string)	Send a CAN formatted message via USB.

Table 6.5 SegwayNativeWrapper.cpp important methods and summary

Method	Summary
SegwayNativeWrapperClass()	Constructor. Calls the Init() method, sets up RX buffer and sets velocity and turn to zero.
~SegwayNativeWrapperClass()	Destructor. Clears the RX buffer and closes the USB DLL
Init()	Makes a call to initialise and start a USB connection to the Segway.
Drive(int,int)	Sends a drive command to the Segway. Takes a velocity and turn integer.
SetMaxVelScale(double)	Sends a message to the Segway platform to set the maximum velocity scale factor.
SetMaxAccScale(double)	Sends a message to the Segway platform to set the maximum acceleration scale factor.
SetMaxTurnScale(double)	Sends a message to the Segway platform to set the maximum turning scale factor.
SetGainSchedule(double)	Sends a message to the Segway platform to set the Gain schedule.
SetCurrentLimitScale(double)	Sends a message to the Segway platform to set the current limit scale factor.
SetOperationMode(int)	Sets operation mode for the Segway. 1=tractor, 2=balance,3=off.
Shutdown()	Causes the Segway unit to immediately turn off.

6.4.2 SegwayBase Service

As previously mentioned, the `SegwayBase` service implements the GDD contract defined within MRDS. The Generic Differential Drive service defines how to control a differential drive robot (Microsoft, 2012). As the service implements the GDD service contract, the Segway platform can be swapped for a generic differential drive system on any robot without need to change code. The `SegwayBase` service is designed to be a generic service that can be used by any MRDS application wanting to use a Segway platform.

The `SegwayBase` service starts by creating a new `SegwayNativeWrapper` class which connects to the Segway platform. The service then defines the main operating port, sets up interrupts for update messages from the `SegwayNativeWrapper`, configures the Segway and then starts a control timer.

The main operations portset (Figure 6.11) defines seven messages that can be used to change the current state of the `SegwayBase` service by external services. Table 6.6 summarises the seven messages.

```
public class SegwayBaseOperations : PortSet<
        Drive,
        SetOperationMode,
        ResetIntegrator,
        Replace,
        Get,
        Subscribe,
        DsspDefaultDrop>{}
```

Figure 6.11 The main operations portset used by the SegwayBase service

Table 6.6 SegwayBase service main operations port messages

Message Type	Description
Drive	Sends a drive command to the Segway platform with a target linear velocity and target turn rate. Values are saved to the state and sent to the Segway at the next command timer interrupt.
SetOperationMode	Sets operation mode for the Segway. 1=tractor, 2=balance,3=off.
ResetIntegrator	Tells the service to reset the odometers on the Segway. The commands are bit field operations so can be OR'd together to reset multiple odometers at once. 1 = right wheel displacement 2 = left wheel displacement 4 = yaw displacement 8 = fore/aft displacement
Replace	Updates the entire `SegwayBase` with the received replaced state.
Get	Sends the entire `SegwayBase` to the service whom sent the Get message.
Subscribe	Informs the `SegwayBase` service that another service wants to subscribe to this service and receive update messages whenever the state is changed.
DsspDefaultDrop	Informs the `SegwayBase` service to stop and shutdown the service.

The `SegwayBase` service is interrupted by the `SegwayNativeWrapper` when a new set of Segway messages arrives. The most recent values from the `SegwayNativeWrapper` update the `SegwayBase`'s state and a notification message is sent to all subscribers indicating the change in state.

```
IEnumerator<ITask> ConfigureSegway()
        {
            _segway.SetGainSchedule(0);
            _segway.SetMaxAccScale(0.5);
            _segway.SetMaxVelScale(0.5);
            _segway.SetMaxTurnScale(0.5);
            _segway.SetCurrentLimitScale(1.0);
            _segway.ResetAllIntegrators();

            //start sending periodic commands
            _timerPort.Post(DateTime.Now);
            Activate(Arbiter.Receive(true, _timerPort, TimerHandler));

            _segway.getUSBData();
            yield break;
        }
```

Figure 6.12 Configure Segway method within the SegwayBase service

The `ConfigureSegway` method is shown in Figure 6.12 which sets the scale factors, resets the encoder values and sets up a control timer that sends the current velocity command and turn command to the Segway at a frequency of 20 Hz. As previously mentioned, if the Segway platform does not receive control messages at a rate of at least 2.5 Hz the Segway platform slews its velocity to zero. The control timer is set to 20 Hz, the same speed as the hybrid navigation system.

The scale factors are sent to the Segway each time it is connected and before any velocity command is issued to the Segway. Table 6.7 shows the scale factor values used in this project.

Table 6.7 Segway scale factor values

Scale Factor	Value
Gain Schedule	0
Max Acceleration Scale	0.5
Max Velocity Scale	0.5
Max Turn Scale	0.5
Current Limit Scale	1.0

Section 3.1.5 describes each scale factor. The Gain schedule is set at 0 for light payloads, the maximum acceleration, velocity and turn scales are set at 0.5, which limit the Segway to a maximum linear velocity of 1.7 metres per second maximum and a maximum angular

velocity of 1.7 radians per second. The current limit scale is set at 1.0 which does not limit the available torque to the motors. The motor torque can be decreased in low friction environments where high torques cause excessive wheel slippage (which was not observed in the operating environment).

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> DriveHandler(Drive drive)
{
      _state.SetVelocityCommand = drive.Body.Velocity;
      _state.SetTurnCommand = drive.Body.Turn;

      _segway.Drive(_state.SetVelocityCommand, _state.SetTurnCommand);
      drive.ResponsePort.Post(DefaultUpdateResponseType.Instance);
      yield break;
}
```

Figure 6.13 Drive handler method within SegwayBase service

The `DriveHandler` method is shown in Figure 6.13. It takes the new velocity and turn targets from the `Drive` message and updates the state velocity values. The new values are then sent to the Segway platform.

6.5 SegwayNavigation Service

The `SegwayNavigation` service implements the components of the hybrid navigation algorithm discussed in Chapter 5.

The `SegwayNavigation` service partners and subscribes to the Sick LRF Service and the `SegwayBase` service. This allows the service to request and receive updates on range finder data as well as command and receive the current state of the Segway. The Segway UI service (Section 6.6) will partner and subscribe to the `SegwayNavigation` service. The `SegwayNavigation` service relies on the Segway UI service for the current operating map, current position, target position as well as commands to start autonomous path following.

Three timers are used to execute different tasks of the hybrid navigation system. These three timers, shown in Figure 6.14, are used to update the current position of the Segway, calculate target angular and linear velocities and command the Segway to move with the target angular and linear velocities.

The Segway navigation service consists of three classes: the `SegwayNavigation` class which is responsible for implementing the hybrid navigation algorithm, the `SegwayNavigationState` class which is responsible for maintaining the current state of the `SegwayNavigation` service, and the `SegwayNavigationOperations` class which is responsible defining communications with partnered services as well as the required MRDS operations. These three classes are discussed further in the following sections. The service runs on the local host and found at port 50002.

6.5.1 SegwayNavigation Class

The `SegwayNavigation` class is the main controlling element for navigation system for the Segway. The service starts by defining two timers: `SegwaySubscriptionTimer` and `SickSubscriptionTimer`. These two timers check for and attempt to subscribe to the `SegwayBase` and `SickLRF_Scanner` services at 10 Hz. When successfully subscribed, the timers are set to 1 Hz and used as a watchdog to ensure communication with the lower level services. If either of the lower level services stops responding then the Segway's navigation algorithm discontinues and the Segway is brought to a standstill until the subscription can be established again.

Replace messages from both the `SegwayBase` and `SickLRF_Scanner` service are received each time the respective service updates its current state. The replace message handlers update the `SegwayNavigation`'s state with new range finder data or odometry from the Segway.

The service then defines three timers (as mentioned above) to execute different tasks of the navigation system. The three timers are called `SensingTimer`, `DriveTimer` and `AutonomousTimer`. All three timers are executed at 20 Hz.

The `SensingTimer` extracts landmarks from the SICK LRF rangefinder data. The landmarks extracted and how they are extracted has been explained in Section 5.5. The `SensingTimer` is also responsible for calling the `SensorFusion` method which determines the position and orientation of the Segway by fusing the odometry information and landmark correction.

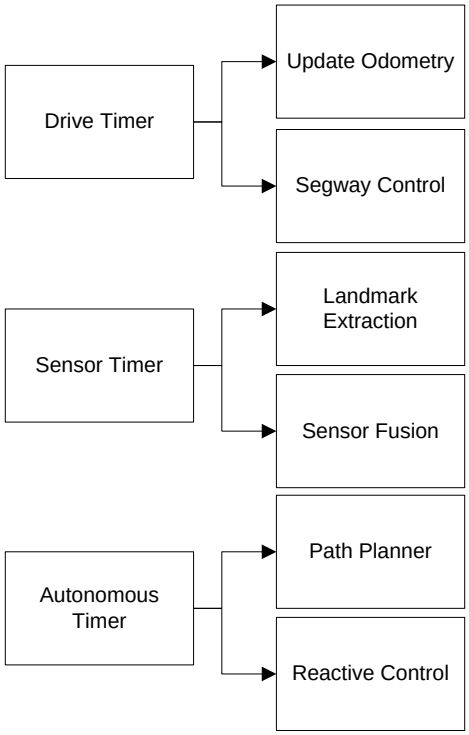

Figure 6.14 SegwayNavigation timers

The `DriveTimer` updates the current coordinate position of the Segway using odometry as explained in Section 5.4.2. If the Segway is operating in autonomous mode, the `DriveTimer` sends the current target angular velocity and target linear velocity from the navigation system to the `SegwayBase` service which instructs the Segway to move. The target angular and linear velocities are given in rad/s and m/s respectively and require conversion to command values the Segway can interpret. The conversion from target velocities given from the navigation system to Segway command values is discussed further in Section 7.2.2. If the Segway is operating in manual mode, the current target angular and linear velocities sent from the Segway UI service are sent to the `SegwayBase` service.

The `AutonomousTimer` checks the current operating mode and path planning flags. If the Segway is currently running in autonomous mode and a path is found by the path finding algorithm, the hybrid navigation control algorithm covered in Chapter 5 is run. The navigation algorithm includes the path tracking algorithm, the direction sensor algorithm and the dynamic window algorithm.

The `SegwayNavigation` class also defines handlers for the operation messages described in Section 6.5.3. These messages can be posted on the main operating port of the service to

update the state of the service. A brief overview of each message handler is given in Table 6.8.

6.5.2 SegwayNavigation State Class

The `SegwayNavigationState` class defines all the variables that make up the current state of the `SegwayNavigation` service. A new service state is created when the `SegwayNavigation` class is run which sets initial values to some service state variables. The list below summarises the important state variables for the `SegwayNavigation` service:

- The Sick LRF data members: distance measurements, angular resolution, angular range and the last received message from the `SickLRF_Scanner` service timestamp.
- The recent values for the Segway data members: all values discussed in Section 3.1.6 including the current encoder counts, wheel velocities, pitch angle, distance between wheels, tyre diameter and the last received message from the `SegwayBase` service timestamp.
- The Segway navigation environment map data points and connections.
- Landmark databases: the landmarks extracted from the environment map database and landmarks extracted from laser scanner database.
- Current and target position coordinates.
- Path planning details: list of nodes along path and error flags.
- Reactive control values: target angular and linear velocity.
- Dynamic window parameters: linear and angular velocity limits.

6.5.3 SegwayNavigation Operations Class

The `SegwayNavigationOperations` class contains and defines the main operating port for the `SegwayNavigation` service. The main operating port defines eight messages which other services can send to change the state of the `SegwayNavigation` service. Four of the messages are required by MRDS while the other four update state parameters. The `SegwayNavigationOperations` class defines the messages while the

`SegwayNavigation` class implements the handlers of the messages. The eight messages are presented in Table 6.8 with a brief description of the contents of each message as well as any state parameters they change.

Table 6.8 SegwayNavigation operations port messages

Message Type	Description
UpdateMapPoints	Updates the service with the current operating environment map. Changes the `MapPointCoordinates` and `MapPointConnectivity` state parameters. Receipt of this message causes landmarks to be generated from the given map and stored in the `MapLandmarkDatabase` state parameter.
UpdateGridResolution	Updates the resolution of the operating environment map. Changes the GridResolution state parameters. Receipt of this message causes the occupancy grid map to be updated as well as the A* path planning method to be invoked.
UpdateInitTargetPose	Updates the starting coordinates (InitPose) and target coordinates (TargetPose) of the Segway platform.
UpdateDriveMode	Updates the current operating mode of the navigation system. This messages tells the navigation system to change either manual or autonomous mode. If autonomous mode is required, the hybrid navigation system is enabled.
Replace	Updates the entire `SegwayNavigationState` with the received replaced state.
Get	Sends the entire `SegwayNavigationState` to the service whom sent the Get message.
Subscribe	Informs the `SegwayNavigation` service that another service wants to subscribe to this service and receive update messages whenever the state is changed.
DsspDefaultDrop	Informs the `SegwayNavigation` service to stop and shutdown the service.

6.6 Segway UI service

The Segway UI service class is made up of the Segway UI service and a Graphic User Interface (GUI) allowing human interaction with the Segway. The GUI is responsible for creating an interaction between the operator and the Segway while the controlling service conforms to the MRDS CCR service requirements. The Segway UI service is capable of running remotely on another computer to control the Segway platform. The GUI is created using WinForms to create a simple interface to the Segway platform. The Segway UI service subscribes to the `SegwayNavigation` service to receive updates about the current navigation state. The service runs on a networked computer and found at port 50003.

The service is responsible for the following tasks:

- Displaying the distance measurements from the SICK LMS100 scanner.
- Displaying the current odometer encoder values.
- Displaying the current wheel velocities and pitch/roll angles.
- Displaying the current coordinate position of the Segway.
- Drive distance (metres) and rotate (degrees) commands.
- Control Segway in manual mode with joystick.
- Setting maximum motor power for drive distance, rotate degrees and joystick commands.
- Reading environment map data from file and sending the data to the `SegwayNavigation` service.
- Set occupancy grid resolution and display occupancy grid.
- Set current and target coordinates and heading.
- Display environment map with current and target positions.
- Emergency stop button on GUI and joystick.
- Change between operating modes: balance, tractor and off as well as manual or autonomous.

A WinForm is a separate module, not a service in its own right. Because it operates as a single threaded apartment model it cannot wait on CCR ports to receive messages. However, the main service needs to update information on the Form in response to notification messages

such as game controller updates. Sending information from the main service to the Form is done using `FormInvoke` method which allows transferring of information to a WinForm.

The form needs to pass back commands to the main service. When the form is interacted with, events fire inside the Form code. The WinForm events are not related to the CCR in any way, but the event handlers in the form send CCR messages to the main services by posting messages to the services `EventsPort`. Messages received on the events port are listed and described in Table 6.9.

Table 6.9 Segway UI service's events port

Message type	Description
DriveDistance	Instructs the Segway UI service to drive the Segway platform a certain distance in metres.
RotateDegrees	Instructs the Segway UI service to rotate the Segway platform by a certain angle in degrees.
OnStop	Instructs the Segway UI service to send an emergency stop signal to the Segway .
OnModeSet	Allows the GUI to set the current operating mode of the Segway, either tractor, balance or off.
OnDriveMove	The GUI sends OnDriveMove request to the UI service when the Segway navigation system is in manual mode and is currently being commanded to move using a joystick.
ResetEncoders	Instructs the Segway UI service to send a message to the `SegwayBase` service to reset all encoder values.
GridMap PointData	Instructs the Segway UI service to send the map point data and connectivity data to the `SegwayNavigation` service.
GridMap Resolution	Instructs the Segway UI service send the map resolution data to the `SegwayNavigation` service.
UpdateInit TargetPose	Instructs the Segway UI service to send the initial and target position coordinates to the `SegwayNavigation` service.
AutoMode Enabled	Instructs the Segway UI service to send a Boolean value to the `SegwayNavigation` service indicating manual control or autonomous mode for the Segway platform.

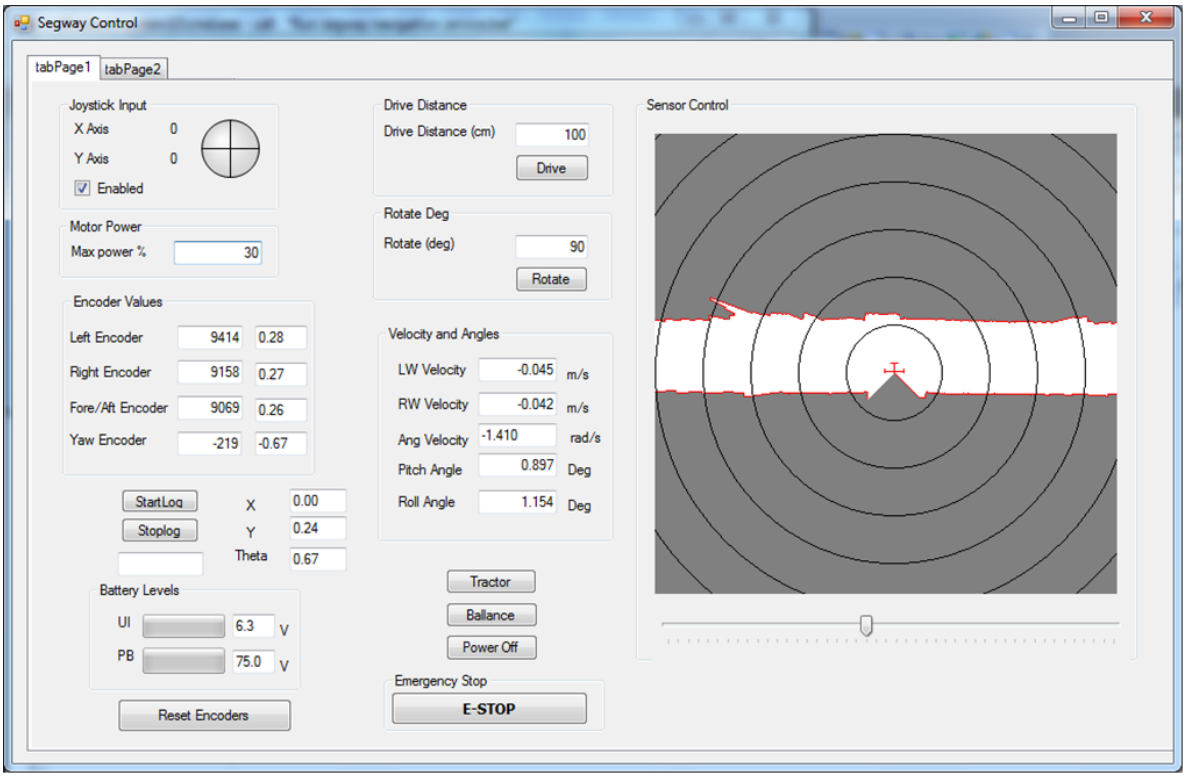

Figure 6.15 User interface tab 1

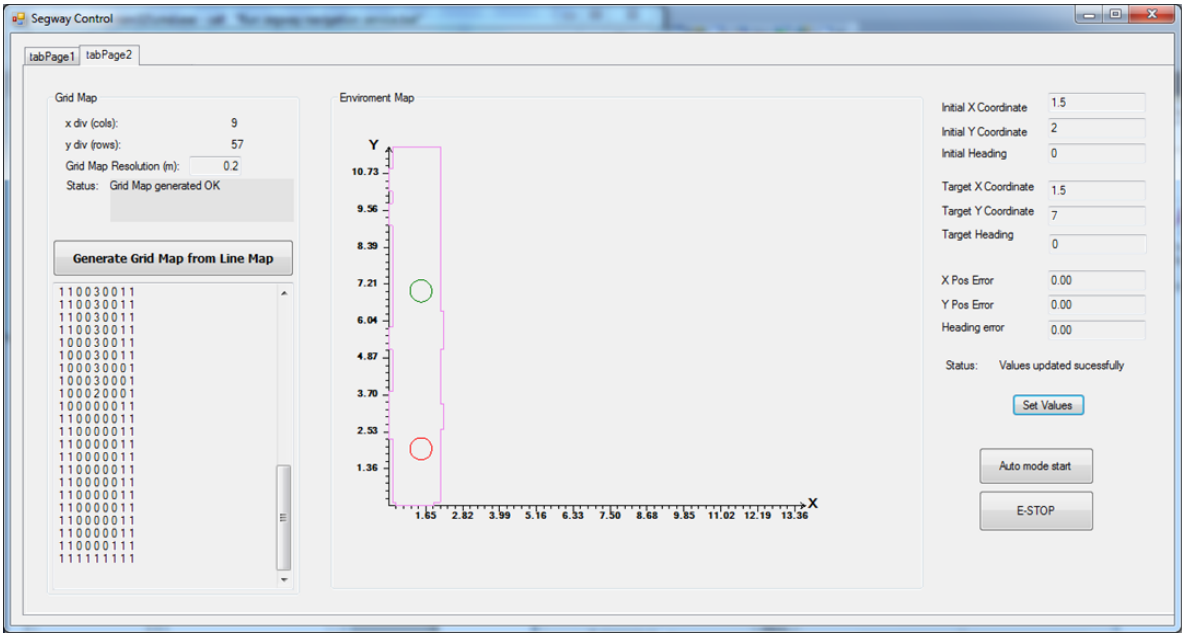

Figure 6.16 User interface tab 2

The first tab of the user interface can be seen in Figure 6.15 User interface tab 1Figure 6.15. The first tab focuses on sensor feedback and manual control of the Segway. It shows encoder, velocity and angle values from the Segway as well as the current position (x, y, θ). The drive distance and rotate degrees commands for the Generic Differential Drive contract and joystick commands can also be set from the first tab. An emergency stop button is available to stop the Segway when required. The first tab also displays a visual representation of the distance measurements received from the SICK LMS100 laser scanner.

The second tab (Figure 6.16) focuses on the navigation features of the system. It shows the current environment map as well as the current position (red) and target position (green) and allows a user to set the two positions. A button allows the operating mode to switch between manual and autonomous operation.

6.7 Summary

Using the SOA architecture instigated in MRDS, the hybrid navigation system designed by Chand has been implemented as a software framework to allow the Segway platform to navigate autonomously. The software is created using a three tiered system where the hardware composes the lowest tier, the navigation system composes the middle tier and the user interface composes the top tier. The software implements both manual and autonomous control of the Segway as desired by the user through the UI service.

The system is made up of four separate services working together: the first service controls the SICK LMS100 laser scanner, the second service controls the Segway platform, the third service implements the hybrid navigation algorithm and the fourth service implements the user interface allowing human interaction with the system.

Chapter 7 Results

7.1 Sick LRF Characterisation

The parameters of the SICK-LMS100 were tested in an indoor environment. All of the measurements were taken inside the Laby building at Victoria University of Wellington. An experimental setup was created that reproduced the main aspects of indoor usage. As the Segway's main operating environment is indoors with florescent lighting, the datasets were collected in a room lit up with florescent lighting at normal light intensity and standard indoor operating temperature (18-20 °C). The SICK-LMS100 sensor was tested with 270° angular range, 0.5° angular resolution with a 50 Hz scan rate. The SICK-LMS100 has several built in data filters implemented in the firmware which improved performance of the sensor in fog as well as measuring the second reflective beam (used for measuring object distances through glass). As none of these features are required for the normal operation of the Segway, they were not tested.

The results of the test can be seen in Figure 7.1. It was found that the settling time (standard deviation within 0.01 m of steady state) of the sensor is approximately 35000 scans which at a scan rate of 50 Hz, is about 12 minutes.

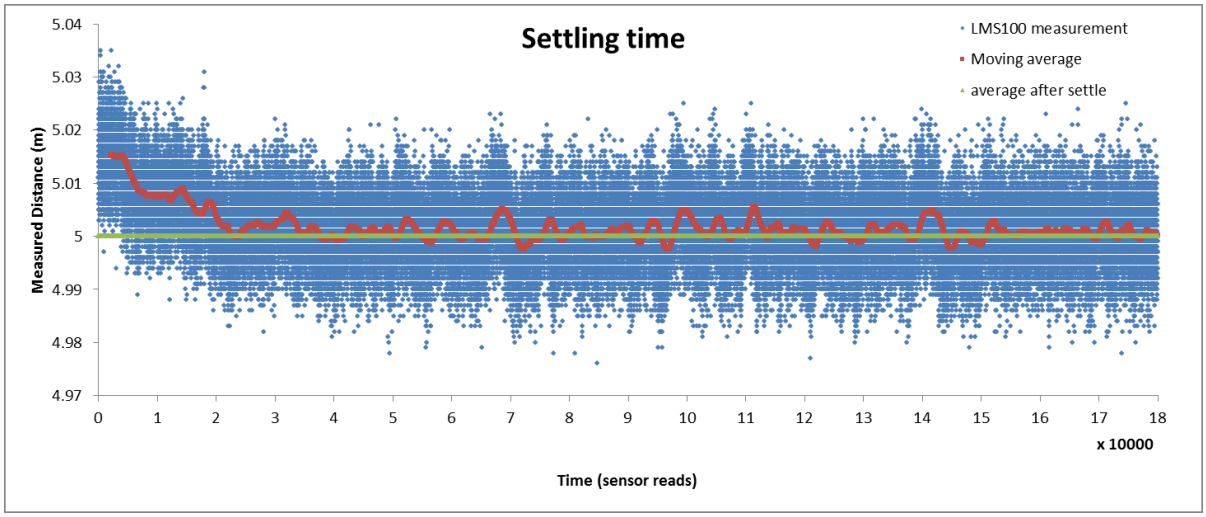

Figure 7.1 Sick LMS100 settling time

The second measurement was set to determine the time-dependent variation of the SICK LMS100 scanner in a static setup. The object was positioned 5 m in front of the scanner. Three reflective surfaces were used for the measurements representing the extremes of the environment the Segway could be expected to operate in. The first reflective surface tested was a black coloured segment of wall, chosen because it represented the minimum reflective object in the operating environment. The results of the experiment on the black reflective surface can be seen in Figure 7.2. The average measured distance to the black surface was 5.004 metres, with a standard deviation of 0.007 metres.

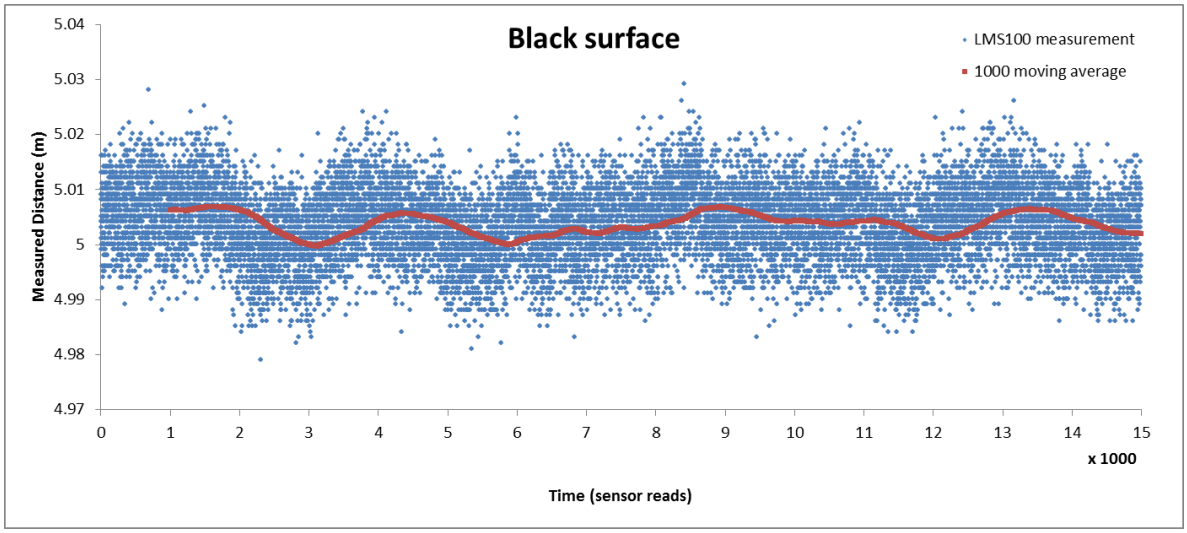

Figure 7.2 Distance measurements to black surface

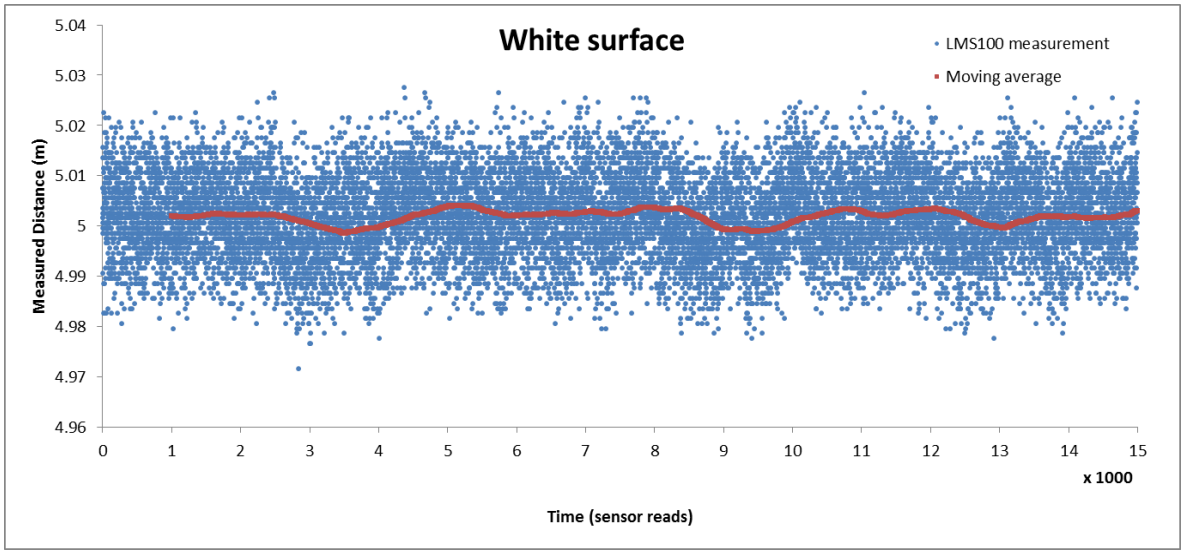

Figure 7.3 Distance measurements to white surface

The experiment was repeated using a white reflective surface, chosen because it represented the maximum reflective object in the operating environment. The results of the experiment on the white reflective surface can be seen in Figure 7.3. The average measured distance to the white surface was 5.002 metres, with a standard deviation of 0.008 metres.

The experiment was repeated a third time using a glass surface, chosen because there are many glass surfaces/walls within other corridors that could be new operating environments at Victoria University. The results of the experiment on the glass surface can be seen in Figure 7.4. The average measured distance to the glass surface was 5.023 metres, with a standard deviation of 0.008 metres.

From these tests the SICK LMS100 laser scanner produces accurate measurements with a maximum standard deviation of 0.008 m over 5 m. These accurate measurements are sufficient for robot navigation and localisation within the desired operating environment.

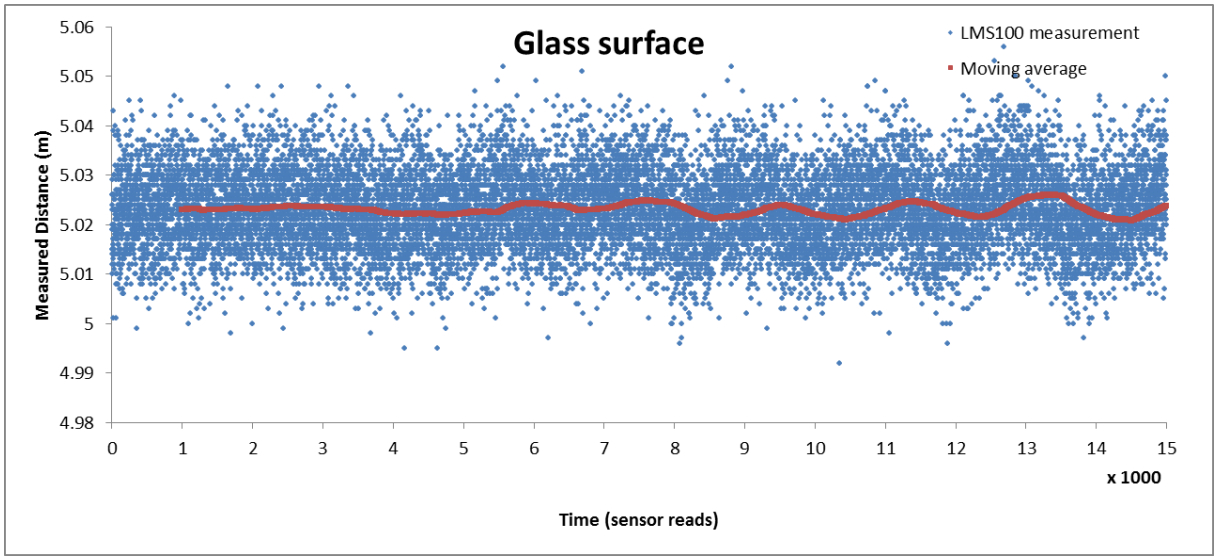

Figure 7.4 Distance measurements to glass surface

7.2 Segway Characterisation

7.2.1 Odometry

The odometry calibration tests were performed in two separate corridors allowing the odometry calibration tests to be performed on two different surfaces. The first in the Laby level 3 corridor measuring 1.75 x 11.4 m was chosen as it is the expected operating environment with a vinyl floor. The second corridor that odometry calibration was performed in was the Cotton level 2 corridor measuring 2.5 x 17 m and was chosen as it could be an operating environment for future projects and the floor is covered with carpet.

The two environments were cleared of any obstacles as the initial odometry tests were conducted before the hybrid navigation algorithm was implemented, meaning the SICK LRF100 rangefinder was not used for localisation purposes. This left only the odometers for localisation, which are susceptible to a number of errors including initial misalignment errors and odometry errors such as wheel slippage and missed encoder counts.

Initial misalignment errors can significantly affect the final position as with only odometry information, the Segway cannot detect or correct initial heading errors. Initial misalignment errors were minimised by using floor markings and a custom jig to align the tyres to be

parallel with the required trajectory. Odometry errors are both random and systematic. Systematic errors can be minimised through odometer calibration while random errors occur due to wheel slippage or missed encoder counts that cannot be avoided, but can be minimised by limiting the acceleration of the Segway and only operate on surfaces with sufficient traction.

Before initial testing, the Segway tyre pressures were set at 6 psi as recommended in the user manual. Straight line tests were conducted over 5 metres in both environments to test initial drift due to non-symmetric wheel diameters. These tests resulted in an average deviation from a straight line trajectory by 42 cm to the right. This was due to the right wheel having a smaller diameter and therefore traveling less distance than the left wheel. This was corrected by increasing the air pressure in the right wheel, thereby increasing its diameter. The tests and adjustments were repeated until the Segway had an average offset error of less than 2 cm over the 5 metres travelled.

An estimated conversion factor of 33215 counts per metre was recommended as the left and right wheel odometry calibration for a nominal rolling diameter of 48 cm. The actual left and right encoder conversion factor was found by measuring the ratio of the actual distance travelled to the distance travelled as calculated by the Segway with a conversion factor of 33215. The test was done over distances of 1 to 5 metres in 1 metre divisions with target velocities ranging from 0.2 m/s to 0.95 m/s in 0.15 m/s divisions. This was to give the average odometry conversion factor for different velocities and distances.

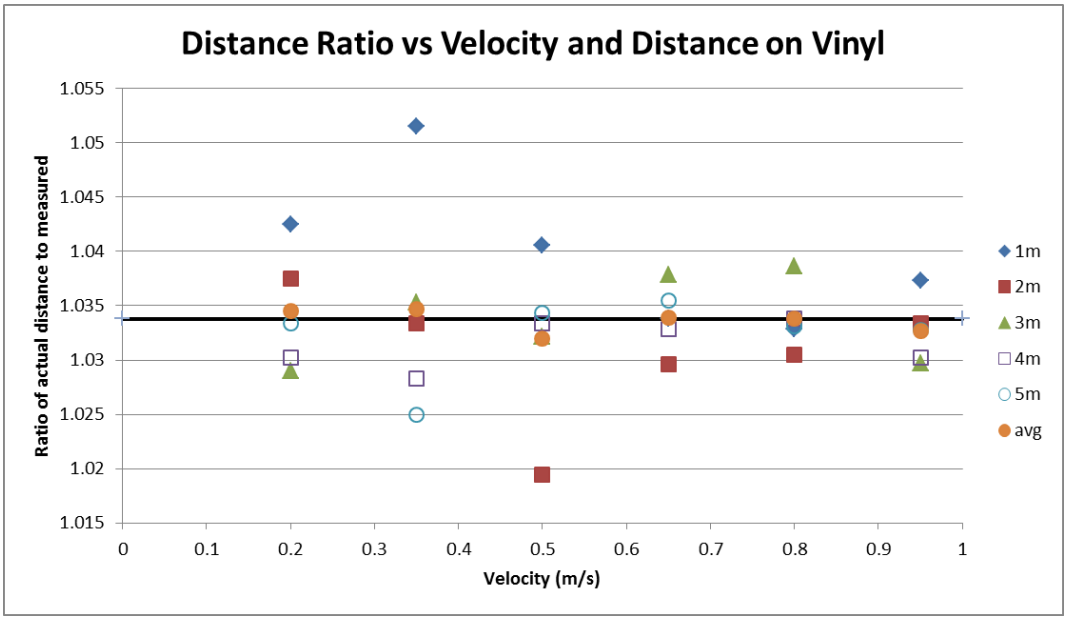

Figure 7.5 Ratio of actual distance to measured distance vs velocity and distance on vinyl

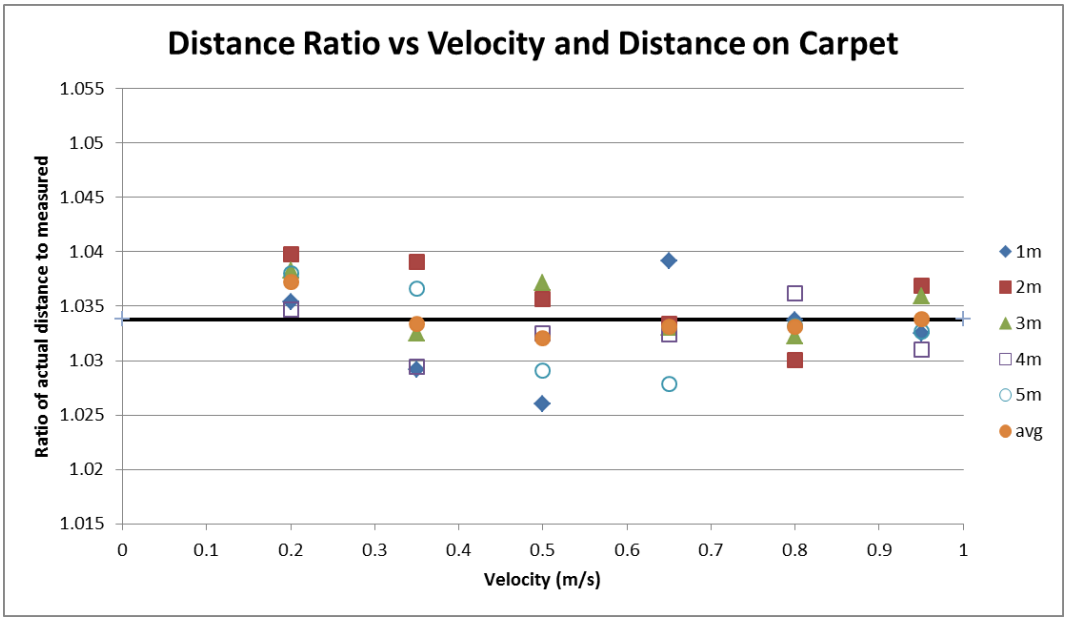

Figure 7.6 Ratio of actual distance to measured distance vs velocity and distance on carpet

Each trial was done three times with the average result for each distance measurement shown in Figure 7.5 and Figure 7.6. The average for carpet was 1.03229 while the average ratio for vinyl was 1.03527. The average ratio for both vinyl and carpet, represented by a solid black line, is 1.03378. This meant the recommended conversion factor of 33215 was too low by 3.38% and was increased to 34337 counts per metre. The standard deviation of all the wheel

odometer errors was 0.11% giving an indication of random error due to wheel slippage over the distance travelled.

The yaw encoder output is calculated internally by the Segway using the left and right wheel encoders. It was expected that the yaw encoder would also have an error of 3.38% and require calibrating.

The actual yaw encoder conversion factor was found by measuring the ratio of the actual rotation in degrees travelled to the rotation turned as calculated by the Segway with a conversion factor of 112644. The test was carried out over rotations of 180 to 900 degrees in 180 degree divisions with target angular velocities ranging from 15 deg/s to 40 deg/s in 5 deg/s divisions. This was to give the average odometry conversion factor for different velocities and distances.

Each trial was done three times with the average result for each rotation measurement shown in Figure 7.7 and Figure 7.8. The average for each angular velocity is also shown. The average ratio for both vinyl and carpet, represented by a solid black line, is 1.03611. This meant the recommended conversion factor of 112644 was too low by 3.61% and was increased to 116711. The standard deviation of the yaw odometer errors was 0.19% giving an indication of random error due to wheel slippage over the distance travelled

The measured error in the yaw encoder was higher than the expected error of 3.38% by 0.23%. The average yaw error for carpet was 3.64% while the average yaw error for vinyl was 3.58%. This error difference could be due to greater wheel slippage during turns when compared to linear movements.

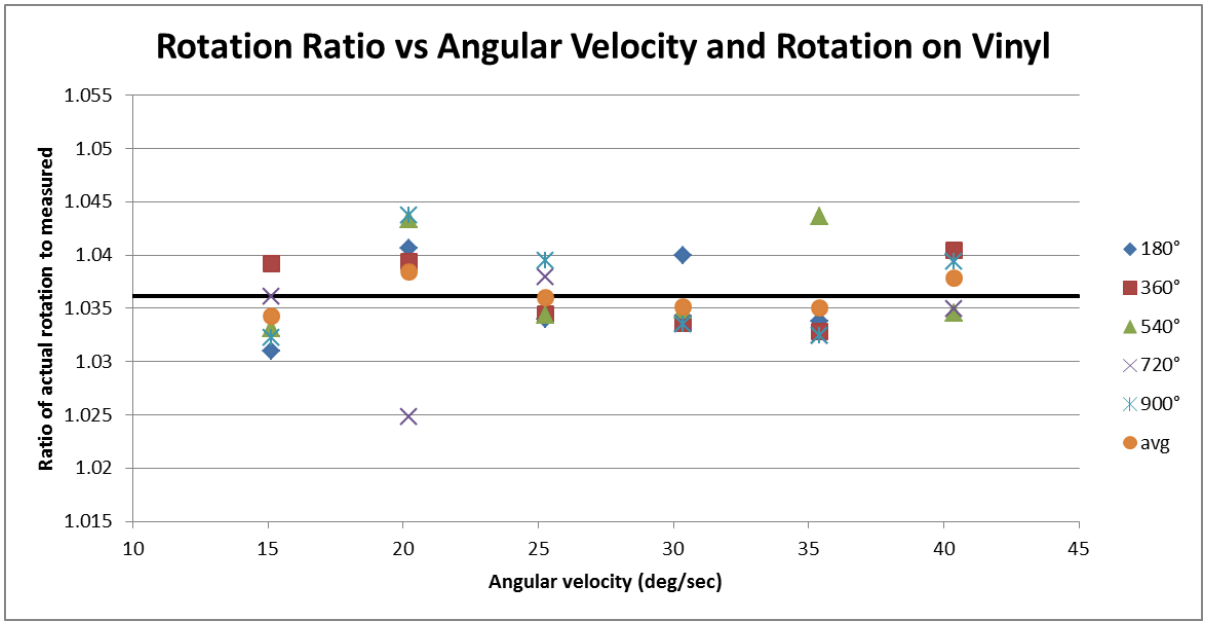

Figure 7.7 Ratio of actual rotation to measured rotation vs angular velocity on vinyl

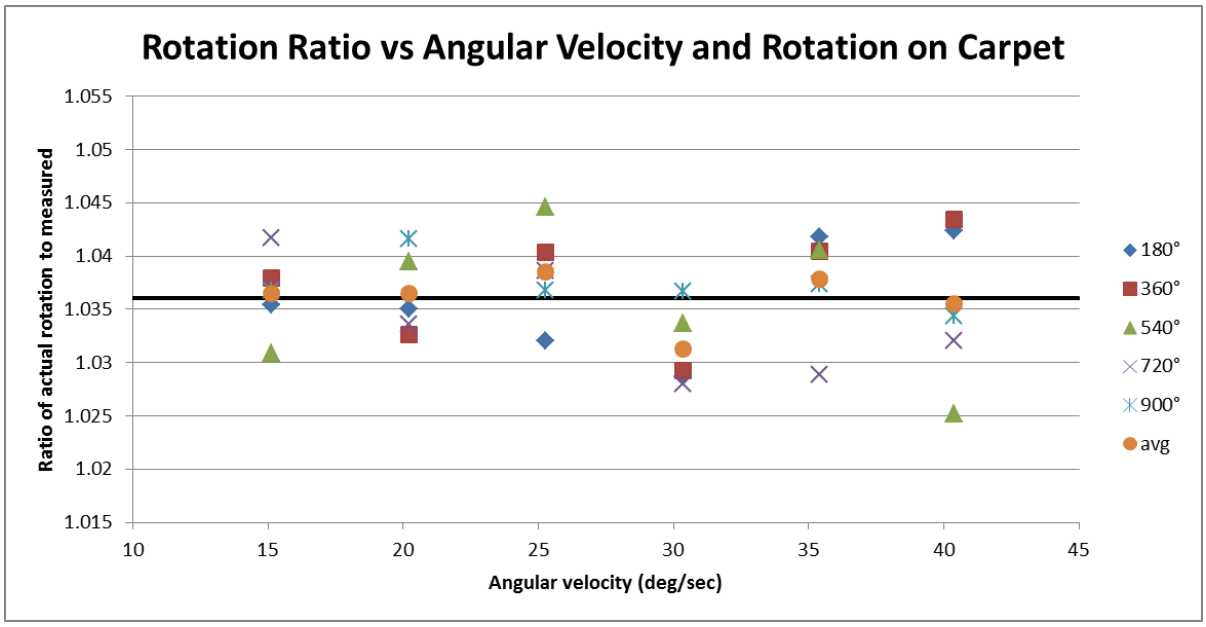

Figure 7.8 Ratio of actual rotation to measured rotation vs angular velocity on carpet

7.2.2 Segway Characterisation

Due to the dynamic stabilisation of the Segway platform there is not a one to one relationship between the velocity of the wheels and velocity of the platform. Tests were carried out in an open environment to observe the motion of the Segway during operation. Particular attention is given to the wheel velocities and pitch angle the Segway undergoes during movement.

These tests were carried out without the navigation system to determine the stopping distance required for different wheel velocities and to investigate the relationship between pitch angle and velocities during straight line movement. The left and right wheel movements were the same during testing and only the left wheel data is graphed as results during straight line trajectories. The pitch angle gives a representation of the centre of gravity of the Segway relative to the wheel axis.

The Segway platform was commanded to move at 0.25, 0.3, 0.4, 0.5, 0.6 and 0.75 m/s. Emergency stop commands were sent to the Segway when the odometers had measured a displacement of 1, 2, 3, 4 and 5 metres. Only the 0.3, 0.5 and 0.75 m/s results are shown in this section as the other results follow similar trends as presented and do not offer further discussion.

Figure 7.9 shows the wheel displacement for a 0.3 m/s target velocity for the five distances. The average stopping distance was 0.47 m, with a maximum stopping distance of 0.55 m during the 5 m test and minimum stopping distance of 0.27 m during the 2 m test. The reason for the 0.29 m difference between the 2 m and 5 m tests can be seen in Figure 7.10.

Figure 7.10 shows the wheel velocity for a 0.3 m/s target velocity for the five distances. From the graph, the Segway's wheels reverse slightly to tilt the platform, stabilisation then occurs as the centre of gravity moves forward of the Segway's axis. After the initial backwards movement, the wheel velocity accelerates up to 0.6 m/s. The wheel velocity then oscillates between 0.2 m/s and 0.4 m/s to maintain 0.3 m/s velocity of the platform. When given the stop command, the acceleration spikes high to bring the centre of gravity back behind wheels axis and then slows. Some undershoot occurs causing a negative velocity during stopping to maintain stabilisation.

The relatively large difference between the 2 m and 5 m stopping distance is due to the current velocity of the Segway platform when the stop command was issued. The Segway's velocity was slowing when the 2 m stop command was given while it was accelerating when the 5 m stop command was given. The two vertical lines indicate when the stopping commands were issued in the three graphs.

Figure 7.11 shows the pitch angle during the 0.3 m/s velocity test. The pitch angle changes rapidly during starting and stopping as the Segway platform stabilises. The maximum pitch angle was 5° during acceleration and -6° during deceleration.

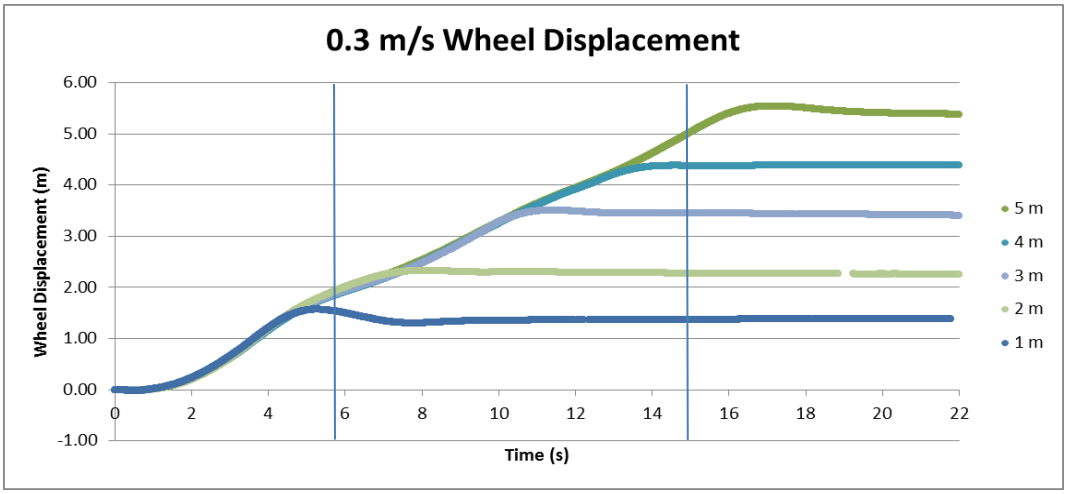

Figure 7.9 Left and right wheel displacement with 0.3 m/s velocity command

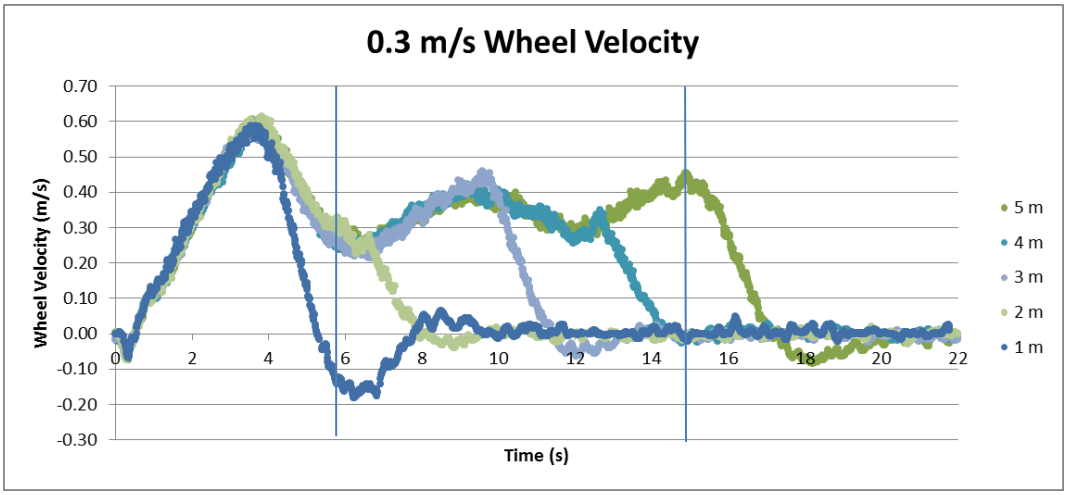

Figure 7.10 Left and right wheel velocities with 0.3 m/s velocity command

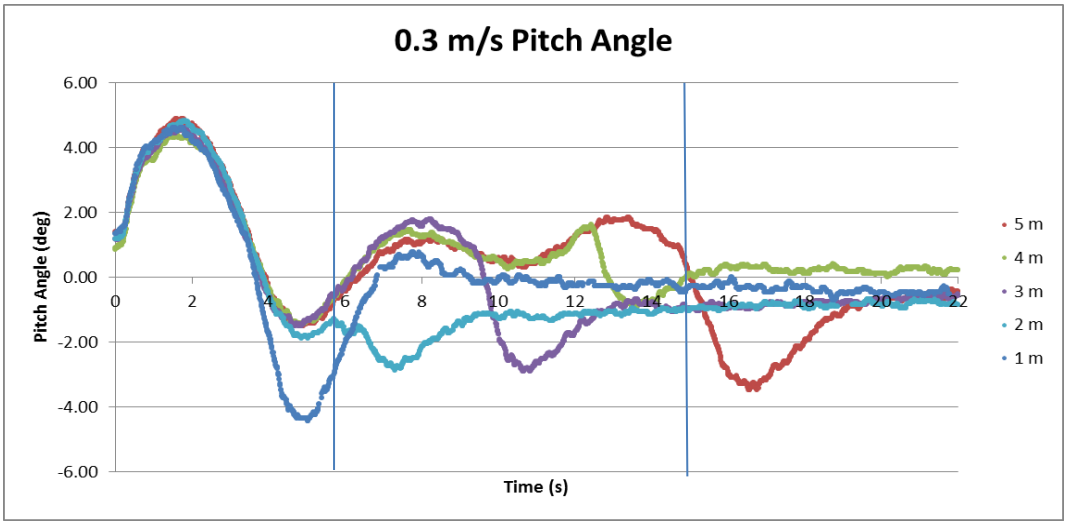

Figure 7.11 Segway pitch angle with 0.3 m/s velocity command

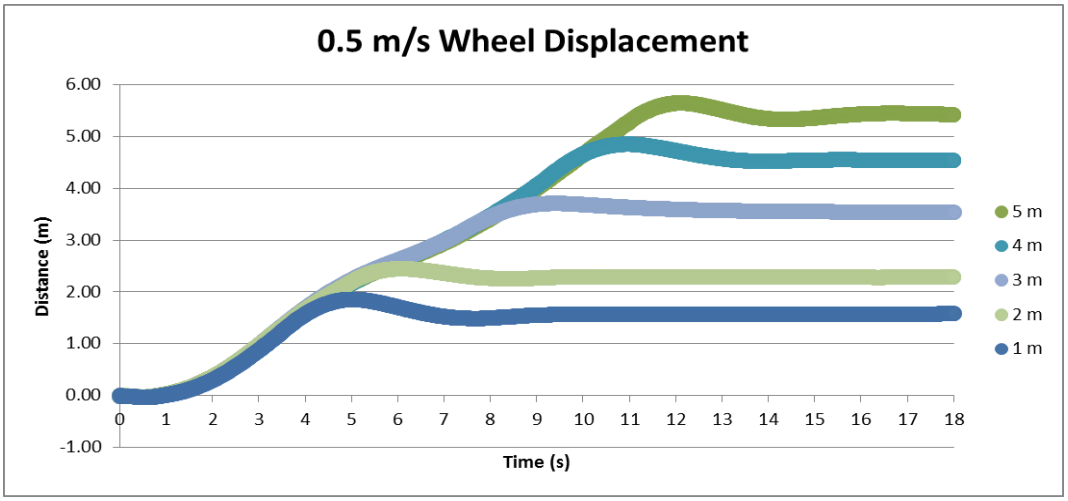

Figure 7.12 Left and right wheel displacement with 0.5 m/s velocity command

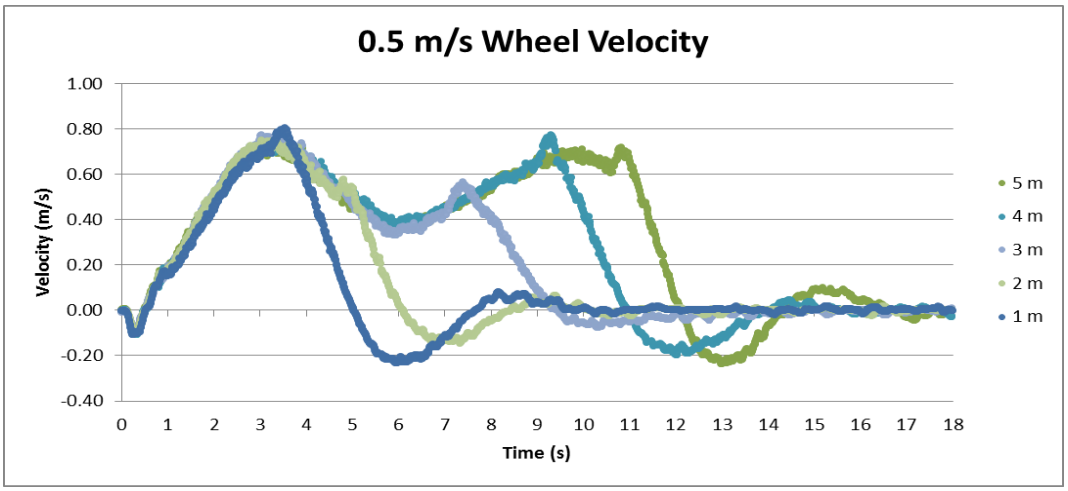

Figure 7.13 Left and right wheel velocities with 0.5 m/s velocity command

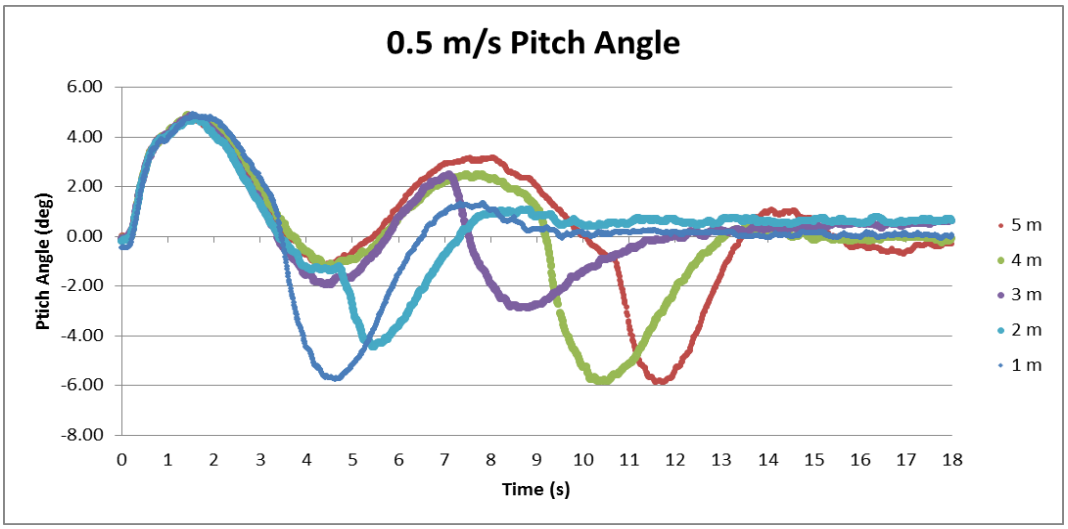

Figure 7.14 Segway pitch angle with 0.5 m/s velocity command

Figure 7.12 shows the wheel displacement for a 0.5 m/s target velocity. The average stopping distance was 0.68 m with a maximum stopping distance of 0.85 m during the 5 m test and minimum stopping distance of 0.43 m during the 4 m test.

Figure 7.13 shows the wheel velocity for a 0.5 m/s target velocity. The Segway again reverses slightly to tilt the platform then accelerates to maintain stabilisation with the centre of gravity slightly in front of the wheel axis. The wheel velocity reaches a maximum at 0.8 m/s and then oscillates between 0.4 m/s and 0.65 m/s. Again there is a spike when the stop command is issued to bring the centre of gravity back behind the wheel axis and then slows.

Figure 7.14 shows the pitch angle during the 0.5 m/s velocity test. Again the maximum pitch angle is reached during acceleration and deceleration, with a maximum pitch angle of 5° during acceleration and -6° during deceleration.

Figure 7.15 shows the wheel displacement for a 0.75 m/s target velocity. The average stopping distance was 0.97 m with a maximum stopping distance of 1.16 m during the 5 m test and minimum stopping distance of 0.53 m during the 3 m test.

Figure 7.16 shows the wheel velocity for a 0.75 m/s target velocity which matches the same characteristics previously mentioned. The wheel velocity reaches a maximum of 1.08 m/s and then oscillates between 0.5 m/s and 0.8 m/s with spikes when stop commands are given.

Figure 7.17 shows the pitch angle during the 0.5 m/s velocity test with a maximum pitch angle of 7° during acceleration and 9° during deceleration.

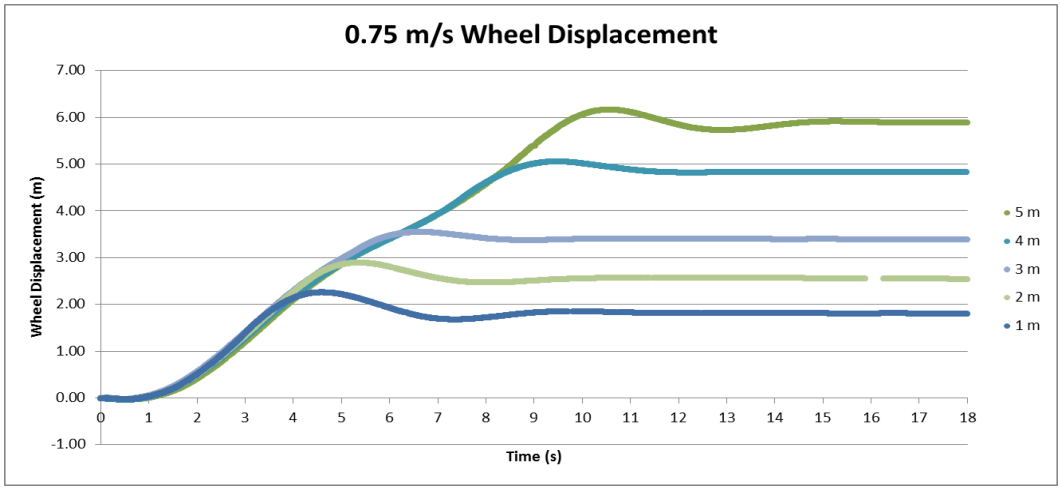

Figure 7.15 Left and right wheel displacement with 0.75 m/s velocity command

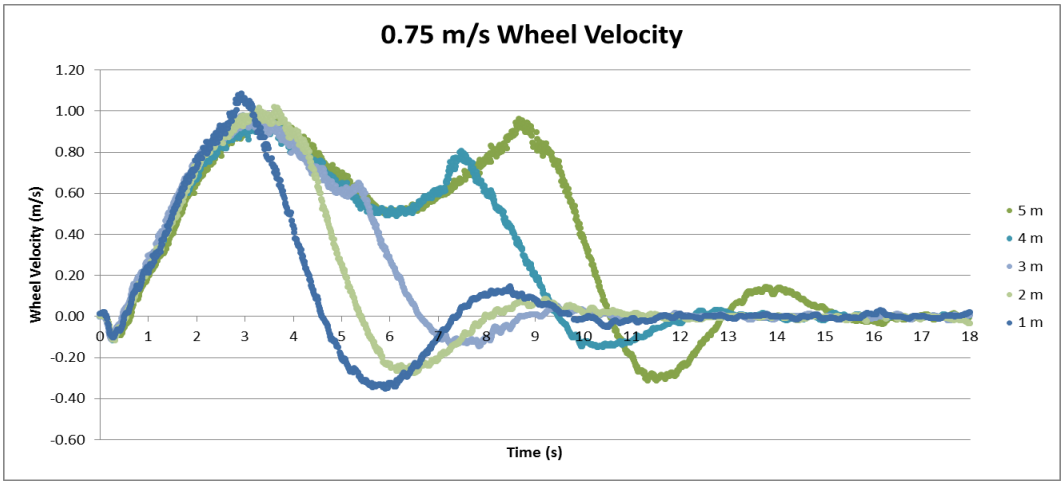

Figure 7.16 Left and right wheel velocities with 0.75 m/s velocity command

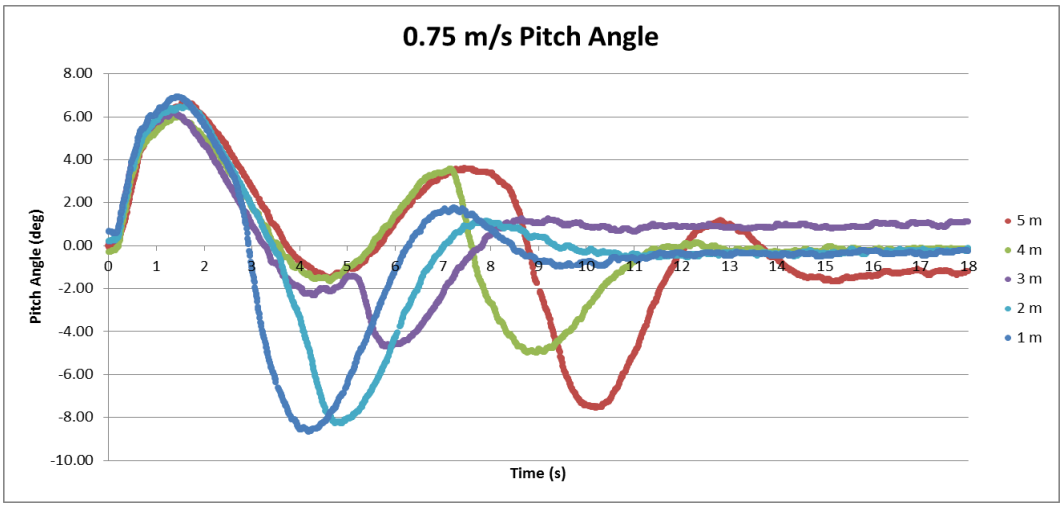

Figure 7.17 Segway pitch angle with 0.75 m/s velocity command

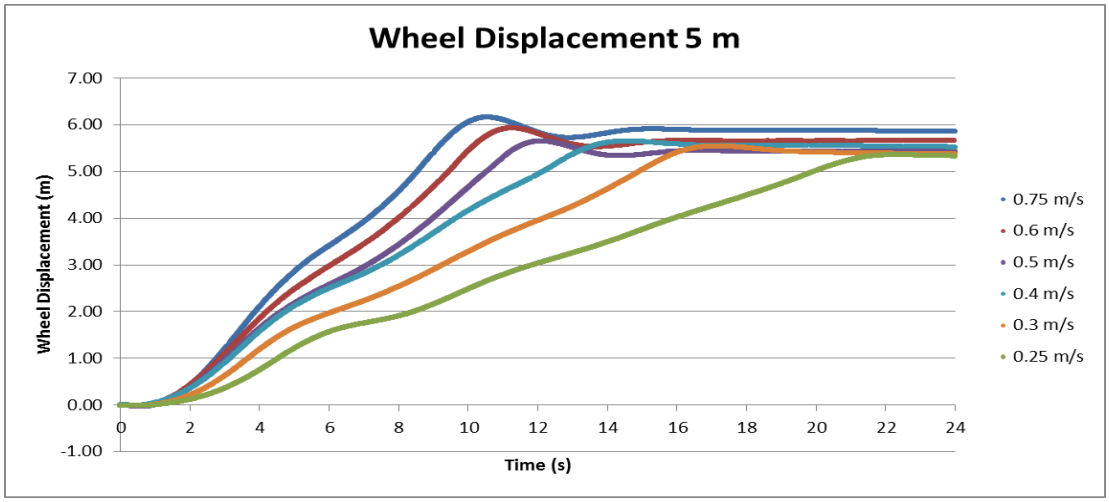

Figure 7.18 Wheel displacement over 5 m for different velocity targets

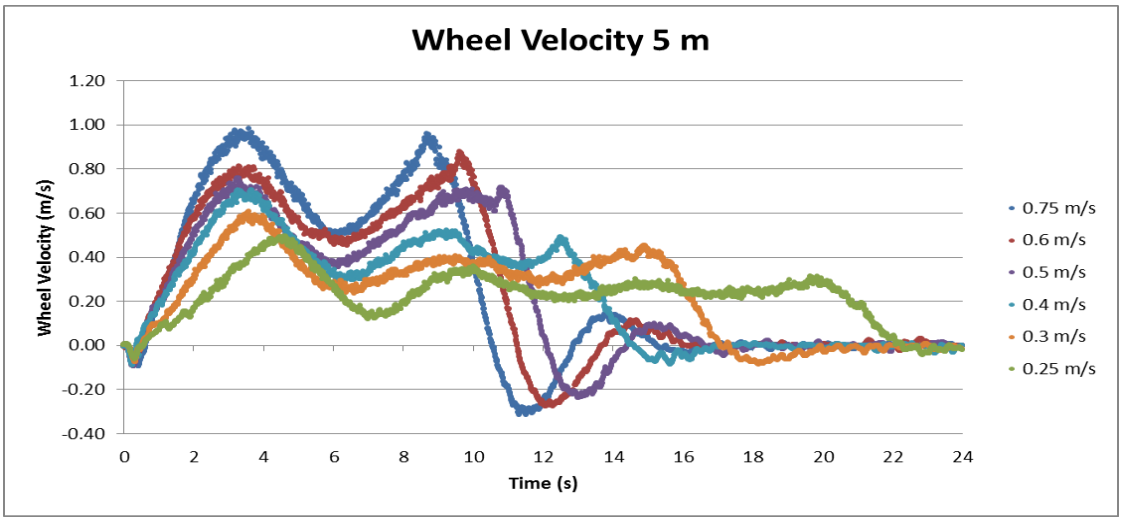

Figure 7.19 Wheel velocity over 5 m for different velocity targets

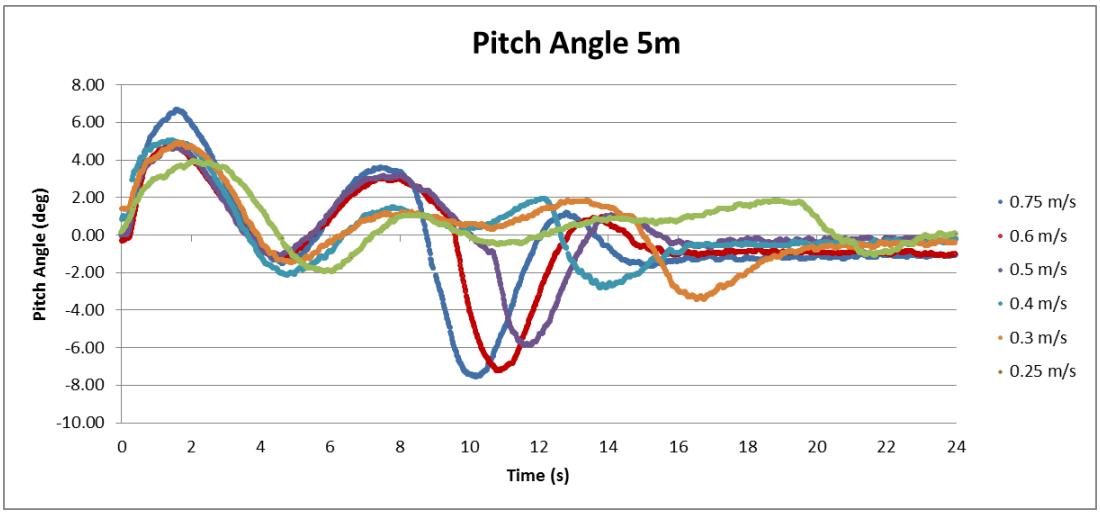

Figure 7.20 Pitch angle over 5 m for different velocity targets

Figure 7.18, Figure 7.19 and Figure 7.20 combine the wheel displacements, wheel velocities and pitch angles for velocity commands of 0.25, 0.3, 0.4, 0.5, 0.6 and 0.75 m/s over 5 m.

Figure 7.18 shows that as expected when the velocity increases, the maximum stopping distance increases as well as the steady state stopping distance. These stopping distances are used to calculate the safety margin growth gain. The stopping distance vs velocity is shown in Figure 7.21.

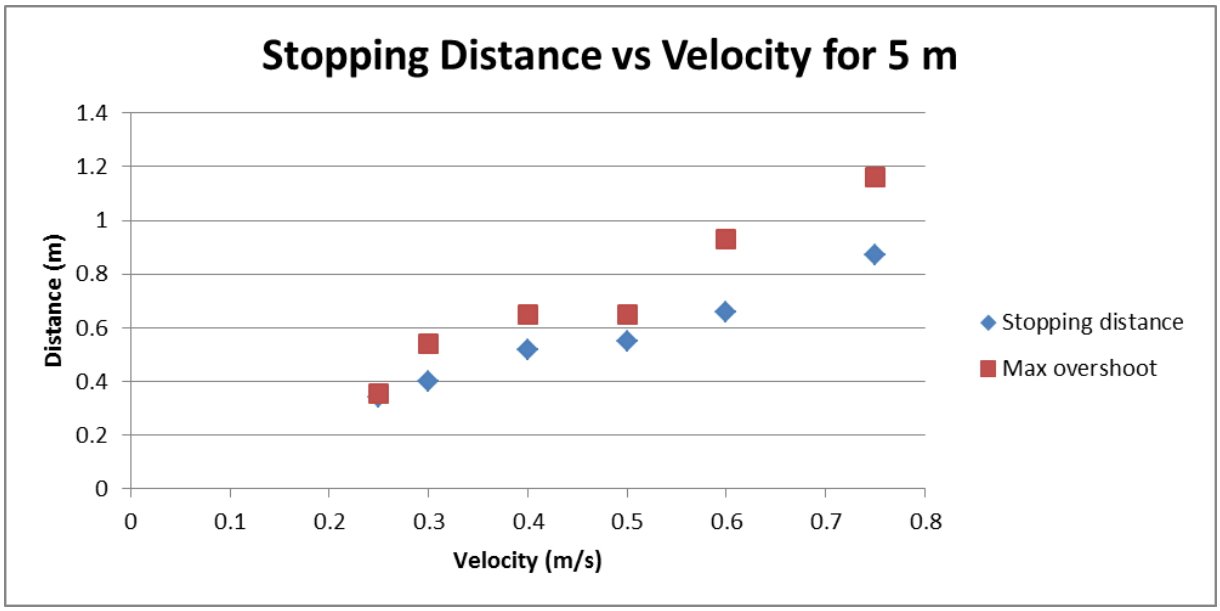

Figure 7.21 Stopping distance over 5 m for different velocity targets

The wheel velocities follow the same profile with an initial negative velocity at 0.5 s followed by a peak velocity at 3.5 s except the 0.25 m/s velocity profile which lags behind other profiles by a second. The average wheel acceleration to the first peak varied between 0.11 m/s^2 for 0.25 m/s velocity target and 0.3 m/s^2 for 0.75 m/s. Velocity peaks occur at 9, 10, 11 and 13 seconds for 0.75, 0.6, 0.5 and 0.4 m/s velocities respectively when the stop commands are issued. The velocity decelerated at between -0.11 m/s^2 for 0.25 m/s velocity target and -0.46 m/s^{-2} for 0.75 m/s. These maximum linear accelerations are used as an input to the dynamic window navigation algorithm.

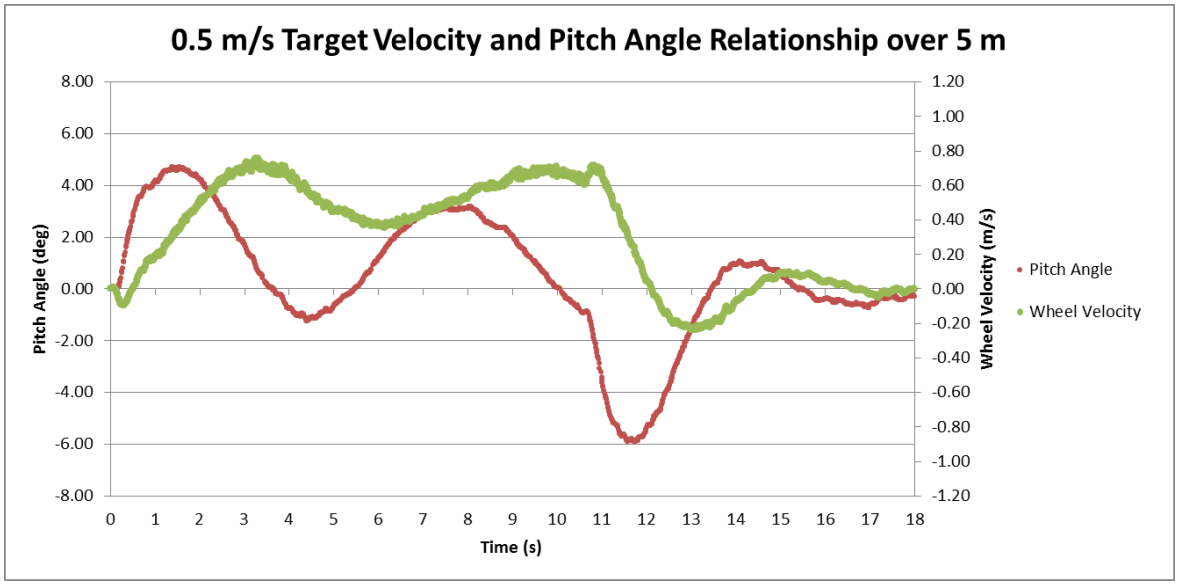

Figure 7.22 Wheel velocity and pitch angle relationship over 5 m for 0.5 m/s target velocity

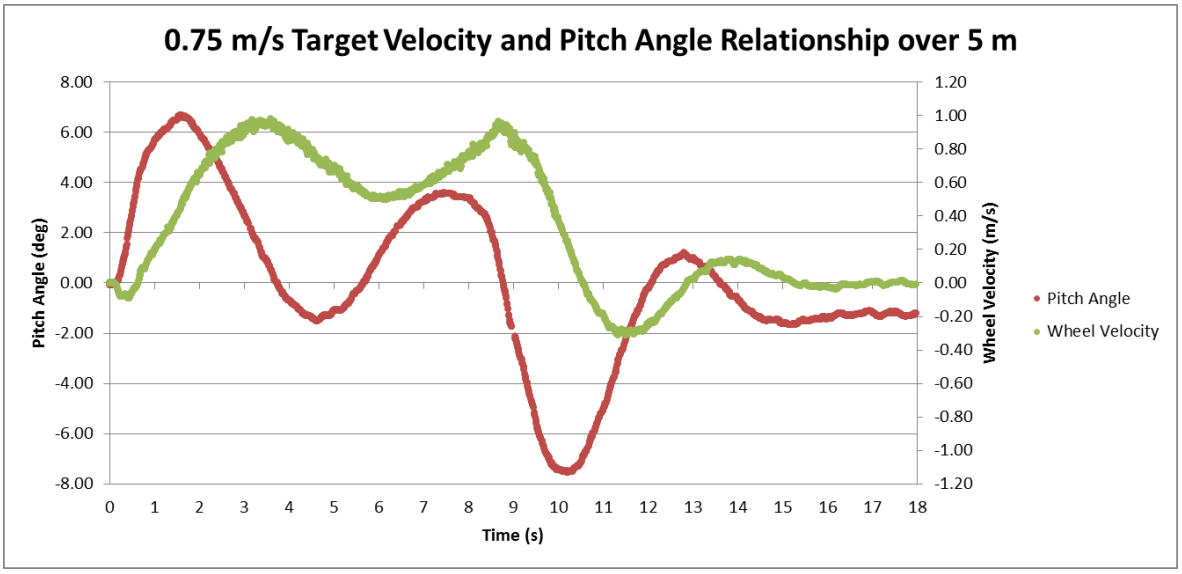

Figure 7.23 Wheel velocity and pitch angle relationship over 5 m for 0.75 m/s target velocity

The relationship between the wheel velocity and pitch angle is plotted for 0.25, 0.5 and 0.75 m/s velocity targets in Figure 7.22 and Figure 7.23. These results show a trend of peaks in the pitch angle being followed by peaks in wheel velocity as well as troughs in the pitch angle being followed by troughs in wheel velocity. This relationship is expected with the dynamic stabilisation occurring. A negative velocity increases the pitch angle shifting the centre of gravity in front of the wheel axis. The wheel velocity increases to maintain stabilisation causing a decrease in pitch angle. Although the pitch angle does go negative between the 4-6

s marks, the top plate of the Segway has a higher amount of momentum, comparative to the wheel base, which coupled with a slowing wheel velocity brings the pitch angle positive while maintaining forward movement of the platform.

The stop command is noticeable at 10.5 s and 9 s, in Figure 7.22 and in Figure 7.23, with a peak in the wheel velocity followed by a sharp decrease in the current pitch angle. The negative pitch angle allows the Segway to oppose forward movement. The largest negative pitch angle occurs at the zero velocity crossing. The velocity continues to go negative and oscillations occur while the Segway dynamically balances.

The dynamic window navigation algorithm calculates a target linear velocity (in m/s) and angular velocity (in rad/s) for the Segway to move. The velocity pair is required to be converted into command values to be sent to the Segway. Table 3.2 shows the relationship between velocity command and speed as [-1176, 1176] = [-12.9 km/h, 12.9 km/h] and the linear velocity scale limits this to [-6.4 km/h, 6.4 km/h]. The turning command has a valid command range of -1024 to 1024 but does not specify the angular velocity values they correspond with. The relationship between input velocity command and linear velocity was tested to confirm the values given in the user manual. The Segway was set to tractor mode so wheel velocities were not affected by the dynamic stabilisation. Velocity commands were sent to the Segway and the linear velocity measured. The results were inversed to give the velocity command value required to set the velocity target (m/s) from the navigation system. The relationship is shown in Figure 7.24 and gives a conversion factor of 666, meaning a target linear velocity of 1 m/s requires a 666 command value to be sent to the Segway.

The relationship between turn command and angular velocity was not given in the user manual and thus was investigated. The test was carried out in balance mode as the dynamic stabilisation does not affect turn capabilities at linear velocity speeds below 1.5 m/s. The results were also inversed to give the angular velocity command required to travel at the target angular velocity from the navigation system. The relationship is shown in Figure 7.25 and gives a conversion factor of 670, meaning a target angular velocity of 1 rad/s requires a 670 command value to be sent to the Segway.

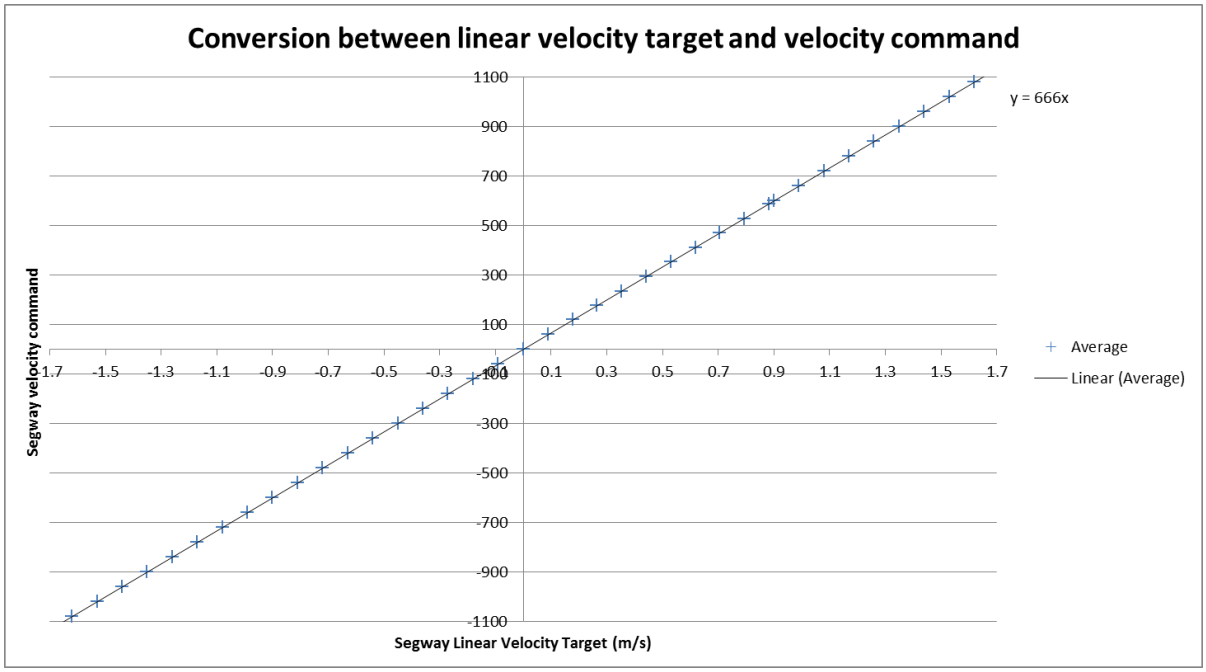

Figure 7.24 Conversion between linear velocity target and velocity command

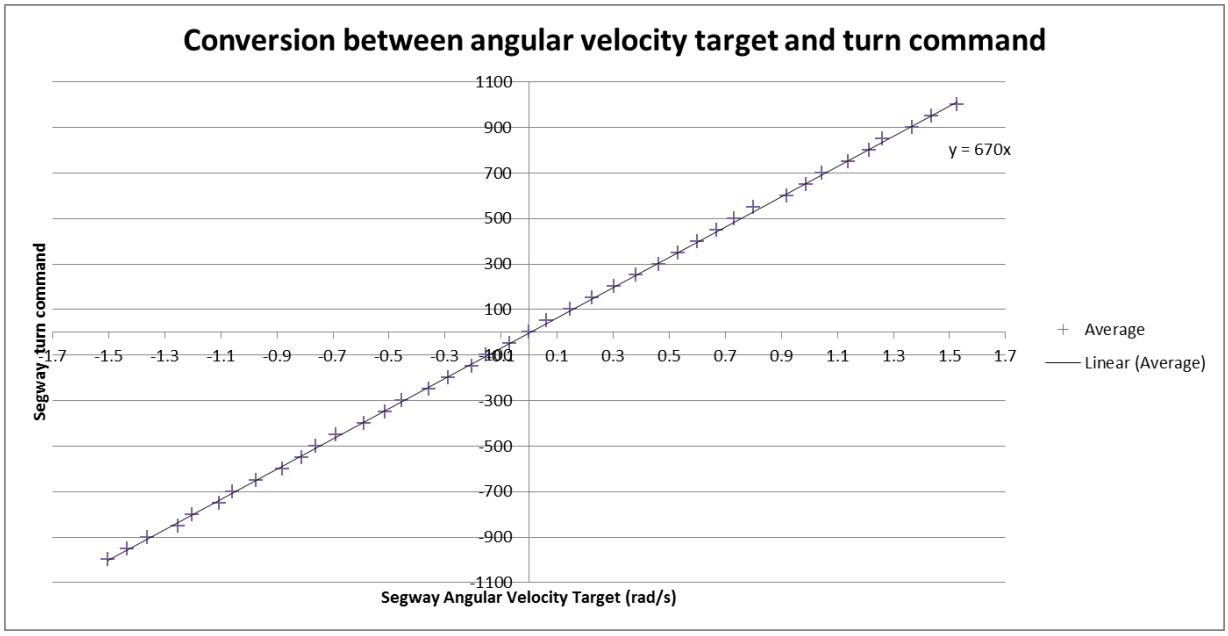

Figure 7.25 Conversion between angular velocity target and required turn command

The Segway configuration parameters for this project are summarised in Table 7.1. The gain schedule is set to 0 indicating the 'light' controller setting. The maximum acceleration, linear velocity and angular velocity scales were set to 0.5 to limit the Segway. As there was minimal wheel slippage due to acceleration and deceleration the current limiting scale was left at maximum.

Table 7.1 Segway configuration parameters

Parameter	Numerical Value
Gain Schedule	0
Max Acceleration Scale	0.5
Max Linear Velocity Scale	0.5
Max Angular Velocity Scale	0.5
Current Limit Scale	1

7.3 Localisation Testing

The navigation system uses landmarks and odometry for localisation within an environment. Landmarks are fixed locations the Segway can find within the environment. The landmark detection algorithm was tested by moving the Segway along the corridor from position A to position C as depicted in Figure 7.26. The seven doors in the environment are labled from 1 to 7 as depicted in Figure 7.26.

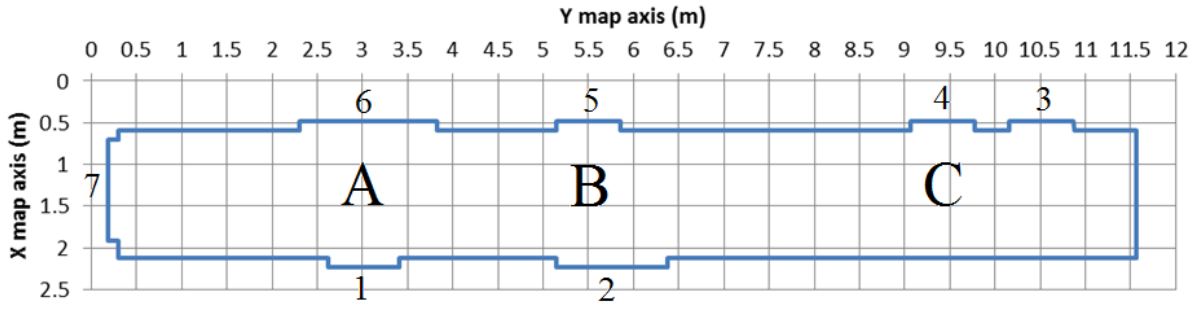

Figure 7.26 Environment map

A graphic indicator of extracted lines and landmarks was temporarily added to the laser data output on the GUI. A blue line is added on top of datapoints that have been associated to a line. Each line is given a number to show how many lines have been found in an environment. The coordinates of a found landmark are depicted by a black circle. Each landmark also displays the type of landmark (door or corner) along with a number indicating how many landmarks have been found.

Figure 7.27 shows landmarks extracted at position A. The The landmark algorithm detected three `Door` landmarks. Both landmarks for door 1 are associated and one landmark associated to door 2. The localisation algorithm was unable to discover the closest landmark for door 2 as the corner of the frame blocked the laser scanner's view of the complete door. This is not a large problem for autonomous operation because as the Segway moves past the door frame the landmark will be discovered.

The doors 5 and 6 were held open to display one limitation of the localisation algorithm. These doors were unable to be found as the landmark detection algorithm searches for parallel lines with close end points. All the doors within the opperating environment have mechanisms that automatically close open doors but if all the doors within the environment were left open, the navigation system would rely on corner landmarks and odometry alone. This limitation and possible solutions are discussed further in Section 8.2

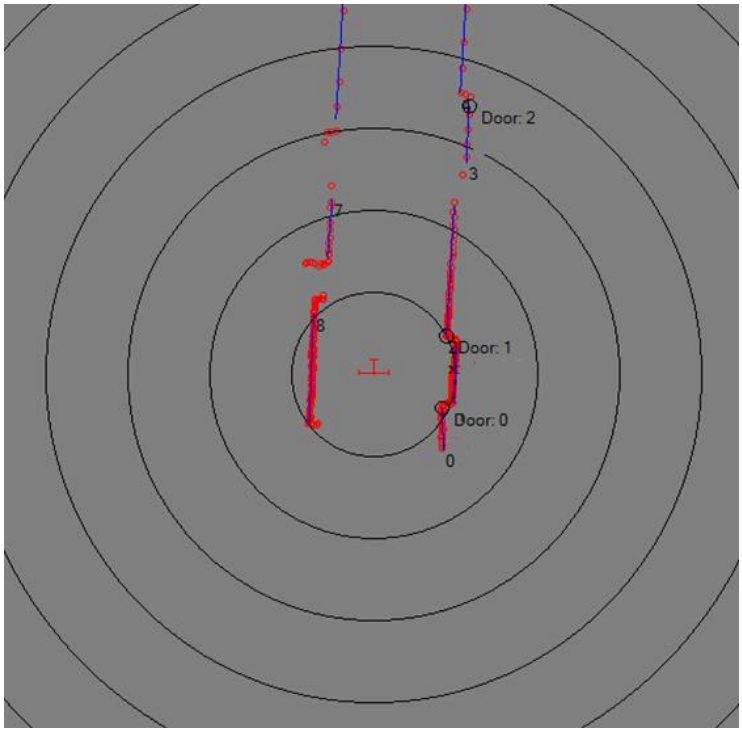

Figure 7.27 Landmarks detected at position A.

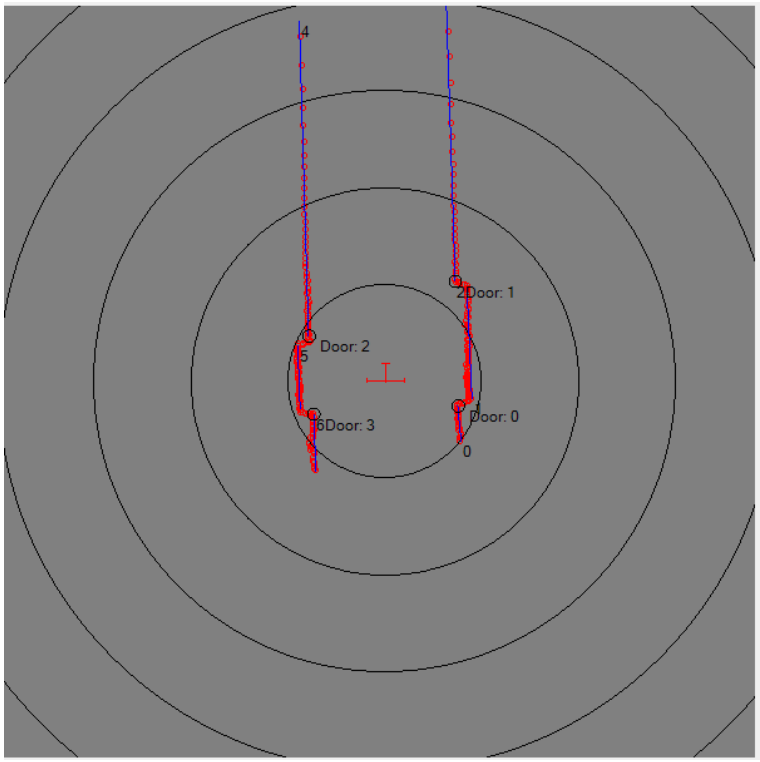

Figure 7.28 Landmarks detected at position B.

At position B four landmarks were found by the landmark algorithm (Figure 7.28). The first two landmarks associate with door 2 and the last two associate with door 5. Door 5 was shut while moving between position A and B. This image shows that `Door` type landmarks are better associated when the Segway is positioned between the two frames as there is no edge to block the view of the laser scanner (as was the case for door 5 in Figure 7.27).

At position C Five landmarks were found by the landmark algorithm (Figure 7.29). Two `Convex` landmarks are associated with the corners of the corridor, two `Door` landmarks are associated with door 4 and one `Door` landmark associated with door 3. Once again the closest frame of door 3 blocks the nearest landmark from being found.

These tests show the landmark algorithm is robust enough to find and associate all landmarks within the operating environment during normal opperation. As the location of landmarks were always static, the difference between the sensed location and actual location can be calculated and used by the navigation algorithm to determine the error in the internal representation of the Segway's location.

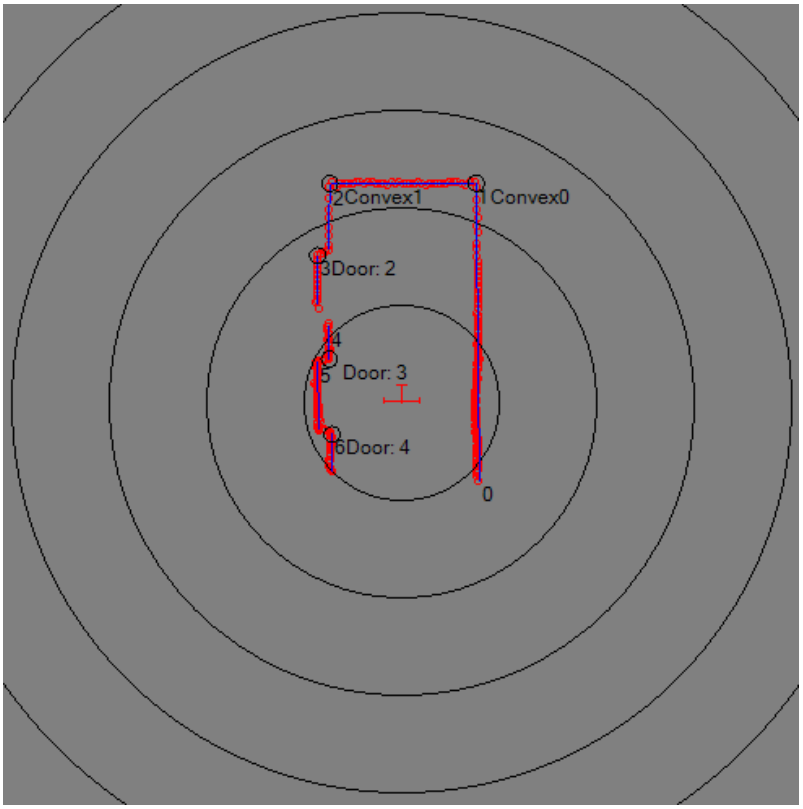

Figure 7.29 Landmarks detected at position C.

7.4 Navigation System Parameters

7.4.1 Direction Sensor

The direction sensor has two parameters, α and β that determine the desired output heading target angle as discussed in Section 5.3.2. The α parameter relates to goal directness and the β parameter relates to obstacle avoidance. As the parameters produce an output dependent on both the input parameters, the α parameter was held constant at 0.5 while the β parameter was varied. The results for β values of 0.5, 1 and 2 are shown in Figure 7.30.

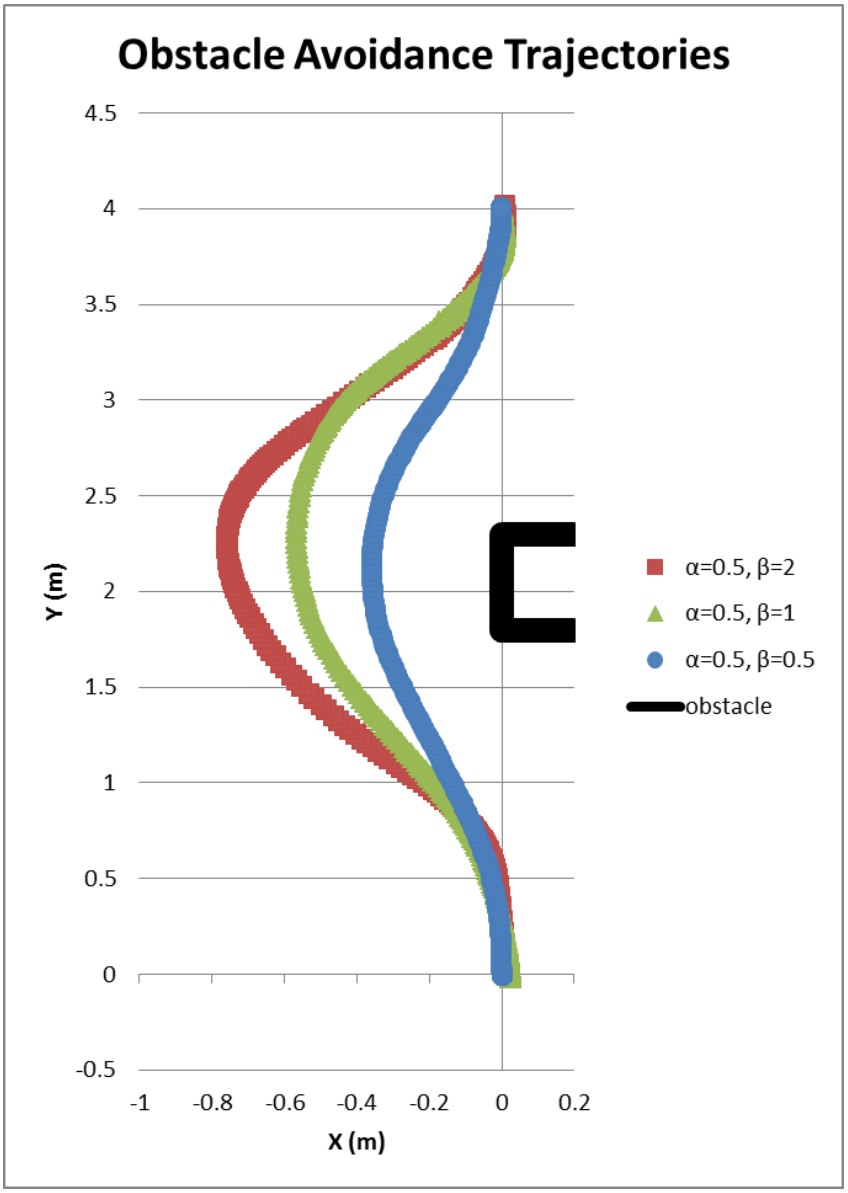

Figure 7.30 Obstacle avoidance trajectories with different β values

The edge of an obstacle was placed in the way of a straight line heading. Figure 7.30 shows the path taken by the centre of the Segway and Figure 7.31 shows the target heading output from the Direction Sensor during the test. The Segway has a radius of 0.35 m. A β value of 0.5 produced a path that missed the obstacle by about 5 cm, a β value of 1 produced a path that missed the obstacle by 25 cm and a β value of 2 produced a path that missed the obstacle by 45 cm.

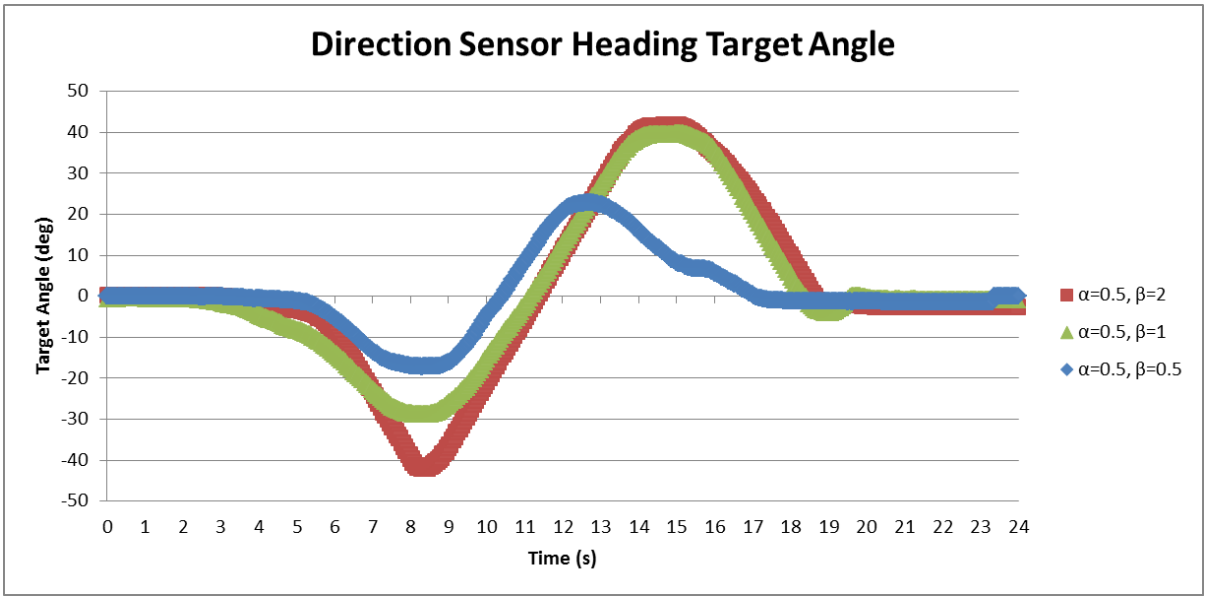

Figure 7.31 Target heading output from the Direction Sensor over time.

Figure 7.31 shows the target heading output from the Direction Sensor for different β values. A β value of 0.5 caused a 20° heading change from a straight line, a β value of 1 caused a 30° heading change and a β value of 2 caused a 40° heading change. Smaller β values create closer paths to the obstacle meaning a shorter time to complete the obstacle avoidance manoeuvre (given a constant velocity). An α value of 0.5 and β value of 1 were chosen as it gave a good compromise between obstacle avoidance and distance travelled.

The maximum distance to obstacles d_{obmax} was set to the maximum range of the SICK LMS100 of 20 m. The number of candidate orientations $N_{\emptyset}$ for the direction sensor algorithm to evaluate is set at 45 to balance computational effort and direction resolution. This gave a direction resolution of 6°. These values are summarised in Table 7.2.

Table 7.2 Direction sensor parameter values

Parameter	Numerical Value
α	0.5
β	1
$\boldsymbol{d_{obmax}}$	20 m
$\boldsymbol{N_{\emptyset}}$	45

7.5 Corridor Environment Tests

7.5.1 Linear Forward Command

For the linear forward command testing, the internal representation of the path travelled is plotted. The actual trajectory the Segway follows is not shown because it is impractical to externally measure the Segway's position while it was moving.

The Segway was commanded to move autonomously from coordinate location (1.5, 1.3) to (6.5, 1.3), a distance of 6 m, with maximum linear velocities of 0.3, 0.5 and 0.75 m/s. Each test was conducted 8 times.

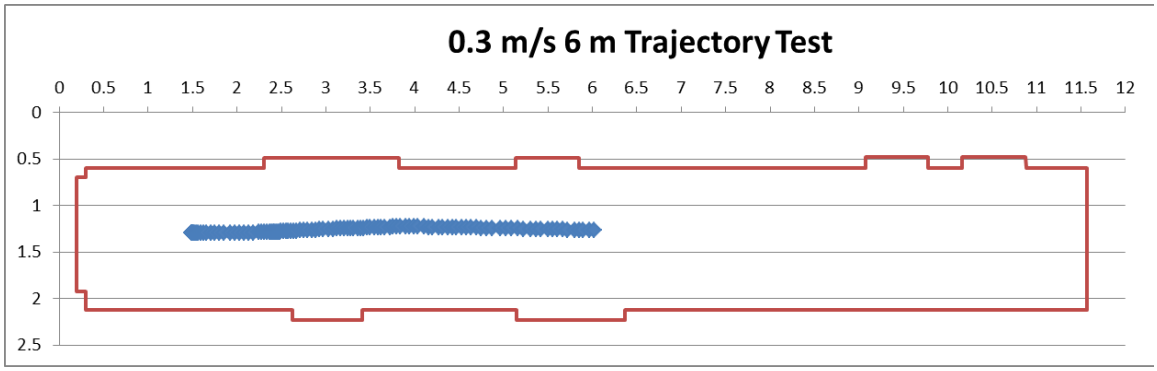

Figure 7.32 X,Y coordinates of the Segway during 0.3 m/s 6 m trajectory test

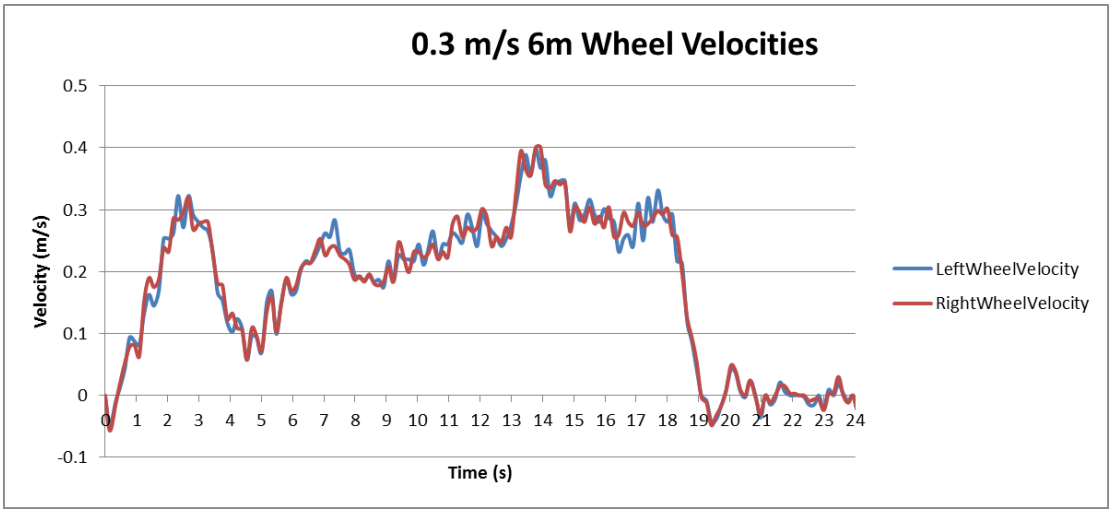

Figure 7.33 Wheel velocity profiles for 0.3 m/s 6 m trajectory test

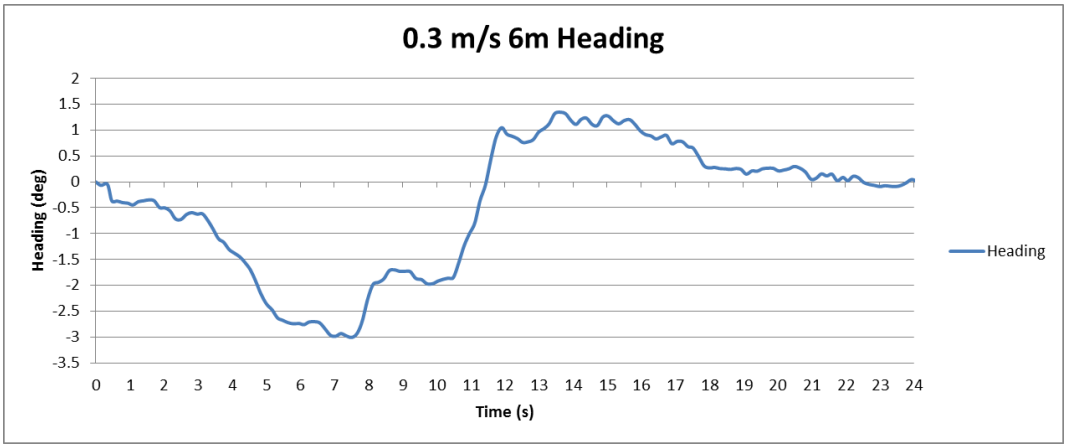

Figure 7.34 Internal heading during 0.3 m/s test over 6 m

Figure 7.32 shows the internal position representation of the Segway in the corridor during the 0.3 m/s test. The trajectory shows that an initial alignment error that caused the Segway to drift towards the left wall. As landmarks are found, this heading error is detected and corrected with an adjustment towards the centre of the corridor.

Figure 7.33 shows the wheel velocities during the test. The Segway follows the same acceleration and deceleration profiles as expected from the open environment tests. Differences in the wheel velocities can be seen as the navigation system corrects for detected position errors.

Figure 7.34 shows the internal heading of the Segway. The initial starting heading was set to be ideally 0°. During operation a heading error of -0.4° was discovered when the first landmark was found. As more landmarks were found during the 3-5 second range, the heading error increased to -3° and the navigation system attempted to correct the error. This error can be attributed to the initial alignment of the Segway not being the same as the initial internal heading. The navigation system set a heading of 1° at 12 s to correct the error in position and brought the target heading back towards 0° at the end of the test.

For the 0.5 m/s tests (results shown in Figure 7.35 to Figure 7.37) the initial alignment error was minimal allowing the navigation system to make fewer heading corrections during the experiment. Again the wheel velocity shows similar acceleration and deceleration profiles, with a negative velocity to start forward movement and negative velocity towards the end of the test after deceleration to stabilise the platform. The wheel velocity peaked at 0.55 m/s

during acceleration and at 0.7 m/s 10.5 s into the test. The internal heading started at 0 and decreased to -0.5° as landmarks were associated.

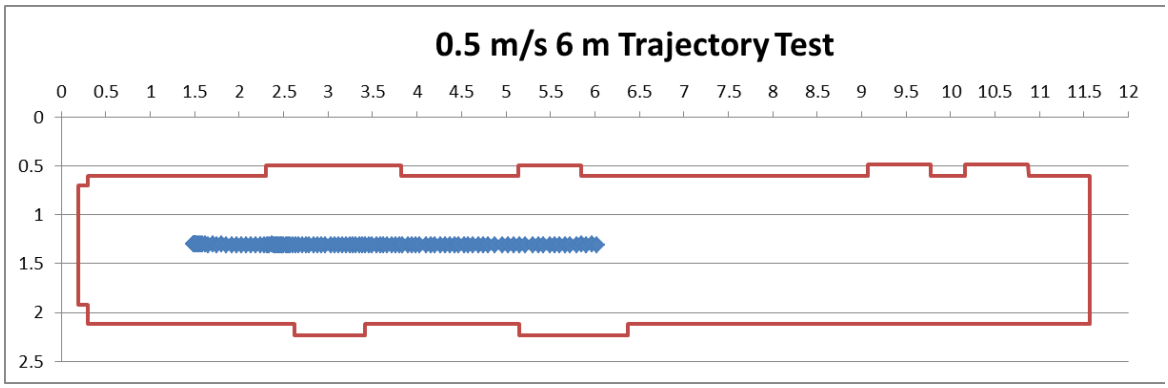

Figure 7.35 X,Y coordinates of the Segway during 0.5 m/s 6 m trajectory test

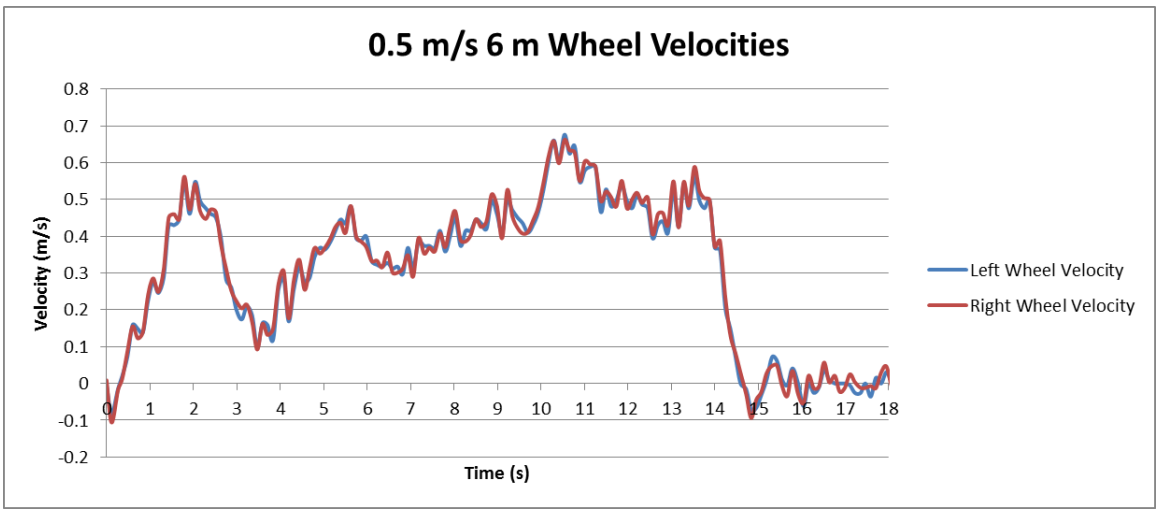

Figure 7.36 Wheel velocity profiles for 0.5 m/s 6 m trajectory test

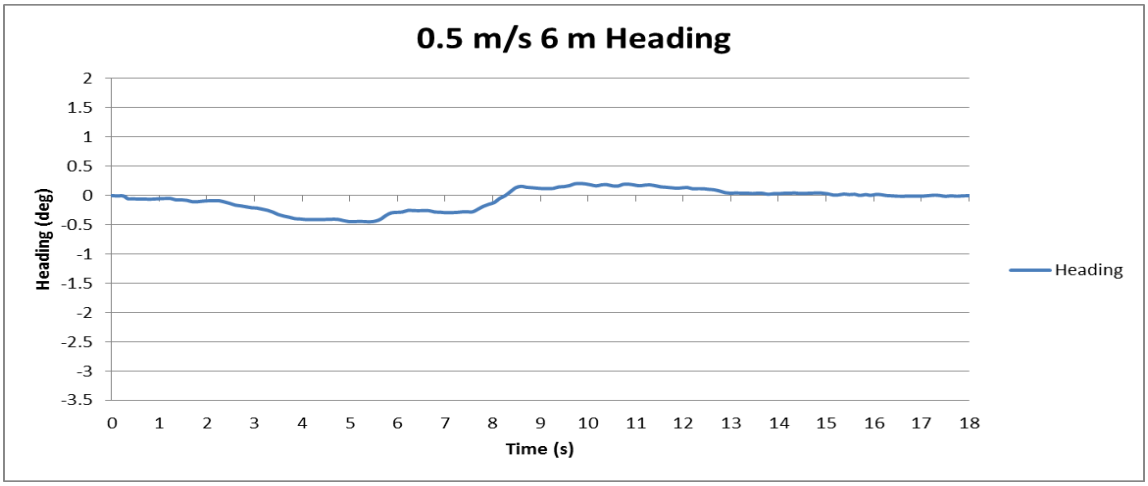

Figure 7.37 Internal heading during 0.5 m/s test over 6 m

Figure 7.38 shows the resulting position errors between actual position and internal position for all 8 tests at the different velocities. These results show a better performance at 0.3 m/s when compared to 0.5 m/s and 0.75 m/s with an average X error of -0.003 m with a standard deviation of 0.013 m and an average Y error of -0.002 m with a standard deviation of 0.002 m compared to an average errors of 0.05 m (std. 0.008 m) in the X direction and 0.001 m (std. of 0.023 m) Y direction for the 0.5 m/s tests and 0.001 m (std. 0.008) in X direction and 0.003 m (std. 0.027 m) in the Y direction.

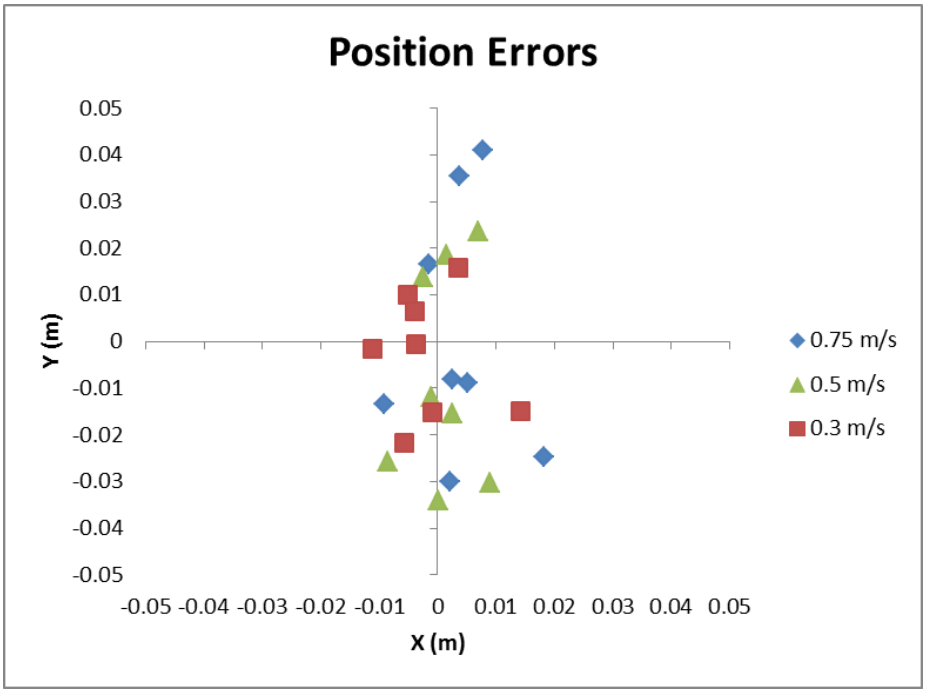

Figure 7.38 Position errors

There is a larger spread in error in the Y direction (7.5 cm) when compared to the X direction (2.9 cm). Errors in the X direction during straight line tests can be due to distance travelled errors while errors in the Y direction can occur from heading errors. The systematic Y error could be caused by inaccuracies in calculating the heading error of the Segway. The heading error is calculated by calculating the difference in angle of the lines that make a landmark. The landmarks are found with parallel and perpendicular lines but allow an error of up to 10° to account for noisy and slightly inaccurate lines being extracted from the laser range finder data.

Chapter 8 Discussion

This chapter discusses the objectives achieved by this project and then outlines future improvements to this project.

8.1 Objectives Achieved

A number of robotic development environments were investigated including Player/Stage, Robot Operating System (ROS), Open Control Robot Software (OROCOS) and Microsoft Robotics Developer Studio (MRDS). Microsoft Robotics Developer Studio was chosen as the development environment.

Different navigation techniques and architectures were discussed and compared. A hybrid navigation architecture, combining both reactive and deliberative control, developed at Victoria University was chosen as the navigation architecture. The hybrid navigation combined an A* path planner with an occupancy grid and used a modified dynamic window and direction sensor to navigate the Segway's environment.

Three common range finders were compared; the SICK LMS100, SICK LMS200 and the Hokuyo URG. The SICK LMS100 was chosen as the laser range finder for this project.

A MRDS service was written in C# to start the SICK LMS100 laser range finder and receive distance measurements. The service is able to post update messages to subscribers when new distance measurements are received.

The characteristics of the SICK LMS100 range finder were examined and discussed. Characteristics tested were the settling time and distance measurements to three surfaces with different reflective properties commonly found in the expected operating environment. Black, white and glass surfaces were used as surfaces representing the extremes within the operating environment.

A service was created to control the Segway platform. The service is designed to be a generic service that could be used for any future projects. It sends control messages at 20 Hz and receives update messages at 100 Hz.

The movement characteristics of the Segway were investigated. The pitch angle/acceleration relationship and wheel velocity profiles during acceleration and deceleration were obtained.

Distance measurements were obtained from the Segway odometer counts and conversion factors were calibrated to reduce the errors. Equations were derived to obtain the distance and heading travelled from the individual displacements measured by each of the Segway's wheels.

The hybrid navigation architecture was implemented in the Segway Navigation service which subscribed to services controlling the SICK LMS100 and Segway platform.

A graphical user interface was also developed as a service which can be run on a remote computer to monitor and update the navigation system properties.

8.2 Future Work

8.2.1 Additional Sensors

When no landmarks can be found the current sensor error is set to zero, meaning that the Segway is relying completely on odometry for localisation. This is not desirable as localisation using odometry alone accumulates error over time due to small wheel slippage or incorrect calibration being emphasized over long travel periods. Implementation of more sensors such as those mentioned below could improve localisation and navigation capabilities.

Lower Rangefinder

The Segway platform does not have the ability to sense objects lower than 1.1 m where the SICK LMS100 has been mounted for this project. This project has assumed all obstacles will be larger and able to be sensed by the laser rangefinder. The Segway's control system cannot tell the difference between an obstacle at wheel level or an external force acting on the balanced system. This causes the Segway platform to continuously run into and bounce off lower obstacles and causes system instability at higher speeds. Two Hokuyo URG laser range finders, discussed in Section 3.2.3, could be mounted at wheel level at the front and back of

the Segway to detect such low objects. Short range Sharp IR distance sensors or ultrasonic distance sensors could also be used to detect low objects.

Video camera

A video camera could be added to the Segway platform and connected to the control computer. A video camera would allow additional object detection and avoidance that a laser range finder could not detect. Along with obstacle avoidance, a video camera could be used to implement object tracking and aid in localisation by identifying and associating visual landmarks.

Compass

A compass would be useful as an absolute heading reference but may suffer from interference by magnetic fields generated from objects within the operating environment. Alternatively an inertial measurement unit (IMU) which contains a gyrocompass could be used.

GPS

A global positioning system (GPS) unit could be used to assist with localisation but these tend not to operate well in many indoor environments without complex external receivers.

8.2.2 Higher Level Control

The current system is capable of moving from one location to another location. A higher level control service could be created which could intelligently select tasks and goal locations to move to. This service could implement functions such as roaming the corridors and automatically returning to a charging point when battery levels become low.

The Segway UI service could be extended to give new goal locations and new maps as the Segway travels through different corridor environments.

8.2.3 System Improvements

Along with additional sensors, the overall system would benefit from a device that could hold the Segway upright when not powered. The addition of two caster wheels (one front and one back) or similar bracing devices that the control system could lift off or drop to the ground when transitioning from between balance and tractor mode. This would allow the Segway platform to power down without falling over and requiring human assistance. The system would have to ensure that the additional ground contact points were lifted before balance mode becomes active.

8.3 Summary

The result of this project is a Segway platform that can execute motion instructions using a hybrid navigation algorithm implemented in MRDS. In the corridor environment the control system was capable of identifying door and corner landmarks and guided the Segway to within 7 cm of the goal location.

Generic services for the SICK LMS100 and Segway platform were made that can be extended and reused for other robots developed with MRDS. The navigation system was implemented in a single service that subscribed to the SICK LMS100 and Segway platform services. A user interface service was also created allowing user interaction with the system.

Overall the project was a success, meeting its objectives and providing a system that can be expanded upon in future projects.

Appendix: CD Contents

The attached CD contains the following:

- Soft copy of this thesis
- Software C# MRDS Project services
  - Segway Native Wrapper
  - Segway Base Service
  - SICK LRF Scanner Service
  - Segway Navigation Service
  - Segway UI Service

Bibliography

Albus, J. S. (2002). *Intelligent Systems: Architecture, Design and Control.*

Anderson, C., Axelrod, B., Philip Case, J., Choi, J., Engel, M., Gupta, G., et al. (2008). Mobile Manipulation - A Challenge In Integration.

Arkin, R. C. (1987). Motor chema based navigation for a mobile robot: An approach to programming by behavior. *IEEE International Conference on Robotics and Automation*, (pp. 264-271).

Arkin, R. C. (1989). Motor Schema-Based Mobile Robot. *The International Jurnal of Robotics Research, 8*(4), pp. 92-112.

Arkin, R. C. (1998). *Behavior Based Robotics.* Cambridge Massachusetts: MIT Press.

Arkin, R. C., & Murphy, R. R. (1990). Autonomous Navigation in a Manufacturing Environment. *IEEE Transactions on Robotics and Automation*, (pp. 445-454).

Axelrod, B. (2011, February 16). *Service Oriented Architectures - Two Leading Systems, MRDS and ROS, Point to the Future of Robotics*. Retrieved 2012, from Robot magazine: http://www.botmag.com/index.php/service-oriented-architectures-two-leading-systems-mrds-and-ros-point-to-the-future-of-robotics

Bailey, P., Beckler, M., Hoglund, R., & Saxton, J. (2008). 2D Simultaneous Localization and Mapping.

Balch, T., Boone, G., Collins, T., Forbes, H., MacKenzie, D., & Santamaria, J. (1995). "Io, Ganmede and Calisto" - A Multiagent Robot Trash Collecting Team. *AI Magazine, 16*(2), pp. 10-16.

Blanco, J. L., Gonzalez, J., & Fernandez, J. A. (2012). An Alternative to the Mahalanobis Distance for Determining Optimal Correspondences in Data Association. *IEEE transactions on robotics, 28*(4), pp. 980-986.

Borges, G. A., & Aldon, M. J. (2000). A split-and-merge segmentation algorithm for line extraction in 2D range images. *Proceedings 15th International Conference on Pattern Recognition, 1*, pp. 441-444.

Brooks, R. (1985). A Robust Layered Control System for a Mobile Robot. Cambridge MA: MIT AI Memo.

Brooks, R. A., & Stein, L. (1989). Building Brains for Bodies. *Autonomous Robots, 1*(1), pp. 7-25.

Brooks, R., Aryananda, L., Edsinger, A., Fitzpatrick, P., Kemp, C., O'Reilly, U., et al. (2004). Sensing and manipulating built-for-human environments. *International Jurnal of Humanoid Robotics*.

Bruyninckx, H. (2001). Open Robot Control Software: The OROCOS project. *Proceedings of the 2001 International Conference on Robotics and Automation.* Seoul, Korea.

Cameron, J., MacKenzie, D., Ward, K., Arkin, R., & Book, W. (1993). Reative Control for Mobile Manipulation. *Processing of the International Conference on Robotics and Automation*, (pp. 228-235). Atlanta,GA.

Castellanos, J., & Tadoos, J. (1996). Laser-based Segmentation and Localization for a Mobile Robot. *In Robotics and Manufacturing: Recent Trends in Research and Applications, 6*.

Cepedia, J. S., Chaimowicz, L., & Soto, R. (2010). Exploring Microsoft Robotics Studio as a Mechanism for Service-Oriented Robotics. *2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*.

Chand, P. (2011). Development of an Artificial Intelligence System for the Instruction and Control of Co-operating Mobile Robots.

Chand, P., & Carnegie, D. A. (2011). Development of a navigation system for heterogeneous mobile robots. *Int. J. of Intelligent Systems Technologies and Applications, 10*(3), pp. 250 - 278.

Chen, Y. (2006). Service-Oriented Computing in Recomposable Embedded Systems. *Workshop on Dependabilitys in Robotics and Autonomous Systems*, (pp. 15-19). Tuscon, AZ.

Chen, Y. (2008). On Robotics Applications in Service-Oriented Architecture. *28th International Conference on Distributed Computing Systems Workshops*, (pp. 551-556).

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., et al. (2005). *Principles of Robot Motion.* Boston: MIT Press.

Diftler, M. A., Ambrose, R. O., Tyree, K. S., & Goza, S. M. (2004). A mobile autonomous humanoid assistant. *4th IEEE/RAS International Conference on Humanoid Robots*.

Elkady, A., & Sobh, T. (2012). Robotics Middleware: A comprehensive literature survey and attribute-based biblography. *Journal of Robotics*.

Fielding, R. T., & Taylor, R. N. (2005). Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*, (pp. 115-150).

Fischler, M. A., & Bolles, R. C. (1981, June). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM, 24*(6), pp. 381 - 395.

Forsyth, D. A., & Ponce, J. (2002). *Computer Vision: A Modern Approach.* Prentice Hall.

Fox, D., Burgard, W., & Thrun, S. (1997). The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine, 4*(1), pp. 23-33.

Fox, D., Burgard, W., & Thrun, S. (1997). The Dynamic Window Approach to Collision Avoidance. *IEEE Robotics and Automation Magazine*, pp. 23-33.

Garage, W. (2012). *ROS - Introduction*. Retrieved 2012, from www.ros.org/wiki/ros/introduction

Gat, E. (1991). Relible Goal Directed Reactive Control of Autonomous Mobile Robots.

Gat, E. (1992). Integrating Planning and Reaction in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots. *Proceedings of the AAAI.*

Gates, W. (2007). A Robot in Every Home. *Scientific American Magazine*.

Gerkey, B. P., Vaughan, R. T., & Howard, A. (2003). The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. *Proceedings of the International Conference on Advanced Robotics.* Coimbra, Portugal.

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

Henning, M. (2006, June 1). *The rise and fall of CORBA*. Retrieved 2012, from http://queue.acm.org/detail.cfm?id=1142044

Horswill, I. (1993). Polly, A Vision-Based Artificial Agent. *Proceedings of the AAAI-93*, (pp. 824-829). Washington, DC.

Informer Technologies Inc. (2012). *SICK SOPAS Engineering Tool*. Retrieved May 2012, from http://sick-sopas-engineering-tool.software.informer.com/

Jackson, J. (2007, Dec.). Microsoft robotics studio: A technical introduction. *Robotics & Automation Magazine, IEEE, 14*(4), pp. 82 - 87.

Johns, K., & Taylor, T. (2008). *Professional Microsoft Robotics Developer Studio.* Wrox Press Ltd.

Junior, V. G., Parikh, S. P., & Junior, J. O. (2006). Hybrid Deliberative/reactive architecture for human-robot interaction. *ABCM Symposium Series in Mechatronics, 2*, pp. 563-570.

Kapach, K., Giorini, P., & Mylopoulos, J. (2007). *Adaptive weighted average sensor fustion algorithms for mobile robots.*

Kramer, J., & Scheutz, M. (2007). Development environments for autonomous mobile robots: A servey. *Autonomous Robots, 22*(2), pp. 101-132.

Lee-Johnson, C. P. (2004). The Development of a Control System for an Autonomous Mobile Robot.

Linux Devices. (2008, 07 29). *Review of robotic software platforms*. Retrieved 2012, from LinuxForDevices: http://linuxdevices.com/articles/AT9631072539.html

Liu, Z., Jin, Y., Cui, Y., & Wang, Q. (2001). Design and implementation of a line simplification algorithm for network measurement system. *Proceedings of IEEE IC-BNMT2011*, (pp. 412-416).

Mataric, M. (1992, June). Integration of Representation into Goal-Driven Behavior-Based Robots. *IEEE Transactions on Robotics and Automation, 8*(3), pp. 304-312.

Mc Guire, A. R., Henriques, B. S., Nguyen, H. C., Jensen, K. F., Vinther, K., & Jespersen, R. (2009). *Trajectory Planning and Control for a Segway RMP*. Aalborg.

McClymont, J. (2011). MARVIN User Manual. Victoria University of Wellington.

Michal, D. S. (2010). *A comparison of development environments for mobile autonomous robots: Player/Stage/Gazebo vs. Microsoft robotics developer studio.* The Universtiy of Alabama, Huntsville.

Microsoft. (2010). *CCR Ports and PortSets*. Retrieved 12 10, 2012, from http://msdn.microsoft.com/en-us/library/bb648755.aspx

Microsoft. (2012). *Generic Differential Drive.* Retrieved 2012, from MSDN: http://msdn.microsoft.com/en-us/library/dd145254.aspx

Microsoft. (2012). *Microsoft Robotics Studio*. Retrieved 2012, from http://msdn2.microsoft.com/en-us/robotics/default.aspx

Nehmzow, U. (2003). *Mobile Robotics: A Practical Introduction (Second Edition).* London.

Nguyen, V., Martinelli, A., Tomatis, N., & Siegwart, R. (2005). A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics. *International Conference on Intelligent Robots and Systems*, (pp. 1929-1934). Switzerland.

Oceanchip. (2009). *FTD245BM Datasheet.* Retrieved 2012, from http://www.oceanchip.com/datasheets/FTD245BM.pdf

Pearl, J. (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. *Reading Massachusetts: Addison-Wesley*.

Pirjanian, P. (2005). Challenges for standards for consumer robotics. *Advanced Robotics and its Social Impacts, 2005. IEEE Workshop on*, (pp. 260-264). Pasadena.

Player. (2010). *The Player Project*. Retrieved 2012, from http://playerstage.sourceforge.net/

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., et al. (2009). ROS: An open-source Robot Operating System. *Proceedings of the Workshop on Open Source Software (IRCA '09).*

Riisgaard, S. (2005). *SLAM for Dummies. A tutorial approach to Simultaneous Localization and Mapping Personal Comments.*

Riisgaard, S., & Blas, M. R. (2005). SLAM for Dummies. A Tutorial Approach to Simultaneous Localization and Mapping.

Rudan, J., Tuza, Z., & Szederkenyi, G. (2010). *Using LMS-100 Laser Rangefinder for Metric Map Building.*

Sauer, C. T., Brugger, H., Hofer, E. P., & Tibken, B. (2001). Odometry Error Correction by Sensor Fustion for Autonomous Mobile Robot Navigation. *IEEE Instrumentation and Measurement Technology Conference.* Budapest, Hungary.

Segway Inc. (2009). *Segway Robotic Mobility Platform User Guide.*

Segway Inc. (2012). *Segway RMP Models*. Retrieved Dec 2012, from http://rmp.segway.com/discontinued-models/

Segway Inc. (2012). *Segway® RMP 200/ATV Specifications*. Retrieved 2012, from http://rmp.segway.com/downloads/RMP_200_Specsheet.pdf

Segway Inc. (2009). *Segway Robotic Mobility Platform (RMP) Interface Guide.*

Segway Inc. (2012, Dec). *About Segway*. Retrieved Dec 2012, from http://www.segway.com/about-segway/

SICK Inc. (2012). Laser MEasurment Systems - LMS100 Product Family Information.

SICK Inc. (2012). Telegrams for Configuring and Operating the LMS1xx.

SICK Inc. (2003). *LMS200 Laser Measurement Systems Technical Description.* Retrieved 2012, from http://www.sick-automation.ru/images/File/pdf/LMS%20Technical%20Description.pdf

SICK Inc. (2012). *Laser Measurement Sensors of the LMS1XX Product Family Operating Instructions.*

Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2004). *Introduction to Autonomous Mobile Robots second edition.* Cambridge, Massachusetts: The MIT Press.

Soetens, P. (2010). *The Orocos Toolchain*. Retrieved 2012, from http://www.orocos.org/toolchain

Talwatta, B. K. (2012). The Implementation of a Hierarchical Hybrid Navigation System for a Mobile Robotic Vehicle.

Tsai, W. T., Huang, Q., & Sun, X. (2008). A Collaborative Service-Oriented Simulation Framework with Microsoft Robotic Studio. *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, (pp. 263 - 270).

Ulrich, I., & Borenstein, J. (1998). VFH+: reliable obstical avoidance for fast mobile robots. *Proceedings of the IEEE International Conference on Robotis and Automation*, pp. 1572-1577.

van Dam, J., Krose, B., & Groen, F. (1996). *Neural Network Applications in Sensor Fustion for an Autonomous Mobile Robot.* University of Amsterdam.

Victorino, A. C., Rives, P., & Borrelly, J. J. (2000). Localization and map building using a sensor-based control strategy. *Intelligent robots and systems., 2*, pp. 937-942.

Vorlesungen. (2010). *Deliberative vs Reactive control*. Retrieved 2012, from http://www.informatik.uni-leipzig.de/~der/Vorlesungen/ROBOTIK/Deliberative%20vs_%20reactive%20control.htm

Wiliams, H. (2012). Integration of Learning Classifier Systems with Simultaneous Localisation and Mapping ofr autonomous robotics. *IEEE Congress on Evolutionary Computation*, (pp. 1-8). Wellington, New Zealand.

Williams, M. L., Wilson, R. C., & Hancock, E. R. (1997). Multi-sensor fusion with Bayesian inference. *Computer Analysis of Images and Patterns; Lecture Notes in Computer Science*, (pp. 25-32).

Wu, H., Seigel, M., Stiefelhagen, R., & Yang, J. (2002). *Sensor Fusion using Dempster-Shafer Theory*. Anchorage, AK, USA.

Ye, C., & Borenstein, J. (2002). Characterization of a 2D laser scanner for mobile robot obstacle negotiation. *IEEE International Conference on Robotics and Automation*, (pp. 2512-2518).