Coordinated LEGO

Segways

Steven J. Witzand

A thesis submitted for the degree of

Bachelor of Engineering

School of Information Technology & Electrical Engineering

University College

The University of New South Wales

Australian Defence Force Academy

November 2009

Abstract

Self-reconfigurable robots provide a future for robotics by being able to modify and change their physical shape to suit their current operational needs. These robots show great promise for future applications, designed to be robust and versatile and capable of achieving multiple tasks [1]. The overall goal of this project is to create a software architecture for coordinated self-reconfigurable robots, and carry out initial research into capabilities of such robots using LEGO Mindstorms technology [2]. Key characteristics of self-reconfigurable robots are their ability to communicate, synchronise the behaviour of multiple systems, perform numerous tasks simultaneously and adapt their behaviour to suit their current operational environment. This project will focus on designing and programming the initial framework required to achieve such a robot. This will be done by focusing in the areas of communication and synchronisation of multiple robots to provide them with the means of joint tasking and joint decision making, the implementation of a behavioural system that allows the robot to make its own decisions and adapt to changes in its environment and parallel programming to allow all these systems to run simultaneously. In this project we have demonstrated that by implementing the behavioural system we have provided the capability to allow the robot to make its own decisions without operator intervention. This project has also confirmed that the communication system used to link multiple robots together can give these robots the means of sharing sensor information between themselves allowing them to get a better overall picture on their environment, and also the ability to control each other in a master/slave configuration. This project has also proposed a method to keep two balancing robots at a set distance from each other by using a distance control system, which can enable balancing robots to come together gradually, with the possibility of connecting their structures into a modular system. This achieves the initial stages of synchronising the movement of multiple robots to allow them to work as a group of robots rather than individuals. By successfully implementing the communication, synchronisation and behavioural systems into the robot, all of which run concurrently, we have provided a scalable software platform which can be built upon to achieve the ultimate goal of creating a self-reconfigurable robot.

Declaration

I hereby declare that this submission is my own work and to the best of my knowledge it contains no material previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by colleagues, with whom I have worked at UNSW or elsewhere, during my candidature, is fully acknowledged. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Signature Date

Acknowledgments

This thesis would not be at the stage it was today if it was not for the assistance of the people I have around me. Firstly I would like to thank my supervisors, Valeri Ougrinovski and Kathryn Merrick, their guidance and feedback throughout the year has been invaluable and a high amount of gratitude goes out to you from me. Secondly to UNSW@ADFA for providing me the resources and knowledge over the years to make me the engineer I am today. To my peers and classmates, thank you for your constant support inside and outside of ADFA. You have made the hard times bearable and the good times amazing. To my mother, Robyn Attwood, thank you for putting in the time to proof read this thesis even though you had no idea what it was about. Lastly to my partner Rachel Penno, your support throughout this degree and during this thesis has been incredible. Allowing me to escape from the world of engineering each night has been a godsend and I thank you for seeing me through from start to finish.

Contents

List of Figures

List of Tables

Glossary

Actuator:
A mechanical device that moves a system, for example a motor.

Arbitrator:
A mechanism which determines that behaviours should become active in a robot.

Autonomous robot:
Can perform tasks without the need for continuous human control.

Balance Controller:
A control system designed for two-wheeled, balancing robots to keep them in the upright position by measuring body angle and velocity and motor angle and velocity, then calculating the power required to keep the robot upright.

Ballistic behaviour:
Is triggered once and follows a predicable process, for example obstacle avoidance.

Behaviour:
An action or reaction of a robot typically initiated by some sort of trigger.

Behavioural System:
Allows the robot to make its own decisions without operator intervention.

Bluetooth:
An open wireless protocol for exchanging data between two systems.

Boolean:
A variable that can consist of two states, true or false, 0 or 1.

C/C++:
A general purpose, object-orientated programming language.

Centrifugal force:
The outward force on a body that is following a curved path.

Data logging:
The process of recording sequential data.

Delay time:
The time taken for the step response to reach 50 percent of its final value.

Distance Controller:
A control system designed for a string of two-wheeled balancing robots to keep the following robot at a set distance from the leading robot.

Disturbance (Controller):
Applying physical force to a controller, such as prodding a balancing robot.

Eigenvalue:
The state-space equivalent of the poles of a transfer function. Used to determine the stability of a system.

Electro Optical Proximity Detector (EOPD):
A sensor which uses an internal light source to detect the presence of a target or determine changes in distance to a target.

Following robot:
A robot which follows a leading robot at a set distance.

GELway:
The name given to the two-wheeled balancing robot designed for this project

Gyro Angle:
In terms of a balancing robot the angle at which the body is tilted, measured in degrees.

Gyro Sensor drift:
The effect of a gyro sensor offset decaying exponentially over time.

Gyro Sensor offset:
The value read from a stationary gyro sensor.

Gyro Sensor/Gyroscope Sensor:
A sensor which enables a robot to measure rotation and direction of rotation.

Gyro Velocity:
In terms of a balancing robot the velocity at which the body has rotated, measured in degrees per second.

Heterogeneous robot:
Robots of a diverse range that are not all identical.

High priority behaviour:
A behaviour that suppresses lower level behaviours and is typically triggered infrequently.

Integrated Development Environment (Eclipse):
An interface which allows programmers to browse, edit, compile and debug programming applications.

Java:
A programming language developed by Sun Microsystems.

Java Virtual Machine (JVM):
A virtual machine that runs and executes Java code.

Kinematics:
The study of motion without reference to the forces that cause the motion.

Leading robot:
A robot which leads a string of robots who follow the leading robot autonomously.

LEGO Mindstorms:
A company that sells programmable NXTs.

leJOS:
A Java-based replacement firmware for the NXT.

Light Sensor:
A sensor that enables a robot to distinguish between light and dark by measuring ambient light.

Linear Quadratic Method (LQR):
An optimal state feedback controller.

Low priority behaviour:
A behaviour that typically runs frequently, but is likely to be suppressed by higher level behaviours.

Marvin:
A two-wheeled, balancing robot designed by Johnny Rieper, Bent Bisballe Nyeng, and Kasper Sohn and programmed using leJOS.

Master robot:
A robot that has unidirectional control over one or more slave robots.

Master/Slave connection:
An architecture in which one system (master) controls one or more other systems (slaves).

Matlab:
A numerical computing environment developed by MathWorks that can be used for complex matrix operations and plotting.

Maximum overshoot:
A measurement of how far the controller output rises above the unit-step input.

Modular robot:
A robot that is made of multiple separate systems that combine to form another larger system.

Motor Angle:
The angle at which the wheels are currently positioned, measured in degrees.

Motor Velocity:
The speed at a motor is rotating in degrees per second.

NXT:
A programmable brick built by LEGO Mindstorms that allows programmers to control various sensors and motors.

NXTway-GS:
A two-wheeled, balancing robot designed by Yorihisa Yamamoto, and programmed using nxtOSEK.

Nyquist criterion:
A semi-graphical method that can be used to check the stability of a system.

PID Controller:
A proportional-integral-derivative controller designed to correct the error between a measured variable and a desired set point.

Plant/Controller Plant:
Part of a control system typically made up of a transfer function.

Recursive filter:
A filter that reuses one or more of its outputs as an input signal.

Remote Control connection:
A communications connection that does not require a physical connection between devices.

Rise time:
Time required for the step response to rise from 10 to 90 percent of its final value.

Segway PT:
A self-balancing, personal transportation device developed by Dean Kamen.

Self-reconfigurable:
Something that can alter its physical structure to adapt to change.

Sensor:
An input device to a robotic system that is able to read environmental information or the internal state of a robot.

Servo behaviour:
A behaviour that utilises a feedback controller loop as its control component. For example a distance controller.

Settling time:
Time required for a step response to decrease and stay within a specified percentage of its final value.

Simulink:
A tool developed by MathWorks that is used for modelling, simulating and analysing systems.

Slave robot:
A robot that must follow instructions from a master robot.

State-space matrix:
A representation of a mathematical model of a physical system as a set of input and output state variables.

Steady-state error:
The difference between an applied reference input signal and the final steady-state response of the output.

Step response:
The response of a system to a step input.

Subsumption architecture:
A mutli-level reactive robot architecture that allows behaviours with higher priority to suppress lower level behaviours.

Synchronous Movement:
To move or react at the same time.

Thread:
In programming a thread represents a single sequence of instructions executed in parallel with other sequences.

Transfer function:
A mathematical representation of the relation between the input and output of a system.

Trigger (behaviour):
The event that initiates a behaviour.

Ultrasonic Sensor:
A sensor that enables a robot to sense and measure distances to objects by using ultrasonic signals.

Unit-step response:
The response to a system when an instantaneous change of one unit is applied to the reference input.

Universal Serial Bus (USB):
A physical hardware interface for connecting a computer to another device such as the NXT.

Chapter 1

Introduction

The importance of research into autonomous, self-reconfigurable robots is due to their three promises of being versatile, robust and cost effective. Versatility is achieved through reconfiguring the system to suit different tasks, robustness since any individual robot can be replaced by another robot and cost effective since the robot is capable of performing numerous tasks [1]. The project has been focused on completing the initial stages of creating a self-reconfigurable robot leaving room for expansion through future work.

The selected areas for this project were the communication and synchronisation of multiple robots, autonomous behaviours in robots and the ability for robots to conduct multiple tasks simultaneously. All these elements are important characteristics of self-reconfigurable robots. Communications is important because multiple autonomous robots need a way to can share information between themselves, synchronisation is required to give them the means of travelling in groups rather than individuals, and a behavioural system is required to allow them to adapt to their environment and make their own decisions. All of these processes need to be running concurrently within the robot. These key areas have been separated into the chapters of this thesis and will now be addressed in more detail.

The first stage of the project involved researching and selecting an appropriate physical structure for the robot that would allow us explore the selected key characteristics of reconfigurable robots. A two-wheeled self balancing robot was selected as an appropriate design over a three or four wheeled robot due to the system's unique zero turn radius, small footprint, high tolerance to impulsive forces and stability over slopes [4]. These capabilities make it a desirable selection. Two-wheeled, balancing robots are also relatively well researched in the literature, thus reducing need for research into the control of robot kinematics. Using a well researched platform has an advantage of enabling us to concentrate on the objectives of the project. However the key reason for selecting the two-wheeled,

balancing robot was the fact that it allowed us to explore the capabilities of simultaneous tasking within the robot by adding a high priority task to stabilise the balancing robot. Several existing two-wheeled, balancing robot designs were researched so that we could ensure we were not following past mistakes that had been made and also to critically analyse what functionality we desired for our robot. This research can be found in section 1.4.4 of the Literature Review.

The critical summary at the end of the literature review in section 1.4.7 also provides a potential future for two-wheeled, balancing robots in the modular robotic world. Modular robots are robots made up of a series of small modules that can connect and disconnect from one another to provide a variety of different shapes and movement patterns. These modules typically have limited movement and are generally incapable of providing useful functions when separated from the main system. However using a two-wheeled, balancing robot as part of a modular system allows the modules to have a larger degree of freedom and is capable of operating as an individual system. This is a particular advantage in situations where multiple tasks need to be performed simultaneously, such as mapping out a room or search and rescue missions. Two-wheeled, balancing robots also provide a good sized module for modular reconfigurable systems as additional wheels can make the size of the module impractical, and having only one wheel could cause too much instability in the individual system.

The first milestone of the project was to select a suitable design of the two-wheeled, balancing robot, capable of sustaining an upright position whilst running other tasks concurrently. The equations of motion of two-wheeled robots are well documented (see secion 2.2.3), but to get a good understanding of the properties of the plant it was decided to carry out the derivation of these equations from the first principles. The derivation of the motion equations allowed us to construct a simulated model of the robot and investigate properties of various control systems using Simulink. The work on this part of the project is described in Chapter 2 of the thesis.

The project's next milestone involved the implementation of a behavioural system on the robot. The behavioural system allows the robot to make autonomous decisions based on sensor or Bluetooth inputs. These behaviours are typically made up of two parts: a triggering mechanism, which activates the behaviour, and an action to carry out once triggered. A simple behaviour that was built and tested on the robot was an obstacle avoidance behaviour. It used an ultrasonic sensor to measure distances, and if a measured distance was less than a desired threshold, it would reverse and turn the robot away. To process all the behaviours a subsumption architecture arbitration system was used and designed in such a way that robot behaviours could easily be added and removed. The provided a scalable platform that will allow future functionality to be added to the robot. Information relating to the behavioural system has been covered in Chapter 3 of this thesis.

The third milestone focused on developing a communication system to enable the robot to communicate with another robot and external Bluetooth devices. A master/slave model was chosen for the robots, so that we could designate one robot, the master, to communicate with the operator as well as another balancing robot, the slave. This system used a modified mobile phone application to send commands to the robot, which could then be interpreted as actions, allowing for remote control of the balancing robot. The project used the Bluetooth features of the LEGO Mindstorms NXT brick, which allowed us to open two communication links: one which connected an operator to the master robot and the other which connect the master and the slave robots. The communications platform allowed us to share sensor information between the robots, providing a better overall view of the robot's environment. Chapter 4 of this thesis covers this part of the project.

The last milestone of the project was to synchronise the movements of two balancing robots which would provide them the capability to follow each other in a straight line. The synchronisation of the robots was one of the more difficult tasks taken on in the project and involves getting the master and the slave robot to follow each other in a straight line, and move around autonomously, without an operator intervention. To achieve this, a distance control system was developed that used readings from an ultrasonic sensor to correct the distance between the two robots. Details of the distance control system are covered in Chapter 5 of this thesis.

The milestones of this project provide the initial stages to developing robots that are capable of self-reconfiguration. Not only have they all been achieved but have been done so in a scalable way to allow for future work to be carried out on the robot to improve its functionality and take it to the next step towards self-reconfiguration.

1.1 Need for Research

The majority of previous work on robots today involves programming them to perform a small number of fixed task and are incapable of solving new problems, even if their physical structure is able to, unless reprogrammed to do so. The same can be said for environmental and structural changes made to the robot hindering it from completing a task that it is more than able to do but cannot since it was never programmed to handle change. Reconfigurable robots provide a future where robots are capable of performing multiple tasks, adapting to changes in their structure and environment and altering their behaviour to overcome unanticipated situations. The more capable robots are at adapting to changes, the more versatile, robust and cost effective they become. This is the reason that so much time is invested into this area of research, to have robots that can not only perform the task they were designed to do, but can also adapt and overcome unanticipated events.

In addition to individual robots adapting to unanticipated events there is also an increasing need for multiple robots to communicate and coordinate amongst themselves to adapt to these changes in a joint manner. This can be achieved in two separate ways, firstly by having the robots make physical connections to each other and synchronising their individual movements to move the system as a whole. This is referred to as a self-reconfigurable, modular robot and is discussed more in the literature review in section 1.4.1. The second is achieved through robot swarms or platoons where the robots will work together as a group to perform a common task. This type of group adaption relies more heavily on the coordination of multiple robots, since they are not physically connected, but still need to move together in a joint manner. Both these types of systems, the modular and the group formation, rely heavily on communication and synchronisation systems to ensure they can adapt to changes as a group, rather than as individuals.

1.2 Project Aims

The primary aim of this project is to design the initial framework for creating a self-reconfigurable robot. This primary aim has been further broken down into the specific aims of this project which are to:

1. Revisit and enhance existing two-wheeled, balancing robot designs, and implement a control system capable of balancing and moving such a robot whilst maintaining it in the upright position.

2. Implement a scalable behavioural system into the robot that permits the robot to make its own decisions and adapt to changes.

3. Develop a scalable communication system to enable communication between the robot and external human-controlled Bluetooth devices as well as another robot.

4. Design a distance control system capable of synchronising two balancing robots at a set distance from one another whilst they move around autonomously.

5. Combine all the previous aims into one complete system allowing the balancing, communication, behavioural and synchronisation systems to run simultaneously on the robot.

1.2.1 Naming the Robot

Over the course of the year the robot has adopted its own name that allows it to be more identifiable and provides its own sense of uniqueness. The robot has been nicknamed GELway due to the fact that it was made from LEGO technology and its resemblance to Dean Kamen's personnel Segway transporters [2, 5]. The word LEGO was simple reversed with the 'O' dropped and combined with the second half of the word Segway. Throughout this thesis the two-wheeled, balancing robot will commonly be referred to as the GELway.

1.3 Thesis Overview

This thesis report has been separated into chapters, with each chapter relating to its own specific area in this project. A brief overview of each chapter will now be provided.

Chapter 1 - Introduction

This is the introductory chapter to the thesis, which provides the aims of this project as well as how this thesis document will be structured. The introduction will also cover today's need for research in reconfigurable robots and, in the literature review, will discuss previous work that has been carried out in this area. After the literature review, a critical analysis will be carried out on the research signifying the knowledge learnt and how it was used in this project.

Chapter 2 - GELway Overview

This chapter first presents the final design selected for the two-wheeled, balancing robot. Following this, in section 2.2.3, the motion equations of the robot are derived, based on existing documentation. This allows us to construct a simulated model of the robot on Simulink. Two existing balance controllers were analysed in this chapter with the PID controller determined as the most viable as it provides characteristics desirable for balancing robots such as its small overshoot. Difficulties faced during the implementation of the balance control system were that the existing design was programmed only to work with one gyroscope sensor and that the robot was incapable of rebalancing once it had fallen over. This was overcome by programming a self-calibrating method for the gyroscope to allow any LEGO gyroscope sensor to be used covered in section 2.3.2. Also by adding a self-reset mechanism which allows the robot to detect if it has fallen over and reset its balance parameters. This allows it to rebalance once placed back in the upright position. This is covered in section 2.3.6. Chapter 2 also covers the development of movement methods which allow the direction of the balancing robot to be controlled. This movement functionality, covered in section 2.5, allowed the robot to control its own movements and allowed external Bluetooth devices to drive it around once the Bluetooth connections were established.

Chapter 3 - Behaviour Programming

This chapter details the design of the behavioural system for the robot. The behavioural system allows the robot to make its own decisions based on sensor or Bluetooth information. To process all the behaviours, a subsumption architecture arbitration system was used and has been covered in section 3.2.1. The arbitration system decides which behaviours should become active, with higher level behaviours taking precedence and suppressing lower level behaviours. It was designed in such a way that robot behaviours could easily be added and removed, providing a scalable platform that will allow future functionality to be added to the robot. The implementation of the behavioural system also allowed us to test the capability of running the balance control, communication and synchronisation systems simultaneously. This was achieved by allowing the behavioural system to modify the balance parameters to cause movements in the robot, using sent Bluetooth commands as triggers in the behaviours and implementing the distance control system as a lower level behaviour. This raised some concurrent programming issues if both systems were trying to access the same data at the same time, but was overcome by synchronising all shared parameters. Section 3.3.1 covers the synchronisation issues in more detail.

Chapter 4 - Communication between Heterogeneous Robots using Bluetooth

This chapter covers the design and implementation of the communications system for the robot, which allowed it to send and receive commands from external Bluetooth devices and other robots. The communication feature is important as it allowed for testing of multiple robot communication and data sharing. One of the main issues faced when sending the robot commands was that it could miss sent information if it was not ready to receive. This was overcome by initiating a Bluetooth thread within the robot that would run in parallel with the balance control system and constantly wait for new commands. This is covered in section 4.2.1 of this thesis. Once the Bluetooth reader thread was established in the robot it was successfully connected to a mobile phone. This was achieved by reprogramming an existing mobile application to send commands to the robot, which could then be interpreted by the robot as actions. Section 4.2.2 details the mobile phone connection with section 4.2.3 covering a similar methodology to connect a laptop PC to the robot. Connection with

external Bluetooth devices meant that the movements of the robot could be controlled allowing an operator to drive the robot around. This provided a means of conducting a lot of tests involved in this project such as testing the robustness of the controller and stability over slopes. Section 4.3 covers the communication link created between two balancing robots in a master/slave configuration. With a two-way communication link established between the master and slave robot they were able to send sensory information to each other providing them with the ability to get a better overall picture of their environment since their amount of available sensor ports effectively doubled. By attaching different sensors to the robots they can take on their own specific role within a group of robots, for example a robot with an attached compass sensor could become the navigator of the group.

Chapter 5 - Synchronous Movement of Heterogeneous Robots

This chapter addresses the synchronisation of multiple robots and is the last topic covered for this project. The synchronisation of multiple robots is important for reconfigurable systems as it allows the complete system to control and manoeuvre its modules to allow it to form new shapes and a range of different movements. A model of the distance control system was first to built using Simulink to enable us to optimise its performance before programming it into the GELway. The distance control system was then programmed as a basic behaviour, which was added as lower priority into the behavioural system and kept the follower robot at a set distance from a given object. The design of this distance control system and its implementation into the robot can be found in sections 5.1 and 5.2 respectively. The successful implementation and testing of the distance control system proved that it could keep a robot at a set distance from a given target however could not be tested against complete robot synchronisation due to limitations in the distance sensors available in the laboratory.

Chapter 6 - Conclusion

This chapter will provide a summation of all the information that has been covered throughout this thesis concluding with the milestones that have been achieved as well as future work that could be conducted into reconfigurable robots.

1.4 Literature Review

The literature review provides a detailed commentary of the similar work to this project that has been previously completed. The intention of this literature review is to provide the reader with background knowledge on reconfigurable robots and previous research that has been made in this area. In addition to covering reconfigurable robots this review will also provide information relating to two-wheeled, balancing robots and the use of behavioural and communication systems in robotics. At the end of each section in the literature review a critical summary will be given detailing how the work relates to this project and the knowledge gained.

1.4.1 Reconfigurable and Adaptive Robots

Reconfigurable robots are basically robotic systems that alter their shape to adapt to changes in their environment or physical structure [6]. Adaptive robots can alter their behaviour to adapt to such changes. The latter change is generally taken as damage caused to the robot such as loss of limbs or sensors. This literature review will look into two types of reconfigurable robots: one that is made of numerous modules that can connect and disconnect themselves to form different structures, and one that alters its behaviour and movement to adapt to changes in its environment and physical structure. Both these robots cover key topic areas that relate to this project such as communication between multiple robots and using a behavioural system to adapt to change.

M-TRAN I/II/III Self-Reconfigurable Modular Robot

The M-TRAN is a self-reconfigurable robot which is separated into modules that allows it to connect and reconnect body parts to form different configurations. The robot is pictured in Fig 1.1 in its four legged walker state.

Figure 1.1: M-TRAN-II in its four-legged walker state [7]

Due to the robot's unique hybrid design, using both lattice and chain type connections, it is able to reconfigure its structure into numerous configurations to optimise itself in its operational environment [6]. This type of adaptability makes the robot perfect for missions where terrain is largely unknown, such as planetary exploration and search and rescue missions [7]. Fig 1.2 shows the numerous structural designs the M-TRAN can form.

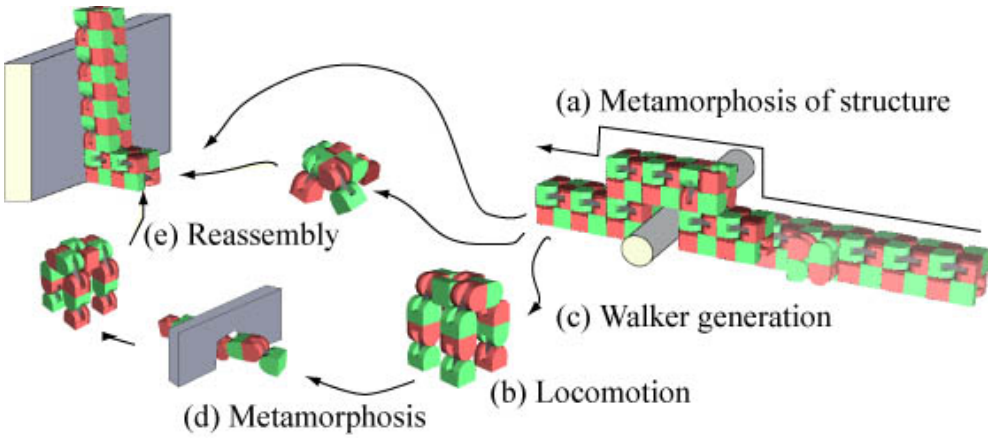


Figure 1.2: M-TRAN reconfiguring its structure to adapt to its environment [7]

This robot relies heavily on communication between all its modules to allow for movement to be performed and is why communication of multiple robots is a large portion of this project. One of the major difficulties with modular robots is getting the modules to move in unison to produce movement of the combined structure [8]. The makers of M-TRAN used a lot of simulated trials to produce locomotion and from there they were able to transform this into movements in the actual robot [6]. Modular robots show great promise in the future of reconfigurable robotic research, with the technology constantly evolving.

M-TRAN Critical Summary

The M-TRAN modular design shows a possible future to take the software platform designed in this project. It highlights key areas that need to be developed such as communication and synchronisation of multiple robots and the ability to adapt to change. These topic areas are the focus of this project and by developing these systems in a scalable way we have allowed for future work to take our system into the modular robotic world.

Cornell "Starfish" Robot

The Cornell robot an adaptive fault-tolerant robot with a unique ability to adapt to injury when it loses one of its limbs [9]. Its ability to adapt to change makes it a desirable research area for this project since it is also a need of our robot to adapt to changes. The Cornell Robot has the ability to teach itself how to walk by first moving each of its limbs to find their degrees of freedom. This process allows the robot to compensate for loss of a limb since it can reteach itself how to move with the damaged part of its body [9]. This is achieved by having tilt and angle sensors in the robot's joints to monitor its own movements [10]. Fig 1.3 shows a picture of the robot in a standing position. The robot begins its lifecycle not knowing what its parts are and how they are arranged. It then finds its range of movements and processes how to move its parts to meet its prime objective of moving forwards [9]. The robot forms a set of behaviours that allows it to move and adapt to changes in its environment by constantly optimising its movement patterns.

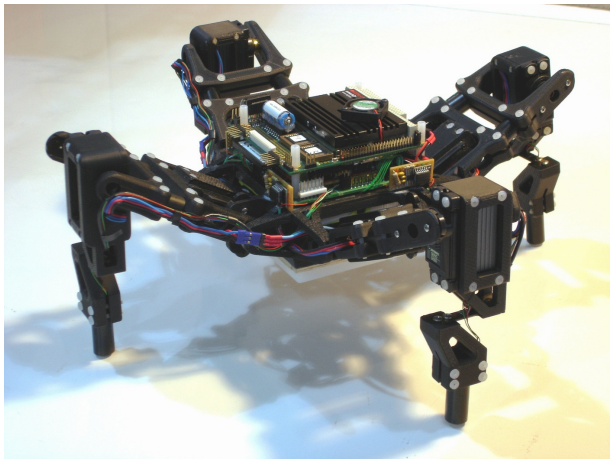


Figure 1.3: Cornell robot standing [11]

This type of technology, like the self-reconfigurable modular robot, allows it operate in uncharted terrain and gives robots the possibility to go to places where they could not previously operate.

Cornell Critical Summary

This fault-tolerant robot shows another possible future research area for this project. The Cornell robot has the ability to change its behaviour in order to adapt to environmental and structural changes. This area is of interest to this project since we require our balancing robot to also adapt to changes in its operational environment. This project has used a behavioural system to account for these changes but has a key advantage over the Cornell robot in the fact that it was designed to able to work with a group of robots and not as an individual.

1.4.2 two-wheeled, balancing robots

two-wheeled, balancing robots stem from the classic cart-pole experiment in which a designed control system attempts to balance a pole which is free to move with one axis of movement on top of a cart that moves in the same axis. This cart-pole design is also well known as the inverted pendulum problem [12]. This classic problem has evolved into robotics with two-wheeled, balancing robots. The added benefit of having a robot balancing is that wheels can be rotated at different speeds, allowing the robot to have two axes of movement. The intention of this section is to describe previous work that has been carried out on two-wheeled, balancing robots as well as the benefits they provide. These days two-wheeled, balancing systems are commonly referred to as Segway robots since the invention of the Segway Personnel Transporter (PT) by Dean Kamen in 1999. For this reason the Segway PT will first be introduced, detailing how it works. From there several Segway robots that have been made using LEGO Mindstorms technology will be covered.

Introduction of the Segway

On July 27, 1999, renowned inventor Dean Kamen established a company, Segway LCC, with a vision to develop a highly-efficient, zero-emission transportation solution that utilised dynamic stabilisation technology [13]. On December 3, 2001, Segway announced the first self-balancing transportation device known as the Segway Personal Transporter [13]. The Segway PT is pictured below in Fig 1.4.

Figure 1.4: Dean Kamen's Segway PT [5].

How the Segway Works

The Segway PT movements work with human equilibrium allowing a user to accelerate by leaning forwards and reverse by leaning backwards with speeds of up to 20 kph [14]. The Segway PT is controlled by five micro-machined gyroscope sensors and two accelerometers, allowing it to detect changes in terrain and body position at 100 times a second [5]. When the operator uses the handlebar to turn left or right, one wheel will rotate faster than the other, or the wheels will spin in opposite directions, allowing the vehicle to rotate around a single axis [15].

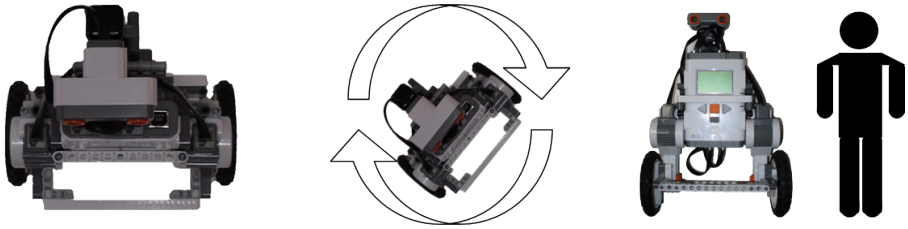
Only three of the five gyroscope sensors are needed to detect leaning forwards and backwards (known as pitching), leaning left and right (known as rolling), and left and right steering (known as yawing). The inclusion of extra sensors adds redundancy to the system, further improving the reliability of the product [5].

The Segway PT uses a special, solid-state angular rate sensor made of silicon. Using this type of gyroscope, an objects rotation can be determined using the Coriolis effect on a very small scale [15]. A Tiny silicon plate mounted on a support frame makes up a typical solid-state silicon gyroscope. An electrostatic current applied across a plate moves the silicon particles around in a particular way. This results in the plate vibrating in a predictable manner [15]. When the plate is rotated along its axis the particles will shift in relation to the plate, in turn altering the vibration proportionately to the degree of rotation. Information on the change in vibration is measured by the gyroscope system and sent to the computer [15]. The onboard computer system consists of 10 onboard microprocessors with two electronic controller circuit boards. The two electronic controller circuit boards act as a failsafe so that if one breaks down, the other can take over all the functions and shut down safely [15].

1.4.3 Benefits of a two-wheeled, balancing robot

There were several reasons why a two-wheeled, balancing robot was selected over conventional three and four wheeled robots in this project. Most of these advantages will be drawn from an article written by Albert Ko, H.Y.K. Lau and T.L. Lau on two-wheeled, balancing robots [4]. The advantages are listed below, along with a visualisation of the robot.

- **Small Footprint** - The physical dimensions of the GELway gives it a very small footprint allowing it to navigate between tight areas. A top down view of the GELway is shown in Fig 1.5(a).
- **Zero Turn Cycle** - A two-wheeled, balancing robot is capable of turning on the spot thus improving its capability of navigating through tight spaces. Fig 1.5(b) shows the GELway halfway through a rotation.
- **Dimensions** - The design of the robot gives it a physical structure much like that of a human, with its visual sensors placed high on the body, shown in Fig 1.5(c). The design could make it more socially acceptable if this robot's task involved interaction with humans, such as a security role.



(a) Small Footprint (b) Zero Turn Circle (c) Dimensions

Figure 1.5: Advantages of a Balancing Robot.

- **Centre of Gravity** - One of the main advantages a two-wheeled robot has over three and four wheeled robots of the same design is that the centre of gravity is shifted above its wheels at all times. This allows the robot to still handle disturbances over slopes, and also ensures that sensors are positioned in the same direction over any incline. Fig 1.6 shows a visual comparison between two, three and four wheeled robots.

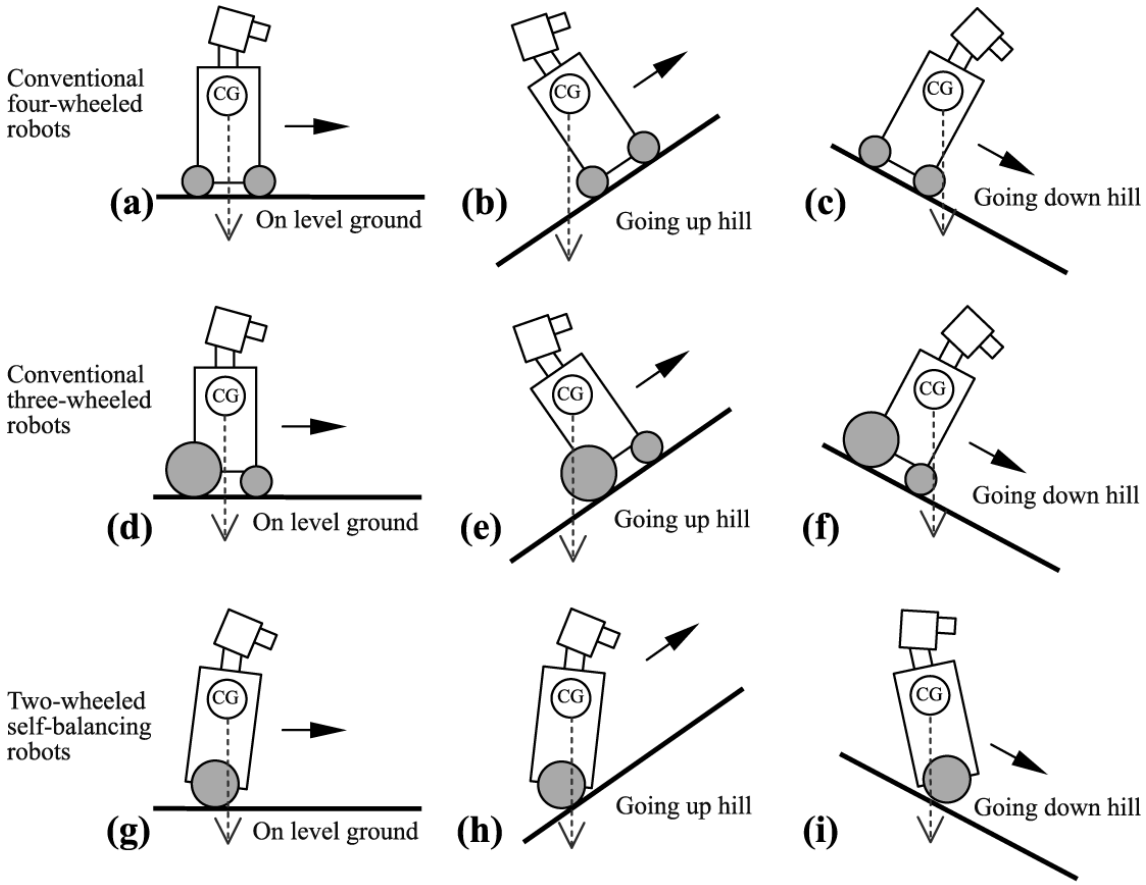


Figure 1.6: Two, three and four-wheeled robots centre of gravity over slopes [4].

1.4.4 History of the LEGO Segway Robot

LegWay (Segway Robot) - RCX and Electro Optical Proximity Detectors

Steve Hassenplug created one of the first successful self balancing robots using a Lego Mindstorm's RCX brick and two Electro Optical Proximity Detector (EOPD) [16] sensors. This robot, nicknamed the LegWay, balanced on two wheels, and had the capability to follow a black line. A picture of his design is shown below in Fig 1.7.

The LegWay remains upright by utilising the EOPDs to maintain a constant distance from the ground. The LegWay is programmed to move forward if the distance decreases and move backwards if the distance increases [16].

The LegWay can function off one EOPD to remain upright with both EOPDs used to follow a black line. Following the black line is done by reversing the motors causing a tilt in the robot, which is automatically corrected by moving forward. Once one sensor passes the line, it stops its respective motor, causing the LegWay to balance on one wheel, thus turning the robot [16].

The EOPDs used in Steve Hassenplug design were manufactured by a third party company known as HiTechnic. They work by using an internal light source to detect the presence of an object and calculates the distance; in this case the ground is the object [17].

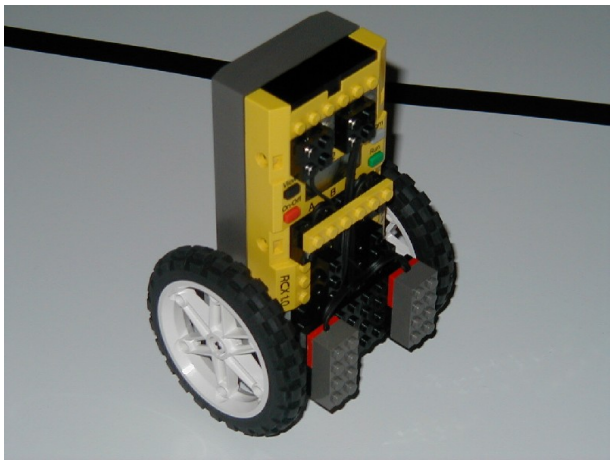


Figure 1.7: Steve Hassenplug's LegWay [16].

NXTway (Segway Robot) - NXT and Light Sensor

Philippe Hurbain, drawing inspiration from Steve Hassenplug's LegWay, built a similar balancing robot using the a LEGO Mindstorms NXT brick and light sensor. The balancing robot, nicknamed the NXTway, was capable of balancing for a short period of time, however had a much more rudimentary control system compared to Hassenplug [18]. Pictured below in Fig 1.8, the NXTway is shown with the LEGO light sensor centred between the two wheels.

A major difference between the LegWay and the NXTway is the type of sensor used to calculate the distance. Hassenplug stated that building a balancing robot with the standard LEGO light sensor would be difficult as it does not have the required resolution [16].

Due to Hurbain's NXTway not being as well documented as Hassenplug's LegWay it is difficult to compare the two. However Hurbain does make reference to his NXTway being affected by environmental conditions, such as surrounding lighting interfering with the sensor readings [18]. This is the reason why Hassenplug favoured the EOPD sensor, as it emits its own light source making it able to filter out external light signals [17].

Figure 1.8: Philippe Hurbain's NXTway [18].

NXTway–G - NXT and Gyroscope

The NXTway-G, created by Ryo Watanabe, is a descendant of both Steve Hassenplug's LegWay and Phillipe Hurbain's NXTway. Both the previous LEGO Segway robots, the LegWay and the NXTway, used a light or EOPD sensor to measure the robots body's rotational angle. Due to information on the wheel's rotational angle not being used the robots do not achieve internal stability. Watanabe's NXTway-G, equipped with a gyro sensor and two rotary encoders, calculates the rotational angle of the wheel and the angular velocity of the body [19]. The NXTway-G has a remote control interface attached at the top, like a head, allowing the robot's movements to be controlled. It can be seen in Fig 1.9

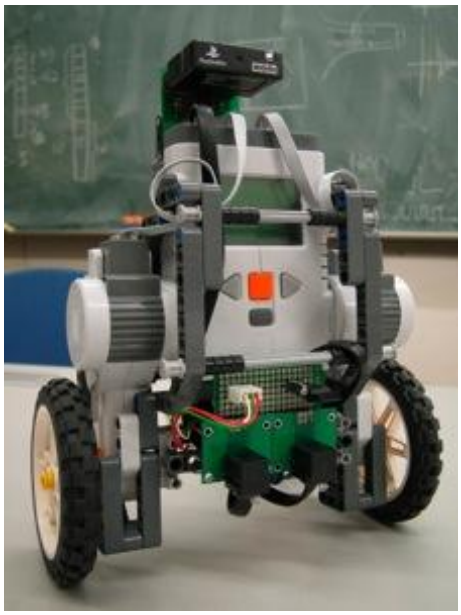


Figure 1.9: Ryo Watanabe's NXTway-G [19].

LEGO Segway Robot Critical Summary

The three LEGO Segway robots researched for this project each provided important lessons learnt for the robot developed in this project. Firstly we learnt from the LegWay that the EOPD sensor is much more capable than the LEGO light sensor when detecting the ground surface. This is the reason why we chose the EOPD sensor to be used in the self-reset mechanism in the GELway over the LEGO light sensor. The NXTway backed up this

previous statement with the developer commenting on the shortcomings of using the LEGO light sensor, such as its susceptibility to external light sources. The last robot researched was the NXTway-G which used a gyro sensor to measure the rotational angle rather than a reflected light source. The gyro sensor was found to be a more capable sensor to keep the robot balanced and it provided an improved method of measuring the robot's rotational angle. The developer of the NXTway-G, however, used a remote control interface to control the movement. This was not deemed necessary in this project since we could use the NXT's Bluetooth capabilities. What these robots lack, however, is the fact that they were only ever designed to be balancing robots. This project will extend from these designs by adding capabilities such as a behavioural system to allow the robot to adapt to changes and provide the communication and synchronisation of multiple robots.

1.4.5 Behavioural Programming

Behavioural programming is typically separated into two key areas, the behaviours that provide commands to the actuators, and the arbitration system used to determine which behaviours should be activated. This section of the literature review will first look into the background theory of the behavioural system. After this information has been presented we will see how a behavioural system implemented into a soccer playing robot has helped improve its performance in its objective processing. Behaviour-based programming in robots enables them to attend simultaneously to hazards they may face as well as unexpected opportunities they may come across [20]. It gives the robot the capability of making decisions on their own based on their environmental surroundings and sensor inputs. By implementing a behaviour based system into robots we can program the robot to act differently under different circumstances rather than do the same thing all the time. Behaviours, referred to as primitive behaviours in Jones' programming book, allow robots to perform numerous functions with only a limited amount of sensors [20]. The behaviours will now be covered in more detail.

Behaviours

Primitive behaviours have two parts, a control component that takes in sensory inputs and converts them into actuator commands, and a triggering mechanism that determines when the control component should act [20]. The control system and the trigger can use the same or different sensors. Fig 1.10 shows the diagram of a primitive behaviour.

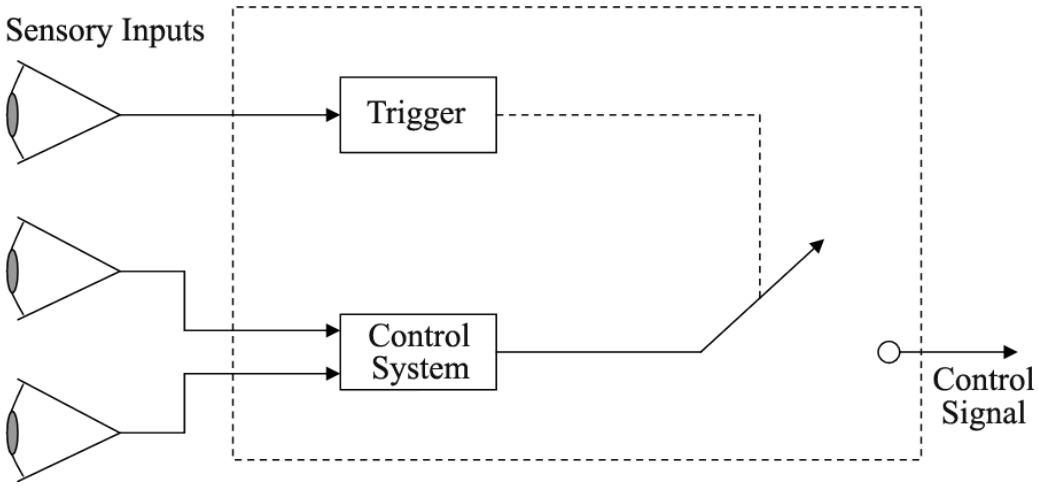


Figure 1.10: A primitive behaviour with sensor inputs, a trigger and a control signal output [20].

It does not matter whether the trigger mechanism prevents the primitive behaviour from computing the control system or only prevents the control system from sending commands to the actuators, as the end result to an outside system is the same [20]. Note that problems could arise if the control system requires extensive processing. Primitive behaviours are broadly described in two categories: servo and ballistic.

Servo behaviours typically utilise a feedback control loop as their control component [20]. In terms of a balancing robot, the mechanism used to keep the robot upright can be seen as a servo behaviour. The behaviour continuously reads sensor information in order to sustain the robot in the upright position.

A ballistic behaviour is triggered once and follows a predictable process until finished [20]. An example would be a behaviour to keep the robot from running into an obstacle. If the robot moves too close to an object, the behaviour triggers and moves the robot into a safe

position or keeps the robot stationary, suppressing any active movement behaviours. This type of behaviour will be described in Chapter 3 in detail.

Arbitration

In order to allow a robot to perform multiple goals simultaneously, numerous primitive behaviours are generally required. The system will work smoothly provided only one behaviour triggers at any given time. Thus in order to ensure behaviours are not all triggered at once, trying to control the same actuators, arbitration mechanisms are put in place which allow one behaviour to activate at any given time [20].

Subsumption architecture provides one means of arbitrating behaviours in a robot, and has been selected for this project as it allows prioritising in the robot behaviours. It works by having a low priority behaviour which is constantly triggered and sends commands to the actuator. The arbitrator then cycles through all higher level behaviours checking if any of their triggers have been activated. When a higher level behaviour wants to take control the lower level behaviours are suppressed. This system ensures that only one behaviour is sending commands to the actuator at any given time and gives higher level behaviours priority over lower level ones [20, 21].

Whilst it is possible to program a robot to perform various tasks without using a behavioural system, it can become quite difficult to implement new behaviours into the robot. The advantages of using a behavioural system lie with the fact that new behaviours can be easily added to the system giving the robot new functionality. Also behaviours can be shut off by removing them from the arbitration mechanism.

Behavioural Robots - Four Legged Robocup

The Robocup is a tournament in which programmers develop software for soccer playing robots and compete against other programmers [22]. This section of the literature review will look at one of the teams who developed a behavioural system to help improve their goalie for the soccer match. Mantz and Jonker set to improve the performance of the vision system of autonomous, perception-based robots by implementing a behaviour based software architecture made up of numerous independent trigger-action loops [23]. The robot they were working with was Sony's four-legged, mechanical pup pictured below in Fig 1.11.

Figure 1.11: Sony's four-legged mechanical pup playing soccer in Robocup [24]

The goalie robot has three main actions that it must perform [23]. They are:

1. Return to goal – When the goalie is not in the goal area he must return to goal.
2. Goalie position – When no ball is near the goalie should remain central to the goals.
3. Goalie clear ball – If the ball enters the goal area, the goalie will clear the ball.

These are visualised below in Fig 1.12 in order of the aforementioned actions.

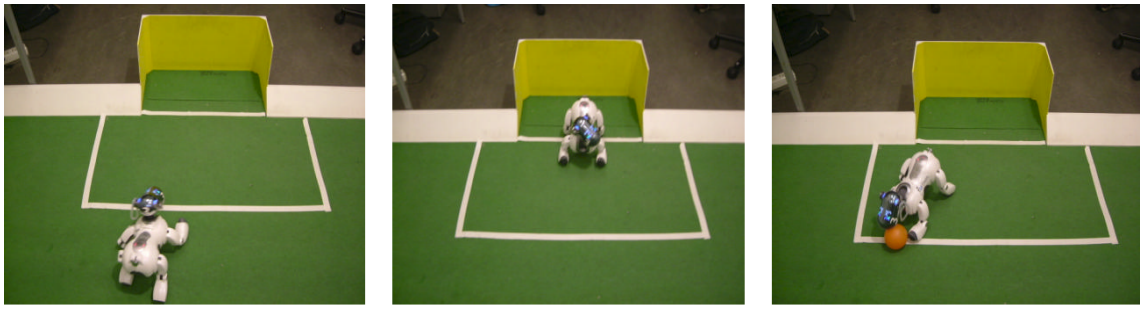


Figure 1.12: The basic goalie behaviours: (a) Return to goal (b) Goalie position (c) Goalie clear ball [23]

Mantz and Jonker designed a behavioural based system for the four-legged robot that could implement the behaviours when triggered by certain events, such as using its camera to detect if the ball has entered the goal area. The behavioural control system has been summarised below in Fig 1.13.

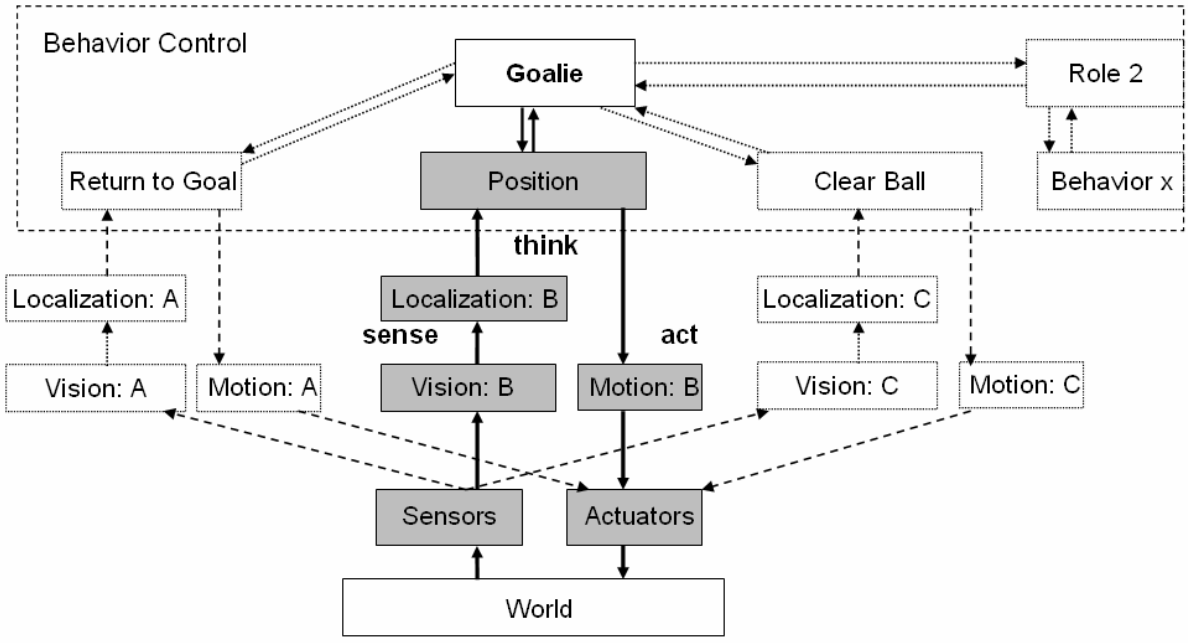


Figure 1.13: Behavioural based control system for soccer robot [23]

Four Legged Robocup Critical Summary

This behavioural system implemented in the four legged robot used the process of having higher level behaviours to suppress lower level behaviours, much like the system implemented in the GELway, which is described in detail in Chapter 3. The behavioural system allowed the robot to process multiple data inputs simultaneously from its sensors. By implementing the behavioural system the programmers saw more than a 50% improvement in performance of localisation and goal-clearing tasks [23]. Whilst the behaviours we utilised in the GELway are different to the Robocup robot we can see that in this case the behavioural system has improved the performance of robots attempting to perform multiple tasks at once.

1.4.6 Communications Systems in Robotics

This project reviewed two types of communication systems to control our robot, centralised and decentralised control. In centralised control multiple robots will work together performing actions based on a group decision. In decentralised control the robots will make decisions based on their own individual input rather than the groups [25]. This section also addresses the difference between decision making in fully autonomous and human-in-the-loop robots. The intention of this section of the literature review is to critically analyse these communicative control types in order to choose which method best suited our project.

Centralised Control

In a centralised control communication system data from multiple systems is sent to one processing unit that can then make decisions based on the group's information [25]. In this project we have used the terminology master/slave to describe this process, where the master is the processing unit that makes the decisions and the slaves send their data to the master and are delegated tasks. The advantage of this type of communication is that decisions are able to be made from a large amount of data, based on the group's surroundings. This means that the master is able to make decisions based on the group's interests and not that of the individual robots themselves. The drawback is that processing the group's incoming

data can be time consuming and decisions cannot always be made quickly. This can become more apparent when there are numerous slaves involved in the system.

Decentralised Control

In a decentralised control communication system all individual robots make their own decisions based on local data [25]. This allows decisions to be made by the individual robots in a much faster time when compared to centralised control as they are do not have to wait for the master unit to make their decisions. Also the amount of data that needs to be processed is much smaller than in centralised systems as only data from the individual unit needs to be taking into account. Decentralised control however provides no means of communicating with other individual robots and as such problems solved by individual robots could take much longer than a group of robots working together [25].

Fully Autonomous Robots

Fully autonomous robots are robots capable of making their own decisions based off their environmental input without the need for operator control. They can either be in centralised or decentralised control systems but are only classified as fully autonomous provided they need no input from a human [25]. The advantage of a fully autonomous robot is that they require no human input and can perform their tasks without the need for constant supervision. This allows autonomous robots to replace humans in tasks that may be considered tedious, such as processing large amounts of information. However there can be social implications with fully autonomous robots, especially if they are making decisions that can directly affect a human's well-being. To a robot, humans are simply another data input making them incapable of valuing human life [26]. As such there is generally a fine line between what tasks we allow fully autonomous robots to do, leaving decisions based on human life to humans, and not robots [26].

Human-in-the-loop Robots

Human-in-the-loop robots, as the name suggest, are robots that have a human operator in the decision making process. This means that these types of robots cannot be classified as fully autonomous, however overcome the social implications fully autonomous robots can have by allowing a human to operate functions within the robot. It loses the advantage of fully autonomous robots since human supervision is required whilst the robot is operating.

Communications System Critical Summary

After reviewing these communications types it was decided the best communication system for our robot would be a centralised, human-in-the-loop, autonomous robot. The decision to choose a centralised system was due to the primary aim of the project of developing self-reconfigurable robots. By using a centralised system we can control a group of robots rather than the individual robots themselves. Also the choice of a centralised system means that future work is able to be carried out in developing modular reconfigurable robots or robot platoons, which requires robot group communication to move the system as a whole. The human-in-the-loop control system was largely selected since the robot is in the development stage. By having a human control in the robot we were able to test all of its functionality in a controlled environment. It should be noted that the behavioural system designed in this project allows for either fully autonomous or human-in-the-loop controlled robots providing a variety of directions to further develop this project in the future.

1.4.7 Literature Review Critical Summary

Whilst it may not be possible to create a modular robot using the NXT system, there is still an opportunity to utilise the research found on the two-wheeled self-balancing robot and build a new system, capable of connecting two Segway robots into a four or more wheeled modular systems. By developing a two-wheeled, balancing robot into a modular system we can create a variety of different configurations that are capable of achieving different tasks. Some advantages of combining two-wheeled, balancing robots are:

- If the robot is required to traverse unstable terrain, such as stairs or rocky ground, a four-wheeled system could provide more stability.
- With four motors working to drive one system, there would be a speed and strength increase, giving the modular system more power to possibly push or pull heavier objects.
- By combining two Segway robots any similar sensors can be shared between the two systems, allowing redundant sensors to be switched off, thus saving battery power. With the four-wheeled arrangement the balancing mechanism used in the two-wheeled system can be switched off, since stability would significantly increase. Only one robot would be required to operate the obstacle detection sensor too, in turn reducing power consumption. Connecting the two robots would also allow for power to be shared within the system.
- The combination of two operating systems would provide dual processing power, increasing the speed at which data can be processed.

In addition to these advantages there are also advantages of having two separate Segway systems. Examples include:

- A search and rescue mission - which would be conducted faster as separate systems. This would be vital in time critical operations where lives may be at stake.

- Occurrences where a four-wheeled system would be too bulky to pass through certain obstacles. Separating into two separate systems may provide more agility and a smaller structure, allowing the modular system to pass through obstacles.
- A requirement for multiple tasks to be completed. By separating the system, and completing the tasks concurrently, a decrease in the overall task time could be achieved.
- If the modular system is damaged, there is opportunity to remove the damaged module, and separate into one operating Segway.

The intention of the system built for this project was to ensure that it was scalable to allow numerous avenues to develop the software. Modular robots are one of these possibilities and allow the system to excel in applications where versatility is crucial.

1.5 Programming Language of the GELway

The leJOS NXJ programming environment was chosen to program the LEGO Mindstorms NXT [21]. There are several programming languages that are capable of being implemented onto the LEGO Mindstorms NXT system. They range from introductory programming languages like the standard firmware that comes with the NXT, to programming languages suited to more advanced programmers such as C, C++ and Java. A complete list of all the programming languages for the NXT can be found at Steven Hassenplug's personal website [27]. The programming language chosen for this project was leJOS NXJ which allows a user to program LEGO robots in Java [21]. The leJOS NXJ firmware will be briefly introduced, covering its advantages and disadvantages, to explain why it was chosen for the programming language for the GELway.

1.5.1 What is leJOS NXJ?

The leJOS NXJ firmware is a tiny Java Virtual Machine (JVM) that was introduced to the LEGO NXT brick in 2006. The leJOS package comes with a library of Java classes and useful PC tools that allow flashing the firmware, uploading and debugging programs and data logging [21]. The leJOS Java programming firmware was mainly chosen due to my programming knowledge being largely Java based, thus reducing the learning curve required to begin programming the robot.

1.5.2 Advantages of leJOS NXJ

There are numerous advantages to using a Java programming language on the LEGO systems, some personal and others related to the capabilities of the firmware. Several will be listed below however there are many more provided on the leJOS website [21]. These include:

- Object-orientated programming – this allows us to separate the different systems in the robot to separate class files showing the separation of these systems more clearly.
- Open source project with a large list of contributors – this means there are a lot of people working to constantly improve leJOS and add more functionality.
- Easy implementation into professional Integrated Development Environment (IDE) such as Eclipse and Netbeans.
- Full support of Bluetooth – This feature is vital to allow us to design the communication system to enable synchronization between robots.
- Behaviour classes implementing a subsumption architecture for simplifying programming of complex robot behaviours.
- Supporting the use of third party sensors – allows us to use the EOPD and gyro sensor.
- Supporting the use of multiple threads – allows us to test the capability of running multiple systems in parallel on the robot. This is one of the milestones for the project.

There are also some personnel advantages for choosing the leJOS NXJ firmware. They are:

- My previous experience in working with Java.
- My supervisor, Kathryn Merrick, has used the leJOS firmware before and will be able to provide assistance in difficult areas of the project.

1.5.3 Disadvantages of leJOS NXJ

Whilst there are numerous advantages to using leJOS NXJ there are a few disadvantages such as:

- Compared to other firmware types such as NQC and LegOS the memory usage and performance is generally lacking.
- There is a more solid network for support on the construction of balancing robots using C and C++.
- The Bluetooth connection speeds are generally slower when using Java compared to firmware based on C.

These disadvantages, however, where not seen as major and were far outweighed by the advantages the Java firmware provided. Thus the NXT was loaded with the leJOS NXJ firmware and all programming conducted on the GELway was done using the Java language and leJOS Java classes.

Chapter 2

Developing a Balancing Robot

The aim of this chapter is to detail the process taken to provide a stabilising control system to a two-wheeled, balancing robot that is capable of keeping the robot in the upright position, as well as provide other functionality such as movement and disturbance handling.

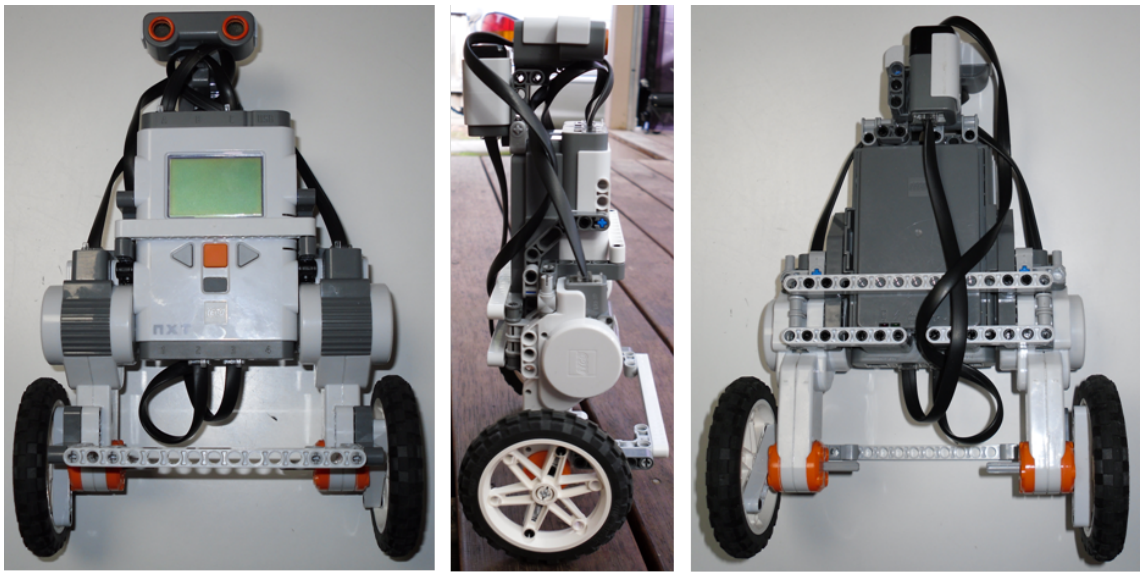
The first part of this chapter, section 2.1, introduces the final physical design selected for the GELway. The selection process will be discussed and we will provide the detailed building instructions for constructing the robot. From there the process taken to derive the equations of motion of this physical design, following well documented methods, will be covered. This allows us to get a good understanding of the properties in the controller plant. This is covered in section 2.2. This section will provide the detailed analysis of two separate balancing control systems, using Simulink to analyse the characteristics of each controller. The analysis showed that the PID controller provided the most ideal performance characteristics for our robot and as such was the controller chosen to be implemented into our balancing robot. This implementation of the PID controller is covered in section 2.2. The control system will build from previous work published by the team who built Marvin - The Balancing Robot, and will focus on improving the functionality and capabilities of the robot [28]. These improvements include a self-calibrating method for the gyro sensor and a self-reset mechanism that allows the robot to detect when it has fallen over.

After the control system implementation is presented, section 2.5 will explain the control mechanism that handles the GELway's movements. This section provides a detailed explanation on how a two-wheeled, balancing robot is able to move forwards and backwards and rotate around. Lastly section 2.6 will provide a detailed overview of the GELway's programming structure, showing all relevant class files and how they are interconnected. This overview will show how the other chapters integrate into the complete system.

2.1 Physical Structure of the GELway

The construction of the GELway follows the NXTway-GS building instructions provided by Yorihisa Yamamoto [3]. Alterations have been made to the position of the gyro sensor so that it sits higher on the body. This is because the original programming code is based off Marvin the Balancing Robot, and pictures of this robot have the gyro sensor positioned higher [28]. The creators of Marvin did not provide building instructions for their robot which is why we followed the NXTway-GS build instructions, which appeared similar in the pictures provided by both designers.

The final design selected for the GELway is shown in Fig 2.1, which gives a front, back and side view of the robot. Note that these designs may differ slightly depending on additional sensors that may be attached.



(a) Front View (b) Side View (c) Back View

Figure 2.1: GELway physical structure

The building instructions for the GELway can be found in Appendix A.12, which shows instructions on how to build the leader robot, pictured in Fig 2.1, as well as the follower robot, which will be used when the robots are moving in formation.

2.2 Control System

The physical structure of the GELway makes it incapable of remaining in the upright position without some form of controller. The centre of gravity is situated low on the body and is centralised between the two wheels, hence the slightest tilt in the robot will cause it to become unstable and fall to the ground. Due to its low centre of gravity, the GELway cannot get itself back into the upright position without external assistance from either a third motor or a human handler. Thus we desire a control system in the robot that can detect the degree and rate of tilt in the robot and correct its position by applying power to the motors.

The construction of a two-wheeled, balancing robot has been the desire of many programmers and control theorists for many years. As such, several attempts have been made to achieve a successful balancing robot, some of which have used the LEGO Mindstorms technology. These were previously presented in the literature review. However, the biggest drawback is that many of these attempts either use a different programming language, do not provide their source code, or are poorly documented, making them difficult to recreate. After numerous research attempts no one robot was found that could meet all the requirements we needed to recreate our own control system for the GELway, thus a compromise was made. Two robots were selected for analysis, each providing their own piece of information which would help complete our control system.

The first was the NXTway-GS, created by Yorihisa Yamamoto [3], which comes with a well documented paper detailing the control system used and most importantly derivations for the equations of motion for a two-wheeled, balancing robot. The software code however has been programmed in 'C', which means it cannot be loaded and tested on the GELway, which uses leJOS. The second selected robot was Marvin – The Balancing Robot, designed by Johnny Rieper, Bent Bisballe Nyeng and Kasper Sohn, who managed to balance a two-wheeled robot using Java firmware. However did not provide detailed information on the control system [28]. Their selection of control parameters was achieved through trial and error, or selecting numbers without providing details on how they were chosen. Thus one of the first tasks for this project was to first recreate and characterise the NXTway-GS control system

and compare the control system on Marvin. This was done using the Simulink simulation software. By doing this we can ensure we implement the most robust control system onto the GELway. Completing this task will also enable us to optimize the control system to suit the needs of our robot, should this be necessary. Both these robots were created using LEGO Mindstorms technology with both control systems using the same inputs and outputs, thus making them perfect candidates for a comparative analysis.

2.2.1 Designing the Controller

The simulated control model is made of four distinct parts:

1. The inputs and output of the controller.
2. The plant, which governs the equations of motion for the GELway.
3. The weighted gains applied to each feedback loop.
4. The feedback controller type used to stabilise the robot.

Fig 2.2 shows how each of these parts are related.

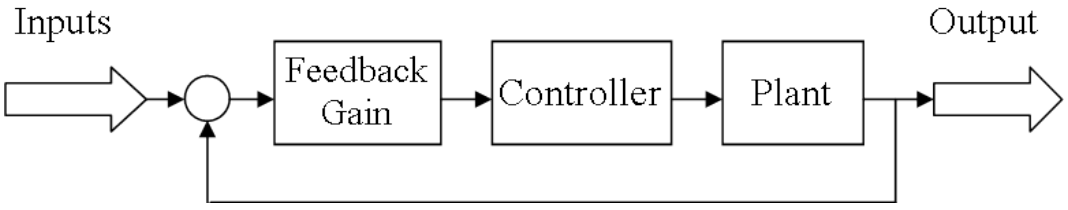


Figure 2.2: Control system for a two-wheeled, balancing robot

Each of these parts will now be presented individually.

2.2.2 Control System Inputs and Output

The inputs to the control system are the motor voltages (PWM) for each of the wheels. The outputs of the controller are the average of the wheel angles and velocities from the left and right motors, $\theta_{l,r}$, and the bodies angular velocity, $\dot{\psi}$. The body pitch angle used in the control system cannot be measured directly, and is instead found by integrating the measured angular velocity. These inputs and outputs are shown in Fig 2.3 below.

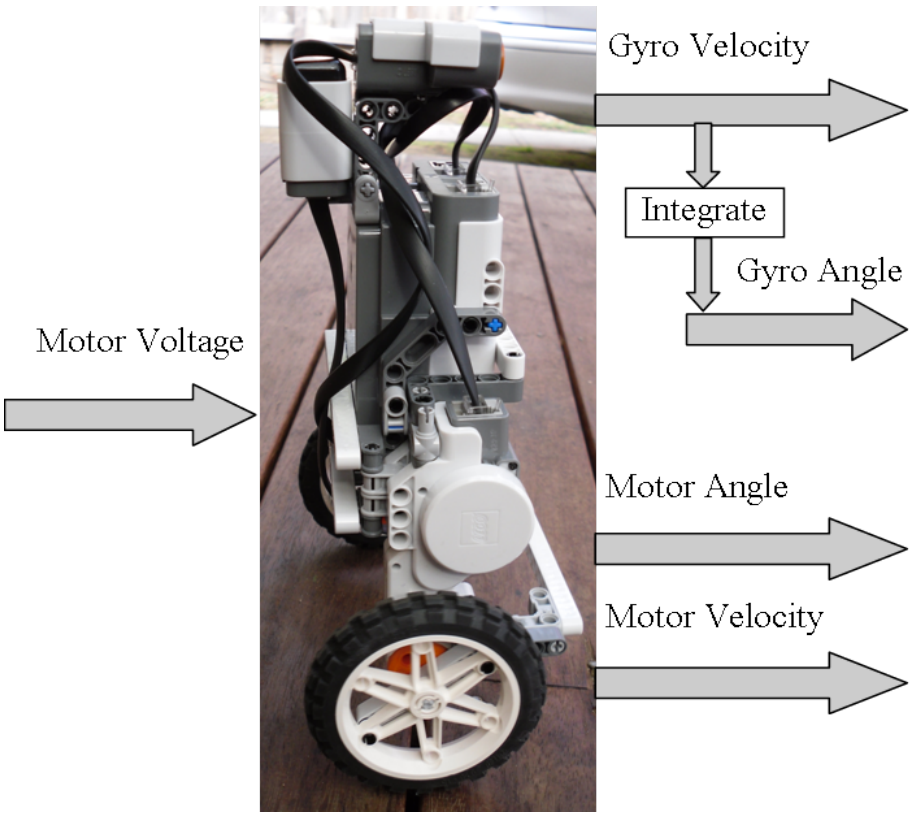


Figure 2.3: Inputs and outputs of the balance controller

2.2.3 The Plant Model

Fortunately, as mentioned before, the NXTway-GS documentation provides a detailed analysis for the construction of the plant. The first step in understanding the motion equations was to work through the coordinate system of the two-wheeled, balancing robot. This helped by allowing us to relate the equations to our balancing robot and highlight where our gyro and motor sensor parameters were being measured. Before the state-space equations and coordinate system are presented a table detailing of all the relevant parameters will be given and their respective units. This has been provided in Table 2.1.

Parameter	**Unit**	**Description**
θ	[deg]	Average angle of wheels
ψ	[deg]	Body pitch angle
ϕ	[deg]	Body yaw angle
θ_m	[deg]	DC Motor angle
g	$[m/sec^2]$	Gravity acceleration
m	[kg]	Wheel weight
R	[m]	Wheel radius
J_w	$[kgm^2]$	Wheel inertial moment
M	[kg]	Body weight
W	[m]	Body width
D	[m]	Body deptth
H	[m]	Body height
L	[m]	Body length
J	$[kgm^2]$	Inertia moment
J_ψ	$[kgm^2]$	Body pitch inertial moment
J_ϕ	$[kgm^2]$	Body yaw inertial moment
J_m	$[kgm^2]$	DC motor inertial moment
n		Gear ratio
K_t	[Nm/A]	DC motor torque constant
K_b	[V sec/rad]	DC motor back EMF constant
f_m		Friction coefficient between body and DC motor
f_w		Friction coefficient between wheel and floor.

Table 2.1: State-equation parameters for the GELway

The coordinate system used to derive the equations of motion for the GELway is shown in Fig 2.4.

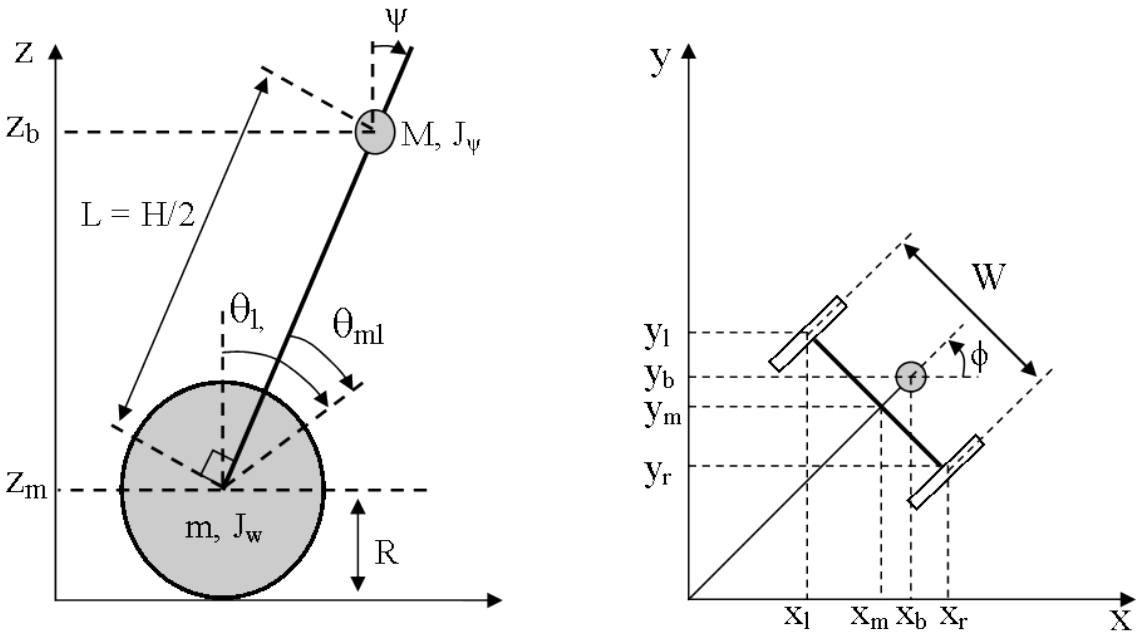


Figure 2.4: Coordinate system for equations of motion for the GELway [3].

The parameters that we are most concerned with are average angle of the wheels, θ, and the body pitch angle, ψ, as these are used to calculate the error signal for the balancing controller. From here the equations of motion were verified to prove they were correct and to ensure there had not been any sign errors, since one sign error in our system could cause it to become unstable. The linearised equations of motion for a two-wheeled, balancing robot are as follows, noting that complete derivations of these equations can be found in Appendix A.1:

$$\left((2m+M)R^2 + 2J_w + 2n^2J_m\right)\ddot{\theta} + \left(MLR - 2n^2J_m\right)\ddot{\psi} = F_\theta \tag{2.1}$$

$$\left(MLR - 2n^2J_m\right)\ddot{\theta} + \left(ML^2 + J_\psi + 2n^2J_m\right)\ddot{\psi} - MgL\psi = F_\psi \tag{2.2}$$

$$\left(\frac{1}{2}mW^2 + J_\phi + \frac{W^2}{2R^2}\left(J_w + n^2J_m\right)\right)\ddot{\phi} = F_\phi \tag{2.3}$$

Eq. (2.1) and (2.2) can be grouped together as they both consist of angles θ and ψ. Eq. (2.3) has only the yaw angle ϕ. The first two equations can now be expressed as:

$$S\begin{bmatrix}\ddot{\theta}\\ \ddot{\psi}\end{bmatrix} + T\begin{bmatrix}\dot{\theta}\\ \dot{\psi}\end{bmatrix} + U\begin{bmatrix}\theta\\ \psi\end{bmatrix} = V\begin{bmatrix}v_l\\ v_r\end{bmatrix} \tag{2.4}$$

where

$$
\begin{aligned}
S &= \begin{bmatrix} (2m+M)R^2 + 2J_w + 2n^2J_m & MLR - 2n^2J_m \\ MLR - 2n^2J_m & ML^2 + J_\psi + 2n^2J_m \end{bmatrix} \\
T &= 2\begin{bmatrix} \beta + f_w & -\beta \\ -\beta & \beta \end{bmatrix} \\
U &= 2\begin{bmatrix} 0 & 0 \\ 0 & -MgL \end{bmatrix} \\
V &= 2\begin{bmatrix} \alpha & \alpha \\ -\alpha & -\alpha \end{bmatrix} \\
\alpha &= \frac{nK_t}{R_m}, \quad \beta = \frac{nK_tK_b}{R_m} + f_m
\end{aligned}
$$

Furthermore, Eq. (2.3)can be written in the form:

$$
I\ddot{\phi} + J\dot{\phi} = K(v_r - v_l) \tag{2.5}
$$

where

$$
\begin{aligned}
I &= \frac{1}{2}mW^2 + J_\phi + \frac{W^2}{2R^2}\left(J_w + n^2J_m\right) \\
J &= \frac{W^2}{2R^2}(\beta + f_w) \\
K &= \frac{W}{2R}\alpha
\end{aligned}
$$

The next step is to write the state-space equations. Using the following variables x_1, x_2 as the state variables for Eq. (2.4) and (2.5) respectively, and u is the input variable:

$$
x_1 = \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \quad x_2 = \begin{bmatrix} \phi \\ \dot{\phi} \end{bmatrix} \quad u = \begin{bmatrix} v_l \\ v_r \end{bmatrix} \tag{2.6}
$$

we can rewrite Eq. (2.4) and (2.5) in the state space form as follows:

$$\dot{x_1} = A_1 x_1 + B_1 u \tag{2.7}$$

$$\dot{x_2} = A_2 x_2 + B_2 u \tag{2.8}$$

The state-space matrices, shown in their complete form in Appendix A.1, were then formulated using the using the *GELwayController.m* Matlab file with the source code to generate these in Appendix A.13. These are modified files provided by the NXTway-GS documentation.

The A_1 matrix, which represents the motor and gyro angle and velocity, was then checked for stability. This was done by finding the eigenvalues for the A_1 and ensuring none where positive, as a positive eigenvalue represents instability in the matrix [29]. The eigenvalues were computed for the A_1 matrix and are shown below in Table 2.2:

0.0000
-241.1937
7.4367
-6.5199

Table 2.2: A_1 Eigenvalues

The results show that the third eigenvalue was found to be positive and, as such, proved the instability of the A_1 matrix. Therefore, to obtain a stabilising controller the Linear Quadratic Regular (LQR) method was used [30].

2.2.4 Linear Quadratic Regular Method

The LQR was used to optimise the state-space matrices in the controller for the motor and body pitch angle parameters. The LQR method can be used to augment the system to reduce steady-state errors by providing feedback weighting for the input parameters which stabilise the system [30]. Consider the double-input, double-output system for the GELway, noting that there is now an additional state due to an integrator being included in the feedback loop:

$$\dot{x_1} = A_1 x_1 + B_1 u \tag{2.9}$$

where

$$A_1 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & A_1(3,2) & A_1(3,3) & A_1(3,4) & 0 \\ 0 & A_1(4,2) & A_1(4,3) & A_1(4,4) & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad B_1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ B_1(3) & B_1(3) \\ B_1(4) & B_1(4) \\ 0 & 0 \end{bmatrix}$$

and feedback

$$u = -Kx = - \begin{bmatrix} k_1 & k_2 & k_3 & k_4 & k_5 \\ k_1 & k_2 & k_3 & k_4 & k_5 \end{bmatrix}$$

The reason why there are two identical feedback control signals is that the control action is divided between two actuators, and due to the symmetry, the same gains are used for the left and right controller. The full equations for the A_1 and B_1 matrices can be found in Appendix A.1.

The LQR method uses the following equation to determine the optimal solution for the system, noting that the Q matrix determines the weighting factors for the feedback and R is the scalar weighting factor [30]:

$$A_1^T P + P A_1 - P B_1 R^{-1} B^T P + Q = 0 \tag{2.10}$$

Solving this equation for P will determine the optimised feedback parameters for our control system. Using the NXTway-GS documentations, the following Q and R matrices were used:

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 6*10^5 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 4*10^2 \end{bmatrix}, \quad R = \begin{bmatrix} 1*10^3 & 0 \\ 0 & 1*10^3 \end{bmatrix} \tag{2.11}$$

The *LQR* function in Matlab was then used to determine specified weights for each of the controller feedbacks. Note that since we used a different feedback controller type to the NXTway-GS, the intergral feedback parameter k_5 was not needed. The feedback controller weightings are as follows:

Variable	**Gain**	**Value**
Motor Angle, θ	k_1	-0.8351
Gyro Angle, ψ	k_2	-34.1896
Motor Velocity, $\dot{\psi}$	k_3	-1.0995
Gyro Velocity, $\dot{\theta}$	k_4	-2.8141

Table 2.3: Feedback gains for the balance controller

The Matlab scripts to calculate the feedback values has been provided in Appendix A.13.

2.2.5 The Feedback Type

The difference between the NXTway-GS and Marvin controller lies with the type of feedback control type used in the controller. The NXTway-GS uses a PI controller, but only the motor angle is passed back through the integral part of the controller. The motor angle is used as the reference variable as the control task is to drive the robot around. The NXTway-GS controller has been constructed using Simulink and is shown below in Fig 2.5.

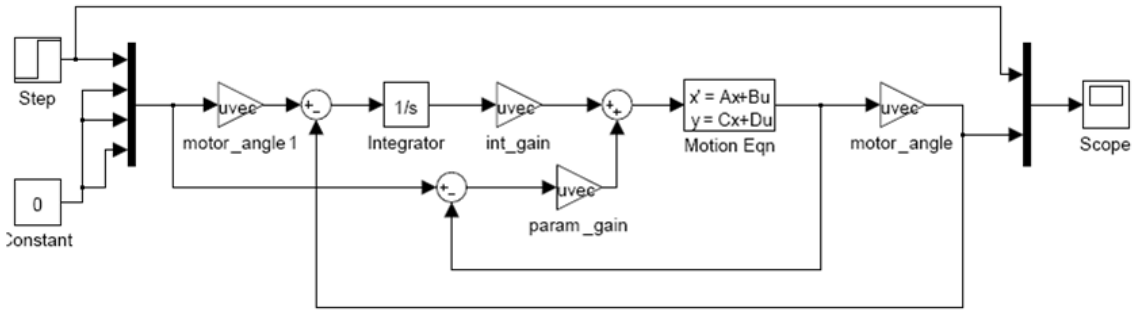


Figure 2.5: NXTway-GS servo controller block diagram

Marvin differs by using a PID controller to remove disturbance from the system. In the case of Marvin, all the feedback control signals are passed through the integral and derivative parts of the controller before it is applied to the plant. As seen from the diagram, the first component in the signal is the error $\theta_{ref} - \theta$. The control system for Marvin is shown in Fig 2.6.

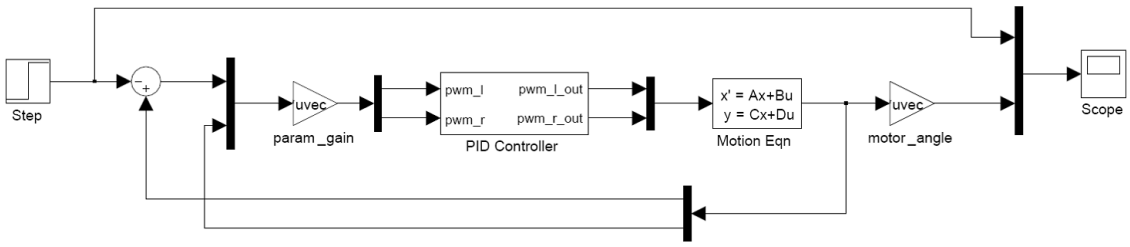


Figure 2.6: Marvin servo controller block diagram

Note that both systems use the same feedback gains and motion equations. This will allow for a comparative analysis between the two designs.

2.2.6 Analysing the Controllers

In order to compare both controllers fairly they will need to be tested under the same conditions. Therefore both controllers will be tested against the same unit-step reference signal applied to the motor angle. This will allow us to compare the performances of the two control systems, when the robot is commanded to move a distance corresponding to one unit of angle theta. Fig 2.7 shows a typical unit-step response of a control system showing the time domain specifications [29].

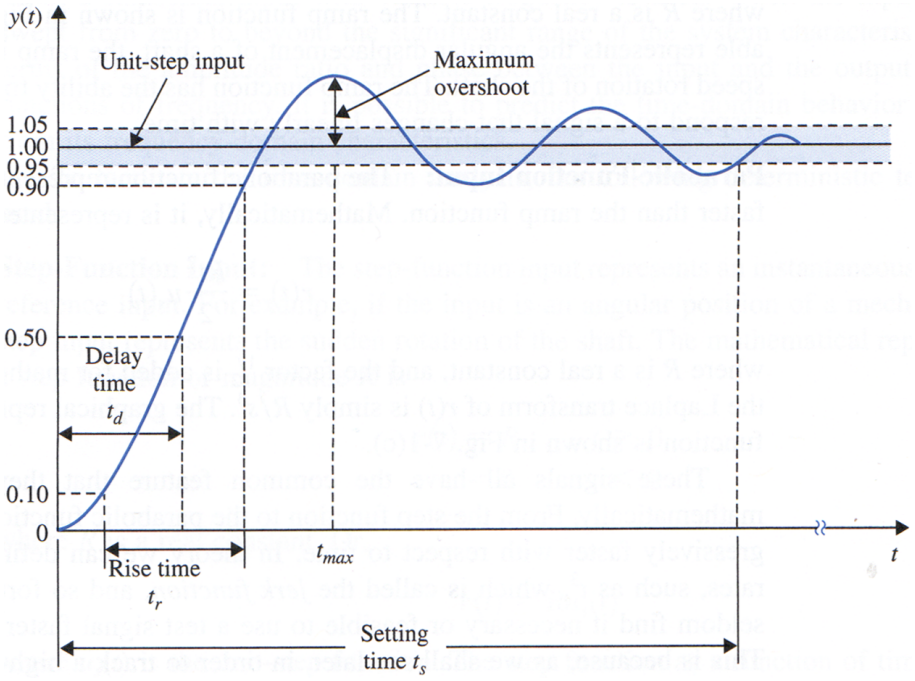


Figure 2.7: Typical unit-step response of a control system illustrating the time-domain specifications [29].

The performance criteria used to compare both controller outputs are defined as follows:

1. **Maximum Overshoot** – The maximum overshoot is a measurement of how far the controller output rises above the unit-step input. It is frequently used to measure the relative stability of a system [29]. Let m_{max} be the maximum motor angle, m_{ss} be the steady state value of the motor angle, then provided $m_{max} \geq m_{ss}$, the maximum

overshoot is defined as:

$$\text{maximum overshoot} \quad = \quad m_{max} - m_{ss} \tag{2.12}$$

It is often represented as a percentage of the final value of the step response, that is:

$$\text{percentage maximum overshoot} \quad = \quad \frac{\text{maximum overshoot}}{m_{ss}} * 100\% \tag{2.13}$$

In reference to a two-wheeled, balancing robot, the maximum overshoot represents how the robot will move past the desired upright position when removing a disturbance from the system. Therefore it is desired to have the percentage overshoot as small as possible.

2. **Delay Time** – The delay time represents the time taken for the step response to reach 50 percent of its final value.

3. **Rise Time** – The rise time is defined as the time required for the step response to rise from 10 to 90 percent of its final value. In reference to a two-wheeled, balancing robot, this represents the speed at which the robot can respond to an applied reference or disturbance. Ideally a small rise time is desired.

4. **Settling Time** – The settling time is defined as the time required for a step response to decrease and stay within a specified percentage of its final value. This analysis will utilise a commonly used figure of 5 percent. In reference to a two-wheeled, balancing robot this represents the time taken for the robot to remove the disturbance from the system to within a 5 percent margin of its upright position. A small settling time is desired for our robot.

The control systems for the NXTway-GS and Marvin can now be analysed against these performance criterion. The outputs of each of these systems were simulated using Simulink with the same unit-step response applied to the motor angle of the robot. The results for the NXTway-GS and Marvin are displayed in Fig 2.8 and 2.9 respectively. The Matlab scripts to produce the plots has been provided in Appendix A.14.

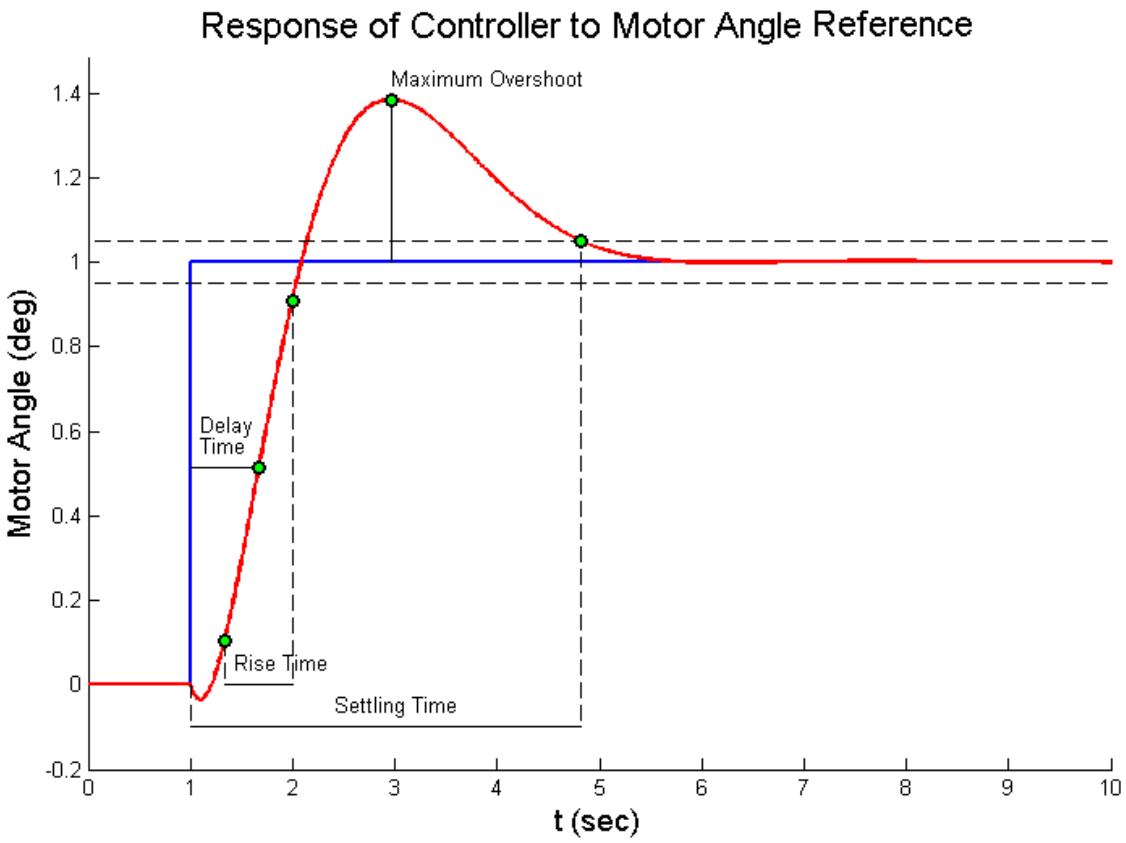


Figure 2.8: NXTway-GS response to the unit-step Motor Angle reference.

Step Response Characteristics	**Value**
Percentage Maximum Overshoot	38.46%
Delay Time	0.67s
Rise Time	0.66s
Settling Time	3.83s

Table 2.4: NXTway-GS step response characteristics

The most notable performance criteria of the NXTway-GS controller is the large overshoot the system has. The delay, rise and settling times show that the system removes disturbances quickly from the robot however it appears the consequences of having such a quick response is the large overshoot. This step response implies the NXTway-GS could be susceptible to large disturbances.

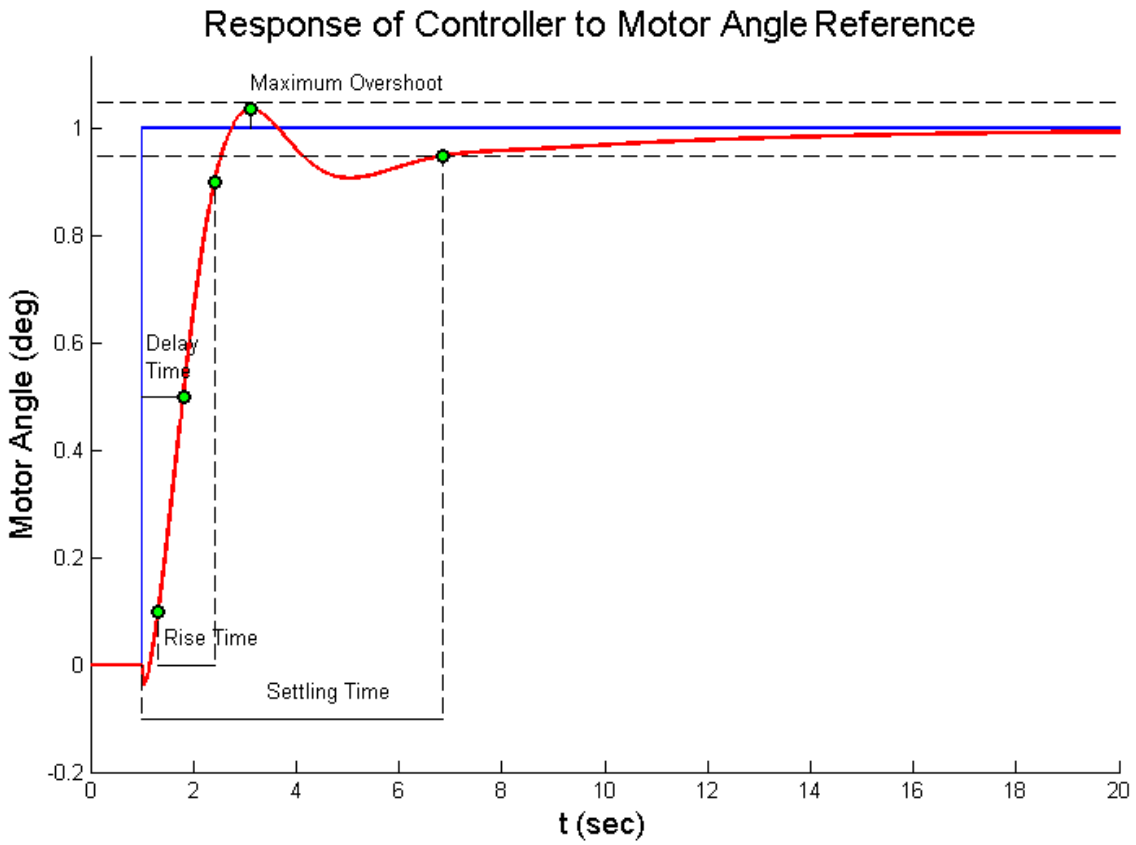


Figure 2.9: Marvin response to the unit-step Motor Angle reference.

Step Response Characteristics	**Value**
Percentage Maximum Overshoot	3.66%
Delay Time	0.81s
Rise Time	1.09s
Settling Time	5.85s

Table 2.5: Marvin step response characteristics

Marvin's control system shows a significant decrease in its maximum shoot, being approximately 10 times smaller than NXTway-GS's control system. The sacrifice of such a small overshoot, however, can be found with the delay, rise and settling time of the system being larger than that of NXTway-GS. These large times leave the balancing robot susceptible to multiple disturbances being applied to the system.

2.2.7 Selecting and Optimising a Controller for the GELway

Based on the information gained from analysing each of the control systems, for the NXTway-GS and Marvin, a decision can be made on the preferred system to implement in the GELway. After careful analysis Marvin's control system was decided upon since overall it provided the most benefits to the GELway. The benefits are as follows:

1. The control system provides a significantly lower overshoot when compared to the NXTway-GS. Taking into account that the model used in the simulation was a linearised model, and the real system is non-linear, it would be dangerous to employ a controller with a large overshoot.
2. Marvin's control system has been proven to work with the leJOS firmware.
3. The simulated model has already been converted into the Java programming language.
4. Due to the above advantages there is less risk in implementing this system since it has been proven to work on a LEGO balancing robot using the leJOS firmware.

With the control system now selected the next step is to attempt to optimise the control parameters to improve its performance. This project used two methods to track the improvements made to the controller, the first using the Nyquist method to check improvements to stability and the second comparing the step response characteristics as we did before between the NXTway-GS and Marvin. The process for determining the closed-loop system's stability using Nyquist is as follows [29]:

1. Determine the number of poles (P) and zeros (Z) in the real positive plane.
2. Check the number of zeros (Z) in the real positive plant equal zero.
3. Determine the encirclement direction of the Nyquist path.
4. Determine the number of critical point (-1, j0) encirclements (N).

5. For the closed-loop system to be stable, the Nyquist plot of the loop-gain transfer function must encircle the critical point as many times as the number of poles of the loop-gain transfer function are in the real positive plane. In other words:

$$N = Z - P \quad or \quad N = -P \tag{2.14}$$

The first part of this process used Matlab to find the loop-gain transfer function of the system. It was found to be:

$$Marvin_{cls} = \frac{16277.1923(s+55.39)(s+4.767)(s+4.57)(s+0.9261)(s+0.2087)}{s^2(s+241.2)(s+1000)(s-7.437)(s+6.52)} \tag{2.15}$$

Eq. 2.15 showed that the Marvin controller had only one pole in the positive plane at $P = 7.437$ and two poles at the origin. Since there are no zeros in the real positive plane we can continue to check for stability. Matlab was again used to determine the encirclement direction of the Nyquist path. It was found to be in a clockwise direction with the plot shown in Fig 2.10.

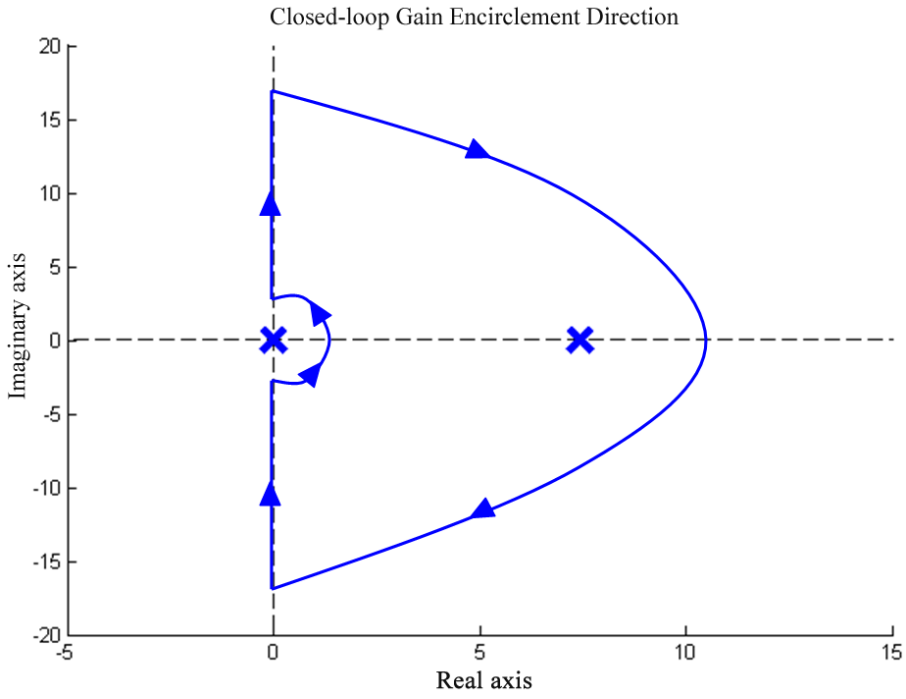


Figure 2.10: Encirclement direction of the Nyquist path

The final plot for the closed-loop system is the Nyquist plot which was done using Matlab's *nyquist* function. Fig 2.11 shows the Nyquist plot and with it the number of critical point encirclements was found to be -1. The negative sign represents that it was a counter clockwise encirclement.

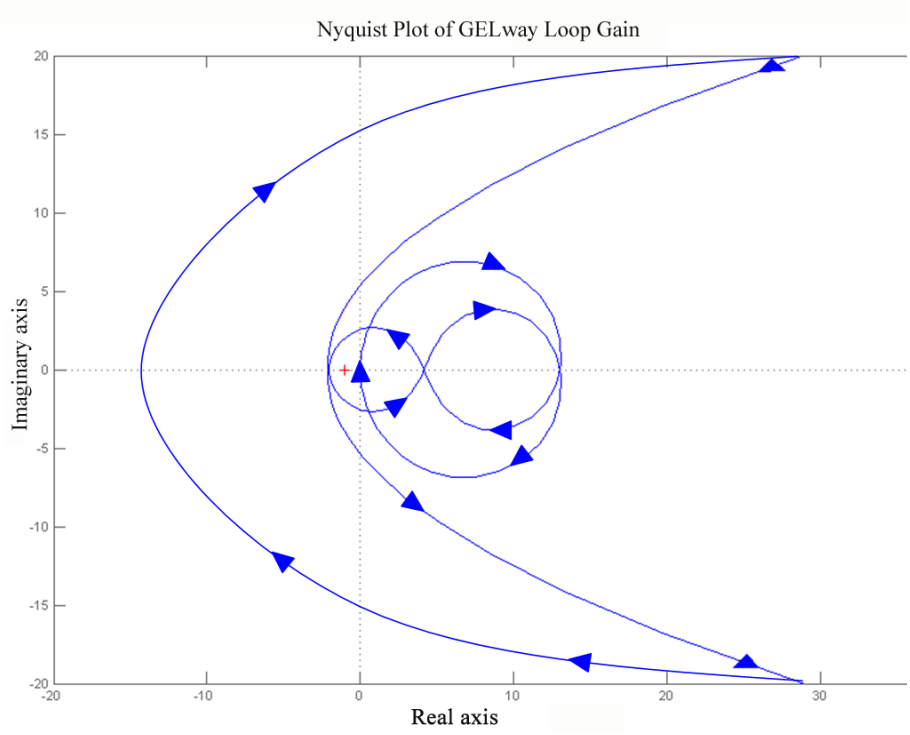


Figure 2.11: Nyquist plot of loop-gain transfer function

Summarising the results we found that:

1. Number of zeros on the real positive side was zero ($Z = 0$)
2. Number of poles on the real positive side was one ($P = 1$)
3. Number of encirlements of the critical point was negative one ($N = -1$)

Since $N = -P$ we determined that the Marvin's closed-loop system was stable. We return the Nyquist plot again after the controller parameters are optimised to determine if stability in the system has been improved.

The PID controller parameters were chosen to improve the performance of the GELway. Through trial and error the new PID parameters where selected and are shown in Table 2.6 against Marvin's parameters.

Control Parameters	**Marvin**	**GELway**
P	1.00	1.15
I	0.20	0.10
D	0.20	0.15

Table 2.6: PID Controller parameters

As mentioned before the advantage of creating the Simulink model was the fact that the PID controller parameters could be easily tweaked without the extensive time taken to run the simulations. The outcome of this optimisation was three new PID parameters that have improved the step response in the areas of overshoot, delay time and settling time. Fig 2.12 shows a plot of the GELway's step response with the step response characteristics summarised in Table 2.7.

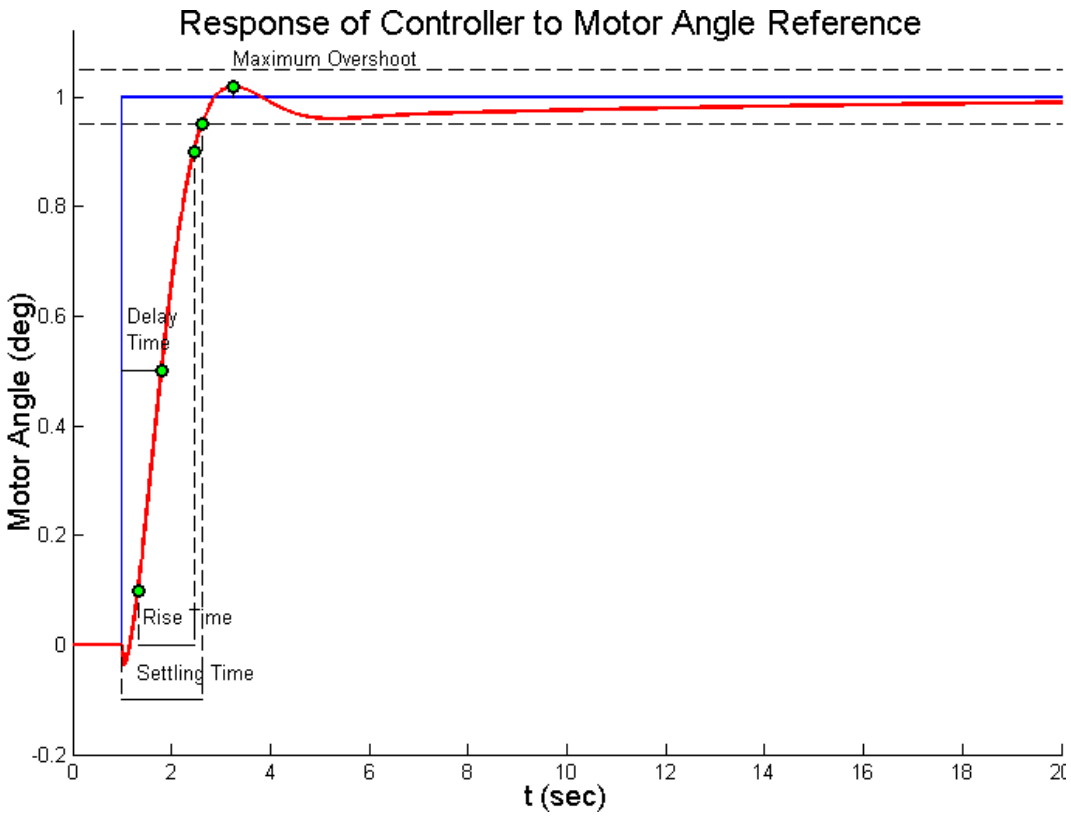


Figure 2.12: GELway response to Motor Angle reference.

Step Response Characteristics	**Marvin**	**GELway**
Percentage Maximum Overshoot	3.66%	1.92%
Delay Time	0.81s	0.80s
Rise Time	1.09s	1.13s
Settling Time	5.85s	1.61s

Table 2.7: GELway step response characteristics

The plot of the GELway step response in Fig 2.12, with the new PID parameters, shows a decrease from 3.66% to 1.92% in percentage overshoot, a slight decrease in rise time from $0.81s$ to $0.80s$ and a significant decrease in settling time from $5.85s$ to $1.61s$ when compared to the original Marvin PID parameters. Only the rise time has been negatively affected but only by $0.04s$ which is not a significant rise. These PID parameters provide a system that settles much quicker than the Marvins and with less overshoot. This performance will allow the GELway to recover from slightly larger disturbances more quickly when compared to Marvin the balancing robot.

The last step in comparing the new PID parameters for the GELway against Marvin was comparing the Nyquist plots to see if the Gain Margin had been increased. The Gain Margin is the amount of gain in decibels (dB) that can be added to the loop before the closed-loop system becomes unstable [29]. It is a measurement of distance from the critical point. As such a larger Gain Margin in the system means the control system is more robust. Fig 2.13 shows the Nyquist plot of GELway and Marvin.

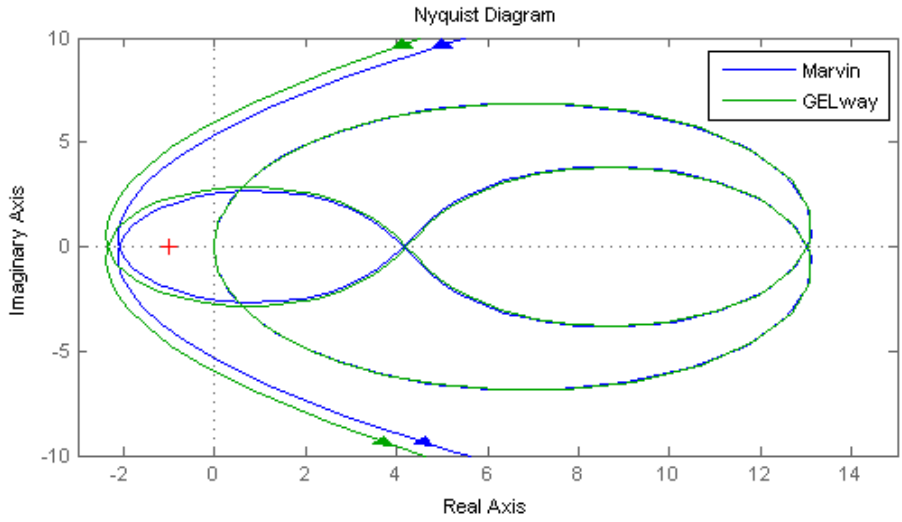


Figure 2.13: Nyquist plot of Marvin and GELway closed-loop system

Fig 2.13 shows that the new PID parameters for the GELway have resulted in the control system being less susceptible to perturbations then Marvins as its Nyquist plot is further away from the critical point $(-1, 0j)$. Therefore we concluded that the new PID parameters for the GELway have improved its performance when compared to Marvin due to its reduced overshoot and settling time and its increased stability.

To build the transfer functions required for the Nyquist plots in this section the *GELway-TransferFunction* Matlab script was used with the source code found in Appendix A.15.

2.3 Balancing the Robot

The intention of this section is to describe the process of implementing the control system design presented in the previous section. Since the decision was made to use the Marvin software, the first stage involves loading the software onto the robot to determine if it is able to balance our robot. The software will be reverse engineered to gain a better understanding of how it functions so that we can include additional improvements to our robot.

2.3.1 Marvin - The Balancing Robot

Marvin was selected as a good starting point for designing the GELway as it is one of the few balancing robots which was programmed using the Java leJOS firmware. The designers had also published videos on YouTube showing the robot successfully balancing and moving around following Bluetooth inputs from a laptop computer [28].

The Marvin Java software was loaded onto the NXT using Eclipse to first see if it was capable of balancing the GELway. After approximately 1 second the GELway would lose its balance and crash to the ground. Since the Marvin software was not able to keep the GELway upright for an extended period of time careful analysis of its balancing methods was carried out in order to pinpoint the problem. Improvements for the GELway software were also of interest during the analysis of Marvin's balancing mechanism. The *BalanceControl* class was analysed first as it contained the PID controller used to keep Marvin upright. The calculations to find the parameters used in the PID controller were not provided in the Marvin documentation and therefore there was a chance that these values could be the cause of the GELway's instability [28]. This was ruled out however since the build design of both robots was the same, meaning their equations of motion would also be the same. This was the first limitation found with the Marvin software, as it had only been programmed to balance with the RCX motorcycle wheels. GELway will improve on the Marvin software by being able to balance with different wheel types and different wheel positions. Other issues with the balance control system include the necessity of restarting the program every time the robot falls over and that the robot cannot sustain an upright position if there is any slip

in its wheels. Since the *BalanceControl* class was not found to be the cause of the GELway's instability analysis was then conducted on the *GyroscopeSensor* class.

Initial analysis of the *GryoscopeSensor* class showed two parameters that appeared to be specific to the Marvin robot. The first parameter was the gyroscope sensor offset value *lastOffset*, which was set at 595.5, with the second being parameter *a*, which is intended to compensate for the gyroscope sensors drift. These two parameters appeared to be specific to the gyroscope sensor used for the balancing robot and were possibly the key to getting the GELway to balance. This lead to several tests being conducted to find the offset value specific to the gyroscope used on the GELway.

2.3.2 Gyro Sensor Offset

According to the manual that came with the HiTechnic gyroscope sensor, the gyro offset was approximately 620 [31]. Utilising this information the variable *lastOffset*, in Marvin's *GyroscopeSensor* class, was changed from 595.5 to 620. By changing the offset value to 620 the GELway was able to maintain balance successfully; however it gradually moved forward over time. This meant that the offset was too large, and that a smaller offset value was required. An offset value of 615 was then tested, which caused the GELway to gradually move backwards. Therefore it was concluded that the initial offset value was between 615 and 620. Whilst it may have been possible to eventually find the initial offset value for the GELway's gyroscope sensor through trial and error, there would be a severe limitation in testing how the gyro sensor offset was affected over time. Therefore a decision was made to implement a data logging method to the NXT which would send information from the NXT to a computer.

2.3.3 Calculating Gyro Offset - Data Logging

The first step in accurately calculating the gyro sensor's offset is to create a simple loop which reads the stationary gyro sensor offset value a number of times and averages all the values read. The method *calcoffset()* was added to the *GyroscopeSensor* and is called when the class is first initialised. The method process can be seen below in Fig 2.14.

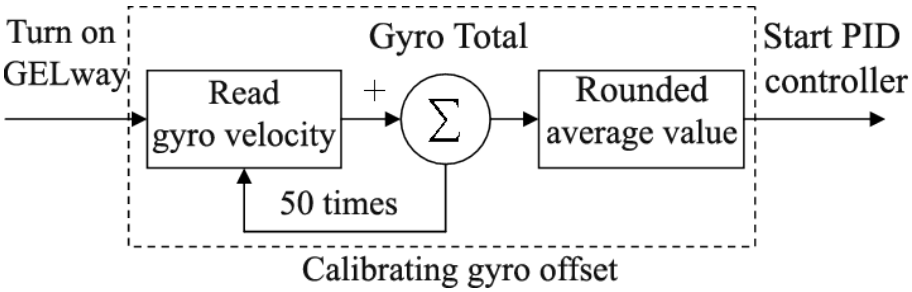


Figure 2.14: The process for calculating gyro offset.

Reading the gyro only once is not sufficient since the gyro readings can vary by ± 1, and thus an average is needed.

With the inclusion of this method, the initial offset value can be calibrated for any HiTechnic gyro sensor, which improves on the original Marvin software. Now that the initial offset value has been calculated the NXJ Datalogger can be used to record the gyro offset value over time. The NXJ Datalogger is a Java program which comes with the leJOS download folder provided by the leJOS developers [21]. Information on how to use the NXJ Datalogger, as well as its capabilities can be found in the tutorial document provided by leJOS [21]. In brief, the NXJ Datalogger allows the NXT to connect and send data with a computer either by USB or Bluetooth.

The GELway was then programmed to read the stationary gyro sensor value average over 20 second intervals. At the end of each 20 second interval the offset value was also recorded. This was done because the gyro sensor only outputs an integer value, and it can be deceiving to only work with the gyro's average value. The results can be seen in Fig 2.15 which shows the plot off the gyroscope sensor drift.

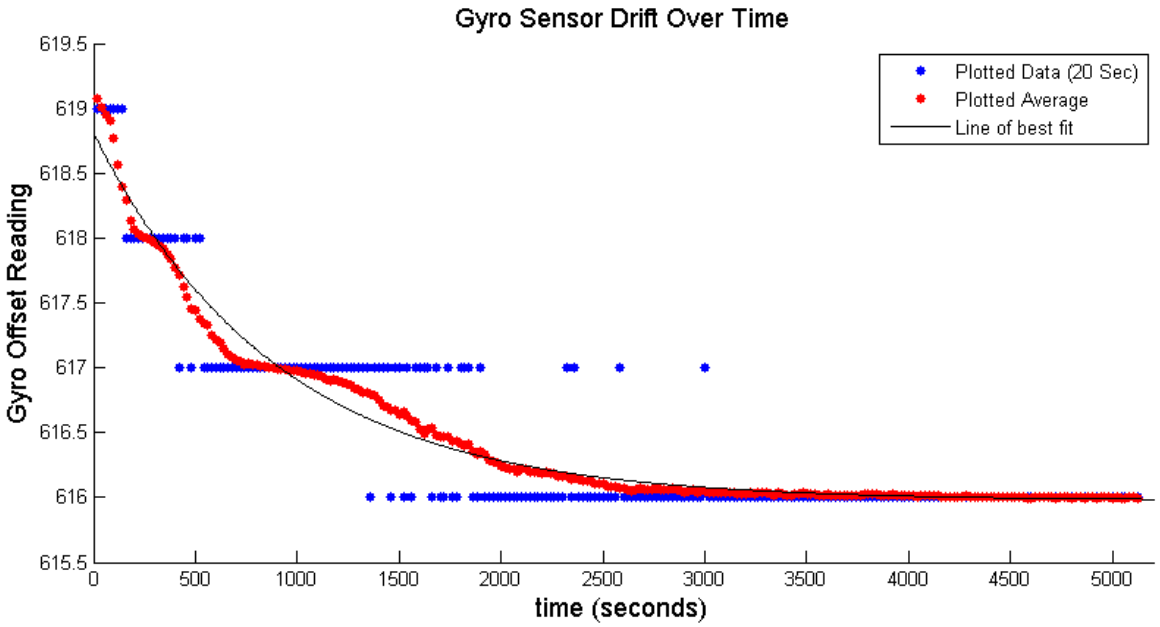


Figure 2.15: Gyro sensor drift over time.

Fig 2.15 shows that the gyro sensor drift appears to decrease exponentially over time. The blue points represent the gyro reading at 20s intervals whilst the red shows the average gyro readings for a 20s period (read at 4ms intervals). The decay in the gyro offset will cause the GELway to gradually move backwards over time. This exponential decay is accounted for by using a recursive filter [28]. The filter is used each time a value is read from the gyro sensor. The equation used is:

$$Offset \quad = \quad a \times O(k) + (1 - a) \times O(k + 1) \tag{2.16}$$

where $O(k + 1)$ is the current offset value being read and $O(k)$ is the previous. The gyro offset is constantly updated by adding a percentage of the previous and current gyro offset.

2.3.4 Programming the Balance Control System

This section will explain how the simulated control system proposed in section 2.2 is programmed into the GELway using the leJOS programming firmware. The balancing mechanism can be found in the *BalanceController* class. This will be given in steps, presented in the same order they appear in the *BalanceController* class.

1. The constants for the PID controller and weighted error constants are set when the program is first initiated. These values are set as final as they will not change.

2. The next step is to create the gyro sensor and servo motor objects. These objects contain the methods to read the sensor and motor angles and velocities. Note that the gyro sensor angle cannot be read directly, and as such, its velocity is integrated over time.

3. An infinite loop is then started that will run continuously whilst the GELway balances.

4. The first step in the loop is to check if the GELway has been reset. If so the balancing loop is paused, all the balance parameters are reset, and the program waits until the balancing process is restarted.

5. Provided the GELway has not been reset, the sensor and motor values are then read for the controller plant.

6. The error value of the controller is then calculated by using the gyro and motor values and multiplying them by their respective parameter weights.

7. This error value is then passed through the PID controller, which will try to correct the error to the desired set point, which is the robot in the upright position.

8. The output of the PID controller is sent to each motor, causing the motors to rotate at the desired speed. A negative output of the PID controller will result in the motors rotating backwards.

9. Since the gyro sensor has an operating frequency of 300 Hz, the balancing loop is delayed to stop it from being read too frequently. The length of the delay determines

how quickly the controller can respond to a physical disturbance such as a push, and will be looked at in more depth in Section 2.3.5.

10. The balancing loop then goes back to step 4, restarting the balancing process once again.

2.3.5 Balance Control Loop Delay

As mentioned previously in section 2.3.4, a delay had to be programmed into the balance control loop in order to ensure the gyro sensor was not being read too frequently. If the delay is too short, the balance control system will not function correctly and the robot will fall over. If the delay is set too long, it will slow the time taken to respond to a disturbance. In order to test the effect of the delay a data logging procedure was added to the *BalanceController* class.

The data logging procedure works by storing the output of the PID controller for each balance loop iteration in an array. In other words the speed sent to the motors is recorded. After a specified amount of data is recorded, the balance procedure is paused, and a connection with the laptop is established. Using the Data Viewer GUI, provided by leJOS, the data can be transmitted to the laptop, and plotted using Matlab. The reason why the information is sent as a whole at the end of the data logging, and not whilst the robot is balancing, is that the Bluetooth connection is not fast enough to transmit data every 4-20ms, and is only capable of transferring data approximately every 40ms [21]. A delay speed of 20ms was first selected, with subsequent tests reducing the value to find the optimum delay.

Balance Control Delay - 20ms

Fig 2.16 shows the GELway control system's response to disturbance when the delay is set at 20ms. The dashed line in Fig 2.16 shows the point at which disturbance is applied, and shows it takes approximately 1000 loop iterations to remove the disturbance. Whilst a delay speed of 20ms is capable of balancing the robot, it is impractical to use since the controller takes an extended period of time to recover from a disturbance.

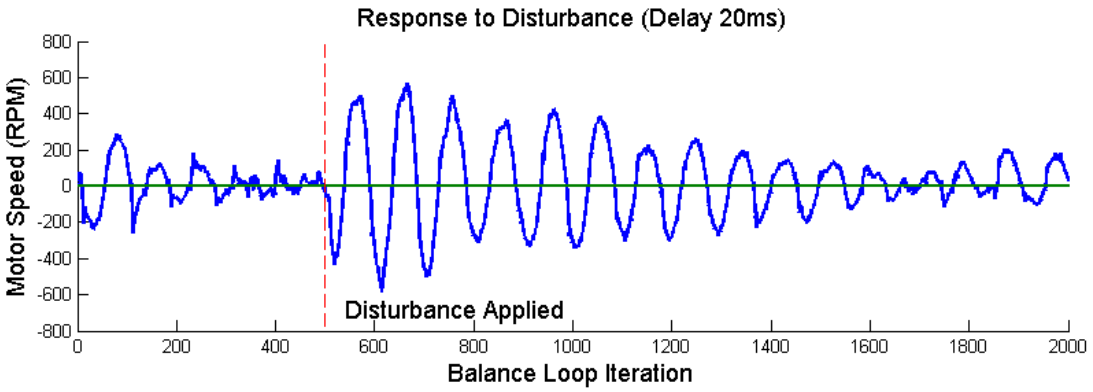


Figure 2.16: Response to disturbance (Delay 20ms).

Balance Control Delay - 10ms

Fig 2.17 shows the GELway control system's response to disturbance when the delay is set at 10ms. When compared to a 20ms delay we see that the controller responds to the disturbance much faster, taking approximately 400 loop iterations to return to the desired set point. A delay of 10ms is capable of keeping the GELway balanced and removes disturbance from the system much faster than a 20ms delay.

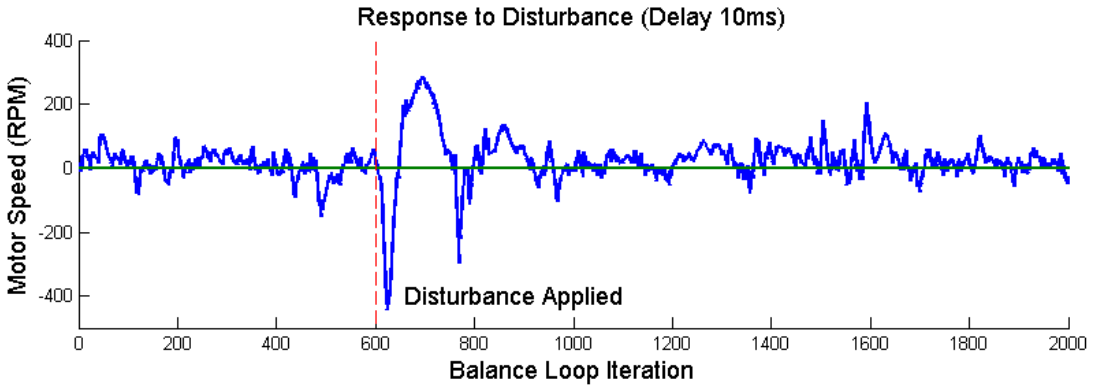


Figure 2.17: Response to disturbance (Delay 10ms).

Balance Control Delay - 6ms

Fig 2.18 shows the GELway control system's response to disturbance when the delay is set at 6ms. At a delay of 6ms the control system is able to remove an applied disturbance in approximately 200 loop iterations. This is the fastest response when compared to the previous tests.

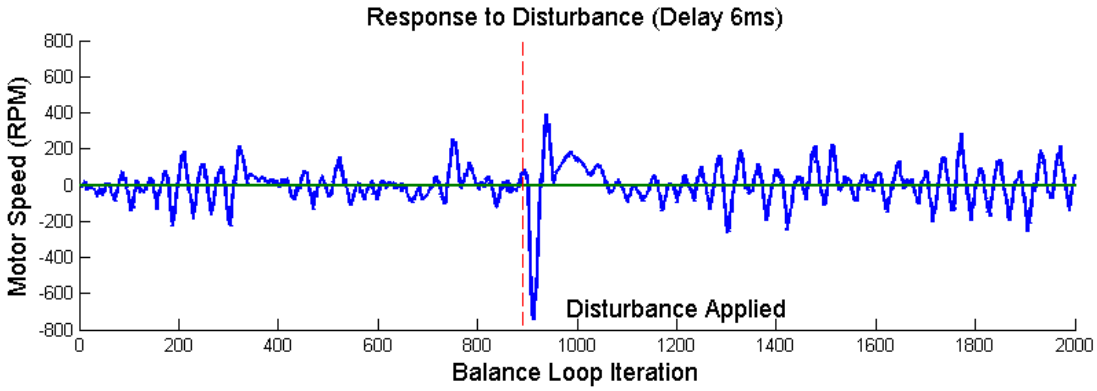


Figure 2.18: Response to disturbance (Delay 6ms).

Delay speeds of less than 6ms cause the GELway to crash, and thus 6ms was selected as it provided the fastest recovery from disturbance.

2.3.6 Resetting the Balance Mechanism

An important feature that needs to be added to the GELway is the ability to reset it once it falls over. With the current design, once the GELway falls over, the program needs to be shut down and be restarted, and due to the need of connecting to a Bluetooth device each time the program is started the process can be very time consuming. In order to rectify this, a reset method was added to the *BalanceController* class which allows the parameters and sensors to be reset. The procedure is shown below in Fig 2.19.

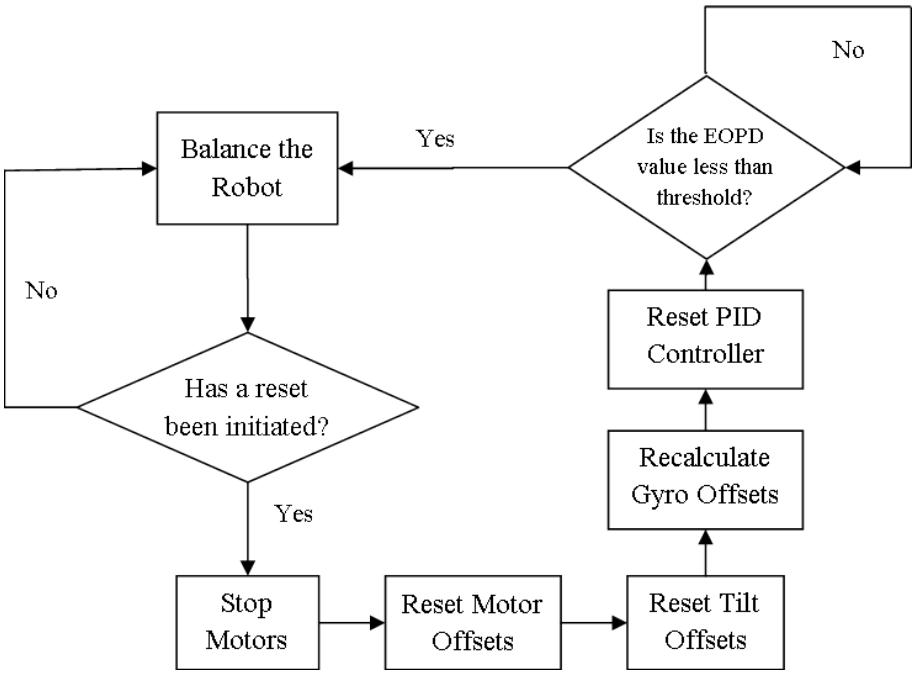


Figure 2.19: GELway self-reset procedure.

The reset procedure can be initiated in three different ways:

1. By manually pressing the 'Back' button on the GELway.
2. By sending a Bluetooth integer of '0'.
3. By using an EOPD sensor to detect if it is facing the ground, since if the GELway falls over, the EOPD sensor will be facing horizontally, unable to detect objects.

2.3.7 Firmware Upgrade Issues

The balancing robot known as Marvin was only intended to work with leJOS firmware 0.7. However during the timeframe of this project, significant upgrades were made to the leJOS firmware, in which two main upgrades directly affected the GELway. The first noticeable upgrade was that the runtime speed of the GELway's programs had been significantly increased. This caused the balancing mechanism in the GELway to fail since the gyro sensor was being read at a speed faster then it could handle. To understand why this is an issue, we first consult the HiTechnic manual of the gyro sensor.

The gyro sensor has an operating frequency of $300Hz$, which means that it is capable of being read approximately every $3.33ms$ [31]. The creators of Marvin estimated that their *BalanceControl* class took approximately $0.33ms$ to run, and as such, they placed a $3ms$ delay at the end, to ensure the gyro was not being read past its operating frequency.

```
start loop
   read gyro sensor and motor data
   pass values through PID controller
   send power to motors
   wait 3 ms
restart loop
```

With the new firmware the *BalanceControl* class's operating runtime became less than 0.33ms causing the gyro sensor being read too quickly, which resulted in the GELway crashing after a few seconds of balancing. The issue was solved by increasing the delay at the end of the balancing loop to a value greater than 3ms, thus making certain the gyro sensor was not being read past its operating frequency.

The reason why we spent time rebalancing the robot with leJOS firmware version 0.8 and not simply reinstalled a perfectly working leJOS 0.7 was the second improvement the upgrade provided. The second upgrade was a fully functional behavioural API class. The class file was flawed in firmware version 0.7, and since implementing a behaviour system into the GELway was a goal in this project, the upgrade was necessary.

2.4 Testing the Balance Control System

This section will analyse how the GELway's controller performs in a real world system. The leJOS data logger will be utilised to track the GELway's sensor and motor readings whilst it remains balanced. This analysis will show the weight each balance parameter has in the controller, and will show the effect the PID controller has on the measured values. The balance parameters that were analysed are the:

- Gyro Angle,
- Gyro Velocity,
- Motor Angle, and
- Motor Velocity.

The control system was tested by recording the balancing parameters and the output of the PID controller for each balance loop iteration. The results were plotted using Matlab, along with the combined error signal that was input to the PID controller, which allows a comparison of each measured value. Fig 2.20 shows each of the measured balance parameters. Note that these parameters have already been weighted by their respective weights displayed in Table 2.3 so that they can be compared against each other.

First, looking at the scales of Fig 2.20(a)-(d) when compared to the combined error signal in Fig 2.20(e) we see each parameter's significance. The motor angle and velocity scales show they have a small impact on the combined error signal, whilst the gyro angle and velocity have a much larger impact. In fact, the gyro angle plot in Fig 2.20(a) appears almost identical to the combined error signal. When these results were first produced, it seemed possible that the other parameters may not affect keeping the GELway upright at all. A test was conducted keeping only the measured gyro angle in the control system, and the robot was only able to remain balanced for a few seconds. Whilst the plots in Fig 2.20(c)-(d) may not show a significant effect on the output of the PID controller, they are important in keeping the GELway upright.

Secondly, when comparing Fig 2.20(e) and Fig 2.20(f) we see how the PID controller smooths the error signal providing a much more constant speed to the motors. Without the PID controller the servo motors would change direction much more rapidly causing the robot to oscillate more until the GELway eventually falls over. It is also worthwhile mentioning that there was no reference input to the system, and the response measured was due to the initial conditions, not ideal zeros, hence the robot would fall if it was not controlled.

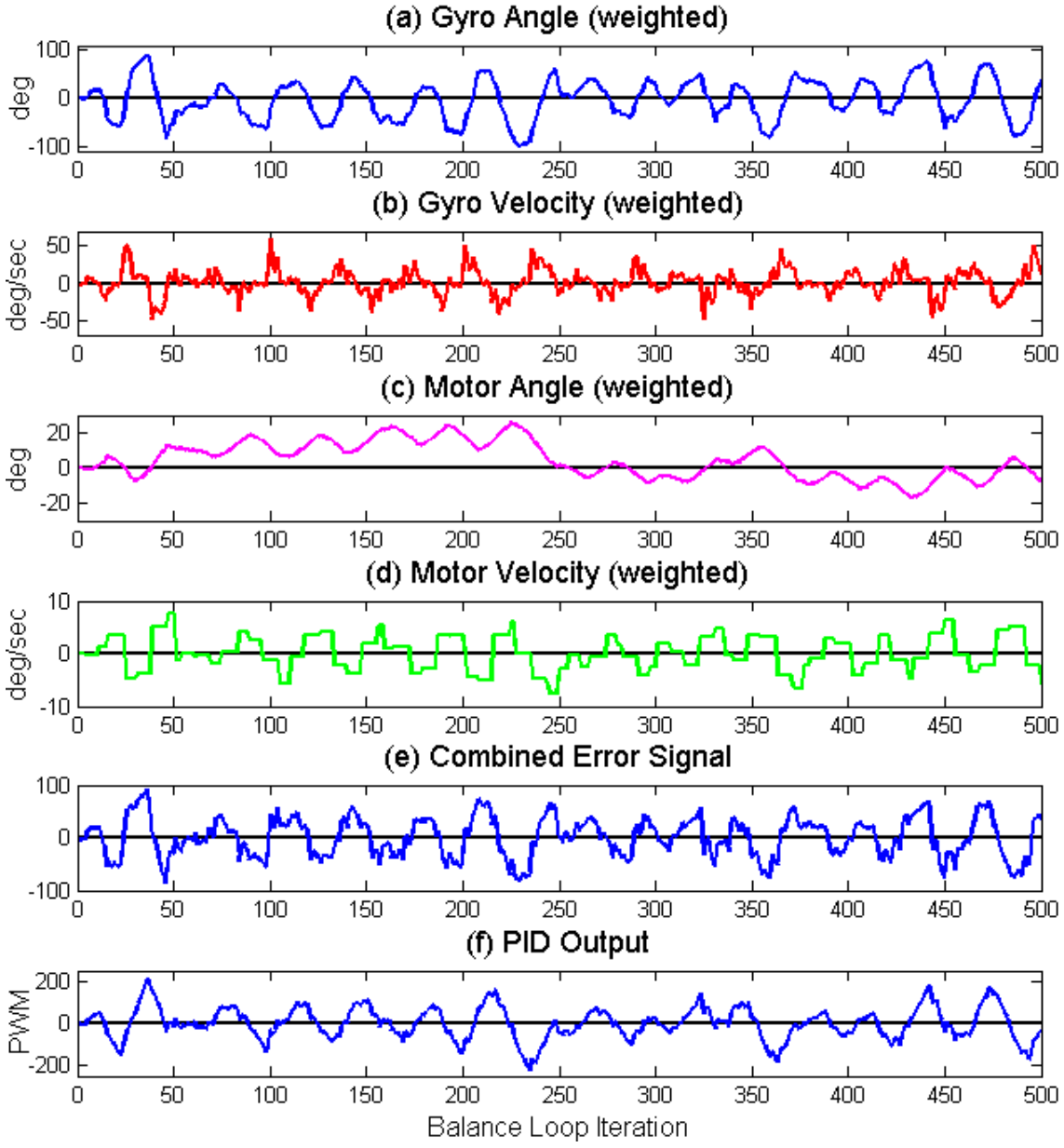


Figure 2.20: GELway balance control system parameter plots.

2.5 Moving the Robot

This section will explain the control mechanism that handles GELway's movements and how the forwards, backwards, left and right commands have been programmed into the robot.

2.5.1 Moving the Robot Forwards and Backwards

Moving a balancing robot forwards is much harder than simply applying more power to the motors. Whilst a three and four-wheeled vehicle can be driven by applying a set speed to the motors, a two-wheeled balancing vehicle needs a much more intuitive method, since its motors are constantly moving due to the robot's instability.

Background Theory

If we look back to the output of the PID controller in Fig 2.21 we see that the average voltage applied to the motors is approximately zero, hence the robot will stay in a stationary position. However, if the average voltage applied to the motors is greater than zero, the robot will move forwards, and an average voltage less than zero causes a backwards movement.

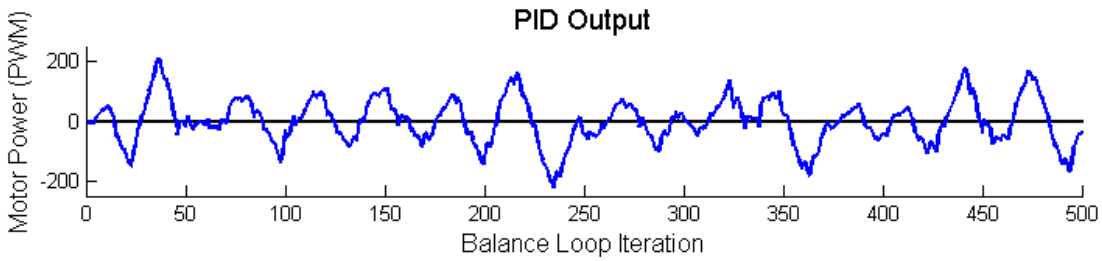


Figure 2.21: Output of the PID controller.

Looking at Fig 2.21 we see that being an inverted pendulum, the robot stabilizes by moving its wheels to reduce disturbance body pitch ψ. Hence, supplying a constant reference body pitch angle ψ would cause it constantly move.

If we remember back in Fig 2.20 when each of the balancing parameters were analysed we

noted that the measurement of the gyro sensor angle had the most significant effect on the combined error signal. Therefore if we provided an offset to the body pitch angle, for instance, we could provide the controller with averages greater and lesser than the zero point, thus providing the robot with forwards and backwards movements, without effecting the main balancing parameters.

It makes sense to be offsetting the body pitch angle if we think about what the motor is actually measuring. Fig 2.22 shows that the angle being measured is related to the robot in the upright position. Therefore if we offset the body pitch angle, we would cause the robot to balance slightly off centre resulting in the robot compensating the offset by moving forwards or backwards.

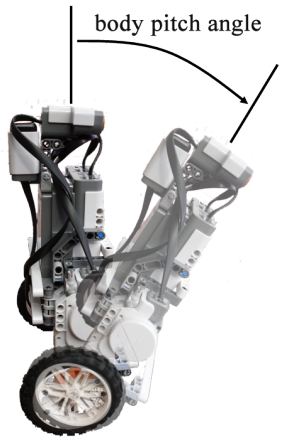


Figure 2.22: Measuring the Motor Angle.

Programming the Forwards and Backwards Movements

Programming an offset into the body pitch angle is relatively simple and is achieved by altering the motor angle value when it is read in by the motor tachometer. However this task became more difficult in practice, and simply offsetting the motor angle proved unsuccessful for two main reasons. First, if the motor is offset by too large a value, the robot will oscillate and crash. Think of this like an instant large prod to the robot. To overcome this problem the offset was applied gradually to the motor angle over time, resulting in a much gentler

forward movement. The second issue was that the robot would move forward only for the period of time when the motor offset was being applied. However this problem was a bit more difficult to overcome since it is the nature of the PID controller to remove any constant disturbance applied to the system. There were two ways to solve this issue:

1. Continuously apply an increasing offset, or
2. Only allow the robot to move gradually.

The second option was chosen as the robot could still move continuously whilst it was receiving commands from a Bluetooth device. If no commands were being sent the GELway would balance in a stationary position. This was seen as a good safety mechanism, with the robot only moving when it was receiving commands.

Plotting the Forwards and Backwards Movements

The forwards and backwards movements are visually represented in Fig 2.23, showing the current gyro offset, and the power being sent to the motors.

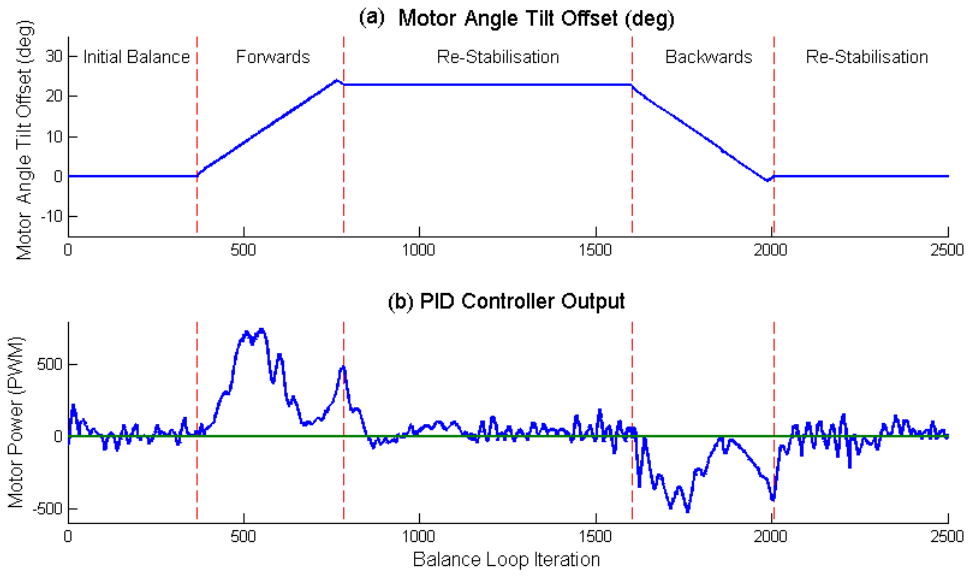


Figure 2.23: Forward and backwards movement plots for the GELway.

In Fig 2.23 we see that when a positive motor offset slope is applied to the controller the average power sent to the motors is greater than zero, thus causing the GELway to gradually move forwards. The GELway moves backwards when a negative motor offset slope is applied. The PID controller restabilises the GELway when the motor offset is constant removing any constant disturbance from the system. Note that a slight offset was provided in the opposite direction of movement just before the robot stabilises, as this helps prevent overshoot in the controller.

2.5.2 Rotating the Robot

Rotating a two-wheeled, balancing robot is far easier than moving it forwards and backwards. This is because all that is required is to rotate one wheel faster than the other.

Programming the Rotation Movements

Programming left and right turns into the GELway is achieved by setting a PWM voltage offset in each of the motors, so that one motor receives more power than the other. The resulting effect is the robot rotating on the spot, giving two-wheeled robots the unique characteristic of a zero turn circle. This is visualised in Fig 2.24.

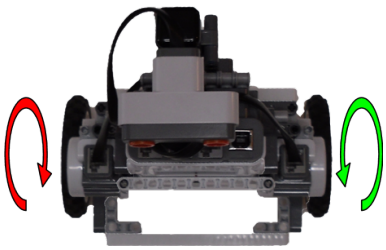


Figure 2.24: Rotating a two-wheeled, balancing robot.

Plotting the Rotation Movements

Fig 2.25 shows the power sent to each of the servo motors attached to the GELway. When the right rotation command is given to the GELway, the left is increased by 200 PWM and the right motor is reduced by 200 PWM. This causes the left wheel to rotate faster than the right, making the GELway turn in the right direction, and vice versa for a left rotation.

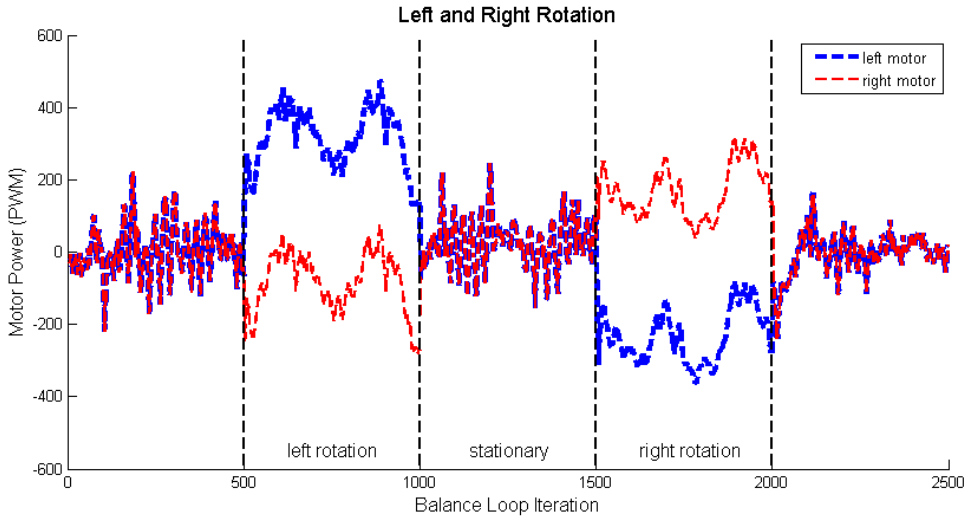


Figure 2.25: GELway rotation movement plots.

Note that the robot can become unstable during the rotation making it more difficult for the controller to remove disturbance due to other forces acting on the robot, such as centrifugal force.

2.5.3 GELway MotorDirection Class

All the methods that handle the GELway movements have been combined into one class called *MotorDirection*. Most of the GELway behaviours utilise these methods to move the robot around.

2.6 Programming Structure

The GELway's functionality is handled by three programming threads that allow the robot to perform multiple tasks concurrently. A thread allows a program to process information in parallel, thus allowing the GELway to respond to events whilst still remaining balanced. The three main threads for the GELway are the:

1. **BalanceController** - this thread has the highest priority and ensures the GELway remains in the upright position. It contains the PID controller used to keep the GELway balanced by using readings from the gyro sensor and motors. The *BalanceController* thread was covered in this chapter.

2. **Arbitrator** - this thread handles all of the GELway behaviours. The arbitrator decides which behaviours should be activated when certain events happen to the robot, such as approaching an obstacle. The behavioural system will be discussed in more depth in Chapter 3 - Behavioural Programming.

3. **BluetoothReader** - this thread allows the GELway to ensure a constant connection is kept with the Bluetooth remote controller. The *BluetoothReader* class reads integers sent from a laptop or mobile phone and triggers events in the GELway such as moving in a certain direction. The Bluetooth connection is covered in Chapter 4 - Communication Between Heterogeneous Robots using Bluetooth.

These threads are initiated by the main GELway class, and call on several other classes to process and read in data. The complete class diagram is shown in Fig 2.26.

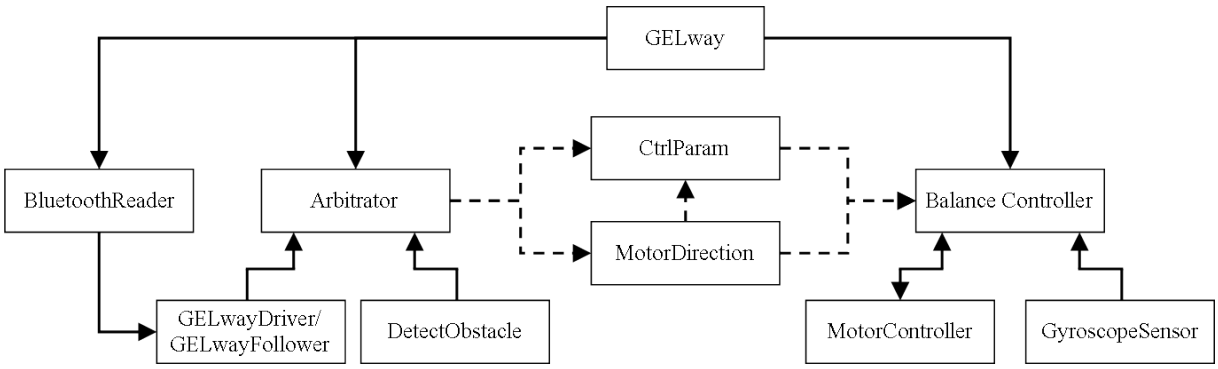


Figure 2.26: GELway class diagram

2.6.1 Class Descriptions

A brief description on each class will be given, detailing what function it provides to the GELway. The relationship between each class is shown in Fig 2.26.

- **GELway -** this class is used to initiate the *Arbitrator*, *BalanceController* and *BluetoothReader* threads. It also establishes the Bluetooth connection between the GELway and either a mobile phone or laptop as well as the master/Slave Bluetooth link. The source code for this class has been provided in Appendix A.2.

- **BalanceController -** this thread, as described previously, contains the PID controller that keeps the GELway balanced. The calculation of the balancing parameters and how the class works is covered in more detail in this chapter. The source code for this class has been provided in Appendix A.3.

- **MotorController -** The *MotorController* class is used to set the power to the motors in order to keep the GELway balanced. The motor power is calculated from the PID controller in the *BalanceController* class. As a secondary function this class also calculates the current motor velocity and motor angle, which are needed by the plant in the PID Controller. The source code for this class has been provided in Appendix A.4.

- **GyroscopeSensor -** this class calculates the current gyro sensor angle and velocity which are needed by the plant in the PID Controller. It also calculates the offset which

is specific to the attached gyro sensor. The source code for this class has been provided in Appendix A.5.

- **CtrlParam -** this class is used for communication between the *Arbitrator* and *BalanceController* threads. It has the control parameters which allow the GELway to move, such as offsetting the motor speeds and the gyro sensors tilt angle. The source code for this class has been provided in Appendix A.6.

- **MotorDirection -** this class provides the methods which move the GELway around. It contains forwards, backwards, left and right commands, and alters values in the *CtrlParam* class. The source code for this class has been provided in Appendix A.7.

- **Behaviour -** the *Behaviour* class is implemented from each of the GELway behaviours and contains methods which decide when a Behaviour should take control and what function it will provide. This class is a part of the leJOS API.

- **GELwayDriver -** the *GELwayDriver* behaviour uses the *BluetoothReader* thread to get integers sent from an external Bluetooth device to drive the robot around. It is the lowest priority behaviour and is always running unless it is suppressed by a higher level behaviour. The source code for this class has been provided in Appendix A.8.

- **GELwayFollower -** the *GELwayFollower* behaviour is used in the follower robot in place of the *GELwayDriver* during the motor synchronisation of two robots. It contains a distance controller that keeps the GELway at a set distance from a given object. The source code for this class has been provided in Appendix A.9.

- **DetectObstacle -** this is another behaviour class for the GELway and is used to stop the GELway from running into obstacles. It is at a higher level to the *GELwayDriver*, and as such, will suppress it when an obstacle is detected by the ultrasonic sensor. The source code for this class has been provided in Appendix A.10.

- **BluetoothReader -** this thread was created so that the GELway can always receive commands from an external Bluetooth device. It has been designed so that a sent integer is only read once by the program. The source code for this class has been provided in Appendix A.11.

2.7 Chapter Conclusion

This chapter has provided the basic framework and structure of the balancing robot that will be used to further investigate its ability to communicate and carry out behaviours in the proceeding chapters. Since the balancing robot is well researched, the start of this chapter focused on understanding the work that had been carried out previously and then working on improving these designs. This was done by first simulating and improving the control systems and then implementing them into the actual robot. By following documented workings of previous balancing robot designs we were able to successfully balance our own robot and add additional functionality such as a self calibrating method for the gyro sensor offset and the self reset mechanism for when the robot falls over.

Chapter 3

Behaviour Programming

This chapter will discuss the behavioural system that has been implemented into the GELway that will allow the robot to act autonomously and make its own decisions based on its surroundings. The behavioural system has been designed to be scalable, meaning that future functionality into the robot can be easily added, such as search and rescue missions and mapping out a room. The behavioural system runs in parallel with the balancing thread described previously in Chapter 2. The intention of this chapter is to prove the concepts of the behavioural system by using several basic robotic behaviours and illustrate its main features.

This chapter begins with a description of the behavioural system, in section 3.1, covering a detailed explanation of the behaviours that have been used to prove the concept of the system. These include an obstacle avoidance behaviour that prevents the GELway from colliding with objects and a driving behaviour that allows an operator to control the GELway's movements. The driving behaviour will utilise the communications system developed in Chapter 4 and will use commands sent by an operator to action movement methods in the robot. In Chapter 5 an additional behaviour has been added, which replaces the driving behaviour, and allows the GELway to follow another leading balancing robot at a set distance. Following this in section 3.2, an introduction to the leJOS behavioural package will be given, covering the methods in each behaviour and the arbitration system that uses subsumption architecture to select which behaviour should become active.

Section 3.3 will detail how the behavioural system was implemented into the GELway robot and will detail how the obstacle avoidance and GELway driver behaviours were programmed. The behaviours designed for the behavioural system are largely based on moving the GELway around and as such needed to modify the balancing system parameters. This raised some

concurrent programming issues if both systems were trying to access the same data at the same time, but was overcome by synchronising all shared parameters. Section 3.3.1 explains how this was achieved. Several issues were also encountered when designing the behaviours, such as infinite loops causing the behavioural system to fail and the need to prioritise the behaviours correctly. Section 3.3.3 covers the solutions to these issues.

Lastly, section 3.4 shows the testing that was carried out on the implemented behavioural system which demonstrated the behaviour suppression mechanism in the robot by allowing higher priority behaviours to suppress lower priority behaviours. This is important to self-configurable robots as it allows them to adapt to their environment by using behaviours specific to their environmental input.

3.1 Description of the Behavioural System

Behaviour-based programming in robots enables them to attend simultaneously to hazards they may face as well as unexpected opportunities they may come across [20]. With a behavioural system the robots are capable of making their own decisions based on environmental inputs, which allow them to act differently under different circumstances rather than do the same thing all the time. Our main aim with the behavioural system was to prove that the concept will work in parallel with the balance control system. By achieving this and ensuring that the behaviours can be easily added and removed, we can provide a scalable platform that will allow future functionality to be added to the robot.

Programs written today typically follow a structured framework which flows as a series of *if-else* statements. This type of programming requires little preparation and allows the programmers to simply sit down and start typing [21]. The problem with this form of programming is that it can be difficult to expand, especially when dealing with complicated systems such as robots. This is referred to as a rule-based system, where the systems have hard coded rules with an "if X then Y" structure [25].

To understand this we will start with a generic model of structured programming first and then develop this into a more detailed example in relation to this project. The generic, structured programming model is shown in Fig 3.1.

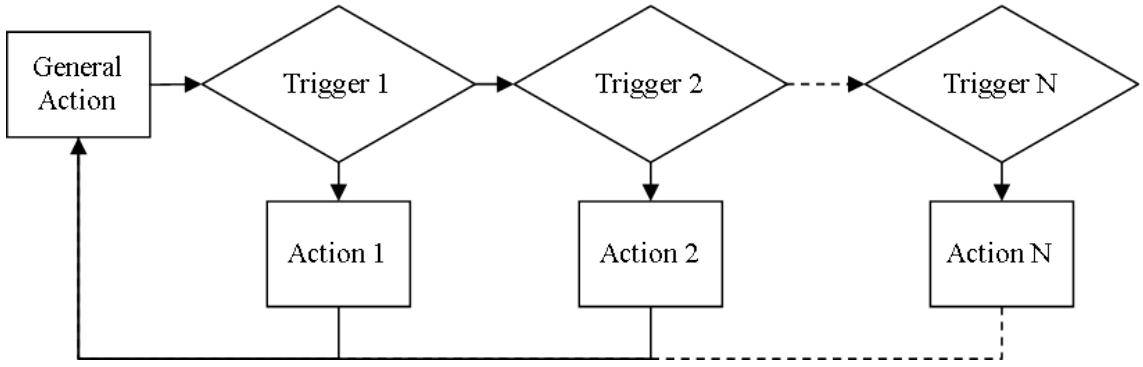


Figure 3.1: Flowchart of a generic, structured programming model

The structured programming works by sequentially checking *if statements* and performing an action if they are currently being triggered. This means that as more and more *if statements* are added the time taken to check if an action should be performed could be significant enough that actions are not initiated when they should be. To make this clearer we have included an example which is specific to this project. Let us start with the three basic behaviours:

1. A drive forward behaviour which keeps the GELway in a constant state of forwards motion.
2. A driving behaviour that allows an operator to control the GELway's movements.
3. An obstacle avoidance behaviour which prevents the GELway from running into obstacles.

Fig 3.2 shows a flowchart of a typical *if-else* programming structure for these behaviours.

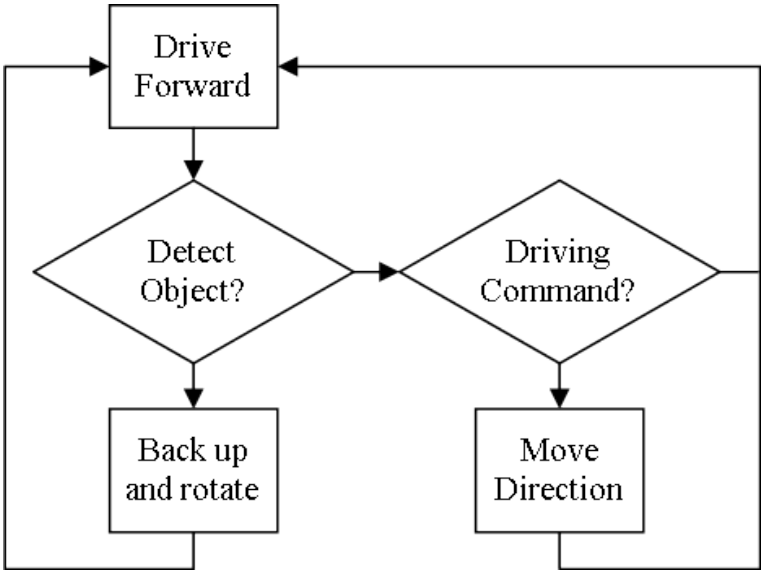


Figure 3.2: Example of structured programming

As mentioned before the major issue with this type of programming is the fact that it executes the code in a sequential manner. For example a basic process could go as follows:

1. Drive Forward.
2. Has the robot detected an object? No.
3. Has the robot been sent a movement command? Yes.
4. Execute the movement command.
5. Drive Forward.

This type of structured programming can cause issues if the robot becomes in range of an obstacle whilst it is executing its movement commands. Since the robot is currently executing the movement command it has no way of reading the sensor data to detect an incoming obstacle and therefore runs the risk of running into the obstacle. This problem only gets worse when more behaviours are added to the system.

In contrast to this, the behavioural control model constantly checks if a behaviour wants to become activated and allows higher level behaviours to suppress lower level ones. It requires

a little more preparation before programming begins but has the advantage of scalability. Each of the robot's behaviours can be encapsulated into individual class files that can be added and removed from the system without the repercussions of affecting other areas of code. A generic model of the behavioural structure is shown in Fig 3.3.

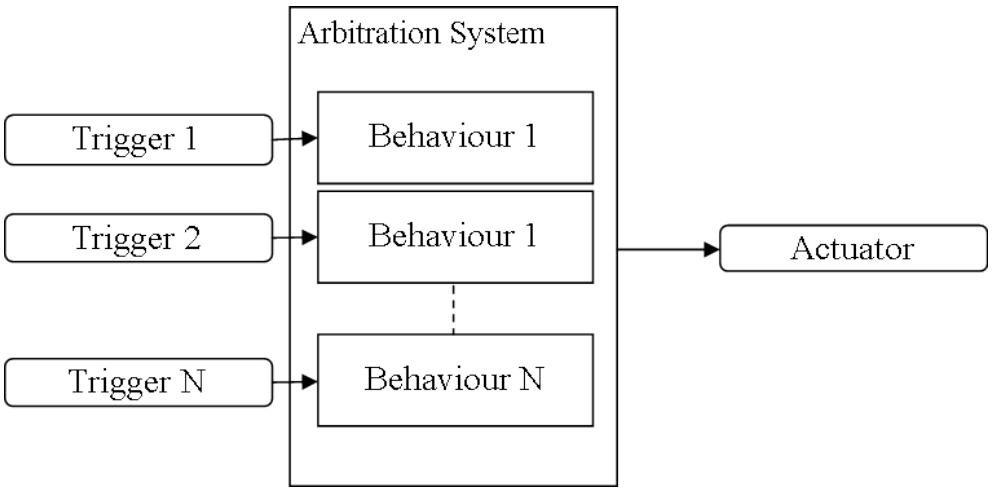


Figure 3.3: Flowchart of a generic behavioural programming model

The behavioural programming works by having an arbitration system that is constantly running to check which behaviours want to become active. Any behaviour that is triggered can send its commands to the actuator. If more than one behaviour is triggered, the higher level behaviour will take precedence and suppress the lower level behaviour's action and execute its own. This improves on the rule-based programming method as the arbitration system constantly checks what behaviours want to become active, regardless of whether any are currently being actioned. Using the same example in Fig 3.2 we will see how the behavioural system has improved on the issues of the structured programming model. Fig 3.4 shows the behavioural system.

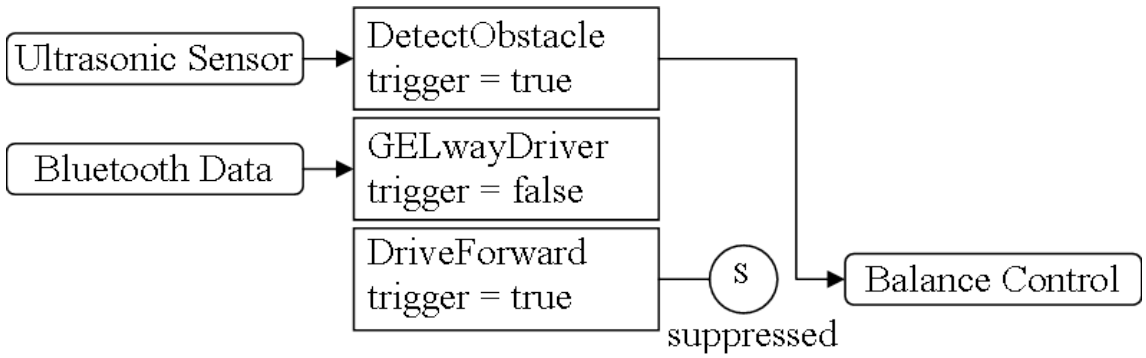


Figure 3.4: Example of behavioural programming

The behavioural system constantly checks which behaviours want to become activated, and as such if the robot encounters an obstacle whilst it is being ordered to move forward, the behavioural system can suppress the movement being executed to allow the robot to reverse and turn away from the object. This has a clear advantage over the typical rule-based system as the behaviour's triggers can be checked almost simultaneously allowing the appropriate behaviour to activated when it is needed. Section 3.2 will now cover how the leJOS package achieves this.

3.2 Introduction to the leJOS Behavioural Package

The leJOS behavioural package was chosen for the GELway's behavioural system since it has the key goal of allowing for easy expansion of a robot's functionality. In this section we will cover how this is achieved with the leJOS NXJ package, starting with the structure of the behaviours and arbitration system that was used.

3.2.1 Structure of the leJOS Behavioural Package

The leJOS Java API has a built in Behaviour package, with an Arbitrator class to regulate which behaviours are activated. There are three methods in the behaviour class:

- boolean takeControl(),
- void action(), and
- void suppress().

Each of these methods will be explained individually below.

Behaviour Method – takeControl()

This method returns a Boolean value indicating if this behaviour should become active. For example if the ultrasonic sensor on the GELway detects an object in its way, this method should return true.

Behaviour Method – action()

The action method for a behaviour is started once the behaviour is triggered. Using the example above, if the GELway detected an obstacle it would reverse and turn away, thus avoiding a collision.

Behaviour Method – suppress()

The suppress() method is used to cease the running code immediately in the action() method. It can also be called to update any data for the robot before the behaviour is suppressed or finished.

Each of these methods are needed whenever a behaviour is created for the GELway. Once a behaviour has been created it is added into the Arbitrator class.

Arbitrator Class

An *Arbitrator* object is created with an array of behaviours as the input. The *Arbitrator* object regulates which behaviours will become active. The priority level of each behaviour is determined by the index array number. The higher the index number the higher the priority. It is possible to run multiple arbitration systems in the NXT robot, with behaviours being grouped together according to the action they perform. For instance the robot built for this project used two of the motor ports, with each of its behaviours altering the power sent to these motors in some way, i.e. offsetting each motors power to cause the robot to rotate. It would be possible to write another set of behaviours which would control a third motor attached to the robot providing more functionality. When running two behavioural systems in parallel the programmer must be careful that they do not try to control the same motors since behaviours can compete to perform their actions as there is no way to determine which behaviour is of higher level.

The *start()* method in the *Arbitrator* class is used to launch the arbitration system. Once started, the *Arbitrator* object determines which behaviours should become active. This is achieved by calling the *takeControl()* method on each *Behaviour* object, starting with the highest array index object. The *Arbitrator* object cycles through each behaviour until it reaches one that wants to take control. When a behaviour that wants to take control is encountered it initiates the *action()* method for one instance only. The subsumption architecture takes effect when there are two behaviours that want to take control, with the higher level behaviour taking precedence. Going back to Fig 3.4 a detailed explanation on

how it works can now be given.

As discussed before the example shown in Fig 3.4 is made of three basic behaviours:

1. *Drive Forward*, which is the lowest level behaviour, which always returns a *true* from its *takeControl()* method.
2. *GELwayDriver*, a behaviour which allows the user to control the movement of the robot and has the Bluetooth data as its trigger.
3. *DetectObstacle*, the highest level behaviour that is used to detect and avoid obstacles which uses the ultrasonic sensor readings as its trigger.

In this example the *Drive Forward* behaviour is always active, that is to say the robot will always move forward unless it detects an obstacle or is given movement commands from an external Bluetooth device. When this happens, the *Drive Forward* is suppressed since it is the lowest of the behaviours and the higher level behaviours will run their *action()* method, such as reversing the robot away from an obstacle.

The next section of this chapter will cover how the leJOS behavioural package has been implemented into the GELway.

3.3 Implementing the Behavioural System

Since one of the goals in this project was to design a scalable software platform for reconfigurable robots, the intention for implementing the behavioural system was to allow for behaviours to be easily added and removed from the system without affecting other functions within the robot, such as the Bluetooth and balancing threads. For this to happen a few things needed to be prepared within the robot's class files. Firstly any parameters that needed to be shared between the behavioural, Bluetooth and balancing threads were required to be added to a synchronised class file. The details of this will be covered in Section 3.3.1. Secondly the behavioural system needed to be initiated from the main

GELway class file but only once the robot had begun balancing. The reasoning behind this can be found in Section 3.3.2. Lastly each behaviour needed to be programmed so that it remained independent from other behaviours which allowed for it to be easily removed without additional programming required to keep other behaviours functioning. The two previous points will now be discussed before details on how the system has been implemented are covered.

3.3.1 Designing a Synchronised Class File

Since the behavioural system thread will be designed to not only run in parallel with the balancing thread, but to also alter its parameters, care needs to be taken with concurrency issues that can arise. When parameters are accessed by more than one thread it is necessary to ensure there is synchronisation to make certain that data is not read when it is in an inconsistent state [21]. The leJOS Java API supports the standard Java synchronisation mechanism and has been used to create the *CtrlParam*, which is effectively a bridge between the behavioural and balancing threads.

The *CtrlParam* Class File

The *CtrlParam* class file was designed to allow the behavioural system to alter parameters that the balance system used within its control system that kept the GELway upright. This allowed the behavioural system to modify how the robot balanced which provided functionality such as moving the robot around. The *CtrlParam* class file is made up of several synchronised Get and Set methods that allow these two threads to access the same parameters. They are as follows:

- *setLeftMotorOffset(double offset)* – this method offsets the power sent to the left motors. This can be used to turn the left wheel faster or slower than the right motor, causing the robot to rotate.
- *setRightMotorOffset(double offset)* – this method offsets the power sent to the right

motors. This can be used to turn the right wheel faster or slower than the left motor, causing the robot to rotate.

- *LeftMotorOffset()* – this method is used to get the current offset in the left motor. It is used in conjunction with the output of the PID controller in the balance thread when sending power to the left motor.

- *RightMotorOffset()* – this method is used to get the current offset in the right motor. It is used in conjunction with the output of the PID controller in the balance thread when sending power to the right motor.

- *setTiltAngle(double angle)* - this method alters the current tilt angle offset for the motor angle used in the balancing thread. It works incrementally since the nature of the PID controller in the balancing thread is to remove any constant disturbance. This method can be used to move the robot forwards or backwards by gradually increasing or decreasing the tilt angle offset.

- *tiltAngle()* – this method is used to get the current tilt angle offset. It is used when calculating the angle of the motor for the balance controller thread.

- *resetTiltAngle()* – since the *setTiltAngle(double angle)* method works incrementally. This method was put in place to allow the tilt offset to be set back to its original value of zero. Without this method the tilt angle offset could be too large when the GELway rebalanced after falling.

- *setDriveState(int state)* – the drive state method is used by the balancing thread to put the GELway into a constant state of motion. It consist of three states, forwards (1), backwards (2) and stationary (0).

- *getDriveState()* – this method gets the current drive state of the robot.

All these methods are synchronised to ensure concurrency between the multiple threads running the system. The *CtrlParam* object is created in the main method of the *GELway* class file and is copied into all other relevant object class files when they are created. The source code for this class has been provided in Appendix A.6.

3.3.2 Considerations on Implementing the Behavioural System

Considering the majority of behaviours in the behavioural system modify the balancing parameters, issues can arise if it is not implemented in the correct matter. Running the behavioural system before the balancing thread has completed its initial set up phase could cause behaviours to trigger prematurely. This could modify the balance parameters, such as the tilt angle offset, causing too large of an initial disturbance in the PID controller resulting in the robot becoming unstable and crashing. The solution to this problem was to ensure that the behaviour system thread was not started until the initial set up of this system was completed. This included establishing the Bluetooth connection, calibrating the gyro sensor and placing the robot in the upright position. By doing this the balancing parameters will start at their expected values.

3.3.3 Programming the Behavioural System

Programming the behavioural system is done in two stages. First, each behaviours needed to be designed separately, each having their own unique triggering mechanism. Secondly the behaviours needed to be prioritised and added to the arbitration system. The behaviours which were programmed for the GELway were previously shown in Fig 3.4, that is a behaviour that moves the robot constantly forwards, a behaviour that allows for the robots movements to be controlled and a behaviour that avoids obstacles. Following a more in-depth description of these behaviours the difficulties encountered when designing the behavioural system will be discussed.

Designing the Behaviours

As mentioned previously the behaviours are made up of three methods, a *takeControl()*, an *action()* and a *suppress()* method. The three behaviours will be covered in order of priority, starting from the lowest priority behaviour.

1. *DriveForward()* Behaviour – The *DriveForward()* behaviour was only designed to be a testing behaviour and as such is very simple. The behaviour required no trigger for its *takeControl()* since it is always returns a true value. The *action()* method sets the drive state of the GELway to forwards with the *surpress()* setting the drive state to the stationary position.

2. *GELwayDriver()* Behaviour – The *GELwayDriver()* behaviour is a little more involved than *DriveForward()*. The behaviour required access to the *BluetoothReader* methods to check that a new command has been sent to the GELway. When new commands are sent over Bluetooth, the *takeControl()* returns a true value and the action of the robot is determined by integer value sent. The *action()* method uses a switch/case statement that determines what movement the GELway should perform. This behaviour required nothing in its *surpress()* method since suppressing the behaviour causes no actions to be performed. This type of behaviour is considered to be a ballistic behaviour, even though it is always triggered, as it performs a predictable action depending on the input of the Bluetooth system.

3. *DetectObstacle()* Behaviour – The *DetectObstacle()* uses the distances measured from LEGO's ultrasonic sensor as its triggering mechanism in the *takeControl()* method. If the ultrasonic sensor measures a distance reading less than $20cm$ the *takeControl()* method returns a true and the *action()* method is initiated. Once activated this behaviour first stops any movements currently running, reverses the GELway and turns it 180 degrees. Since this behaviour has the highest priority it cannot be surpressed and therefore requires nothing in its *surpress()* method. This type of behaviour is ballistic as it is once triggered and follows a predictable path each time.

Issues Encountered when Designing the Behavioural System

The integration of the behavioural system into the GELway was not without its difficulties. The issues encountered were as follows:

- That the original firmware running on the NXT, leJOS 0.7, had errors in its Arbitrator class files which meant that it did not cycle through the behaviours correctly. This meant that only the highest level behaviour would remain active. During the course of the year however the developers of the leJOS firmware released an update with a fully functional behavioural package.
- Originally the *BluetoothReader* thread was a part of the *GELwayDriver()* behaviour. The problem with this was that waiting for an integer to be sent pauses the behaviour's action, effectively freezing the arbitration thread until an integer is sent. The solution to this was to create a separate thread specifically for waiting for sent Bluetooth commands. By doing this we also opened the possibility to control other functions with the robot rather than just movement commands.
- Similar to the above issue we have to ensure that each action has a set start and end point. Having an infinite loop inside a behaviour's action means that the program is unable to exit the behaviour which causes the behavioural system to fail.
- Setting the correct priority for the behaviours is also very important. For instance having a behaviour that triggers regularly high on the priority list means that the lower behaviours are constantly being suppressed which could cause problems. For example if driving the robot around was higher in priority than avoiding an obstacle, its obstacle avoidance could be suppressed by constantly driving it forwards causing the robot to run into the obstacle. We need to ensure that the correct priority is assigned to each behaviour.
- Another known issue with the behavioural system, although it has not been encountered with this project but could be a future problem, is that there could be a significant delay in checking a behaviour's trigger. This would be more apparent if there were multiple behaviours. For example, consider a touch sensor being used as a trigger to

detect if it has connected with an object. If the touch sensor releases from the object before the arbitrator can check if there has been a connection the behaviour might not be activated. The solution to this is to use sensor listeners, and have them set a flag when a certain event has occurred.

With these issues taken into account we can ensure we have fully functional behaviours that can now be added to the arbitration thread.

Adding Behaviours to the Arbitration Thread

The next stage in implementing the behavioural system is to add the behaviours to the arbitration thread. First the behaviour objects are created, noting that each use the *CtrlParam ctrl* object, which contains the synchronised parameters used between both the behavioural and balancing systems.

```
Behavior b1 = new DriveForward(ctrl);
Behavior b2 = new GELwayDriver (ctrl, br);
Behavior b3 = new DetectObstacle (ctrl);
Behavior [] bArray = { b1, b2, b3 };
Arbitrator arby = new Arbitrator(bArray);
arby.start();
```

Once the behaviour objects are created they are added to the arbitrator in an object array list, noting that the higher a behaviour is in the array the higher its priority. In this example the *b3* behaviour object has the highest priority. Once added, the arbitrator thread is started and begins checking the behaviour's *takeControl()* methods and initiating their actions.

3.4 Testing the Behavioural System

Whilst it is possible to visually note if the behavioural system is working by triggering each of the behaviours, it can be quite difficult to see how the suppression mechanism is functioning, especially if there are multiple behaviours running on the robot. Thus, in order to get a better understanding of the behavioural system in action, a test was designed that would check which behaviours have been triggered against which action method had been called by the arbitrator. The results are shown in Fig 3.5.

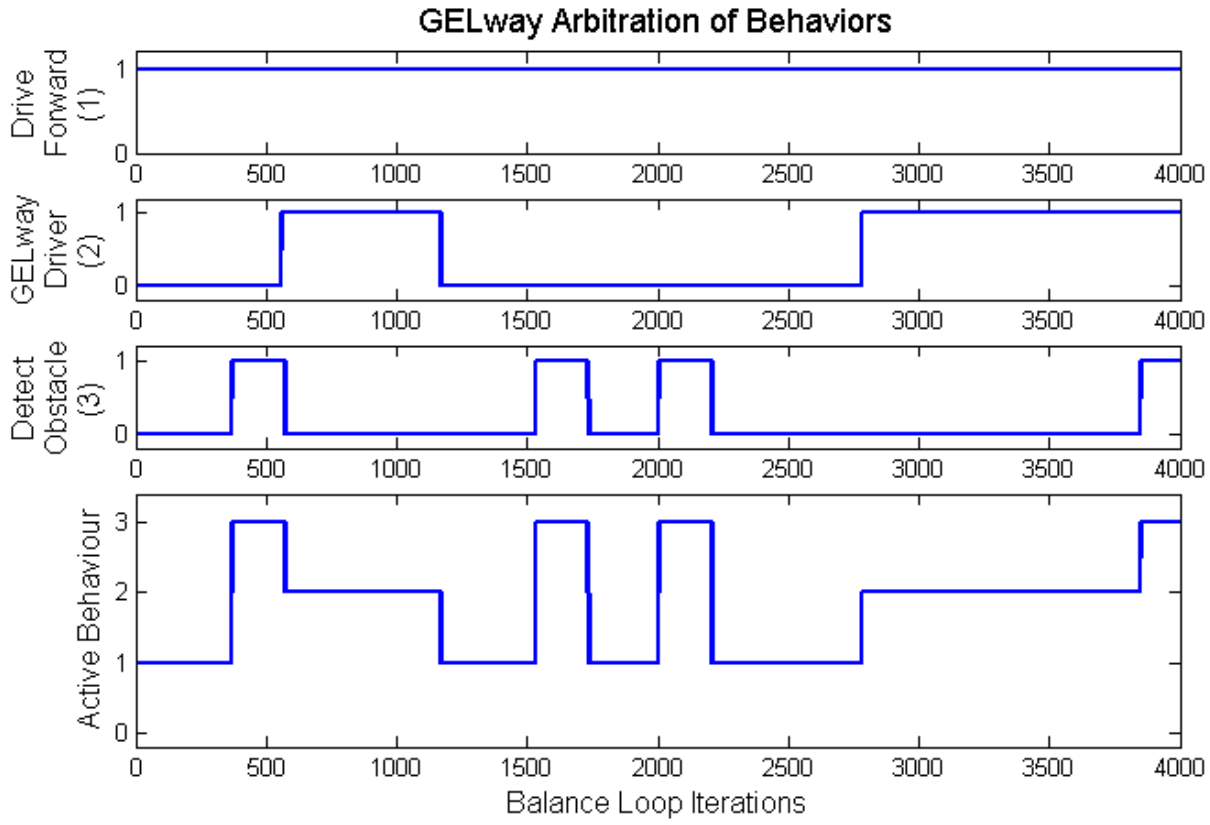


Figure 3.5: Testing the arbitrator in the behavioural system

The first three plots in Fig 3.5 represent each of the behaviour's *takeControl()* methods respectively. A '0' represents when a behaviour has not been triggered and a '1' represents a triggered behaviour. Note that the *DriveForward* behaviour is always triggered. The bottom plot shows which behaviour is currently active, as in executing its *action()* method. In this plot a '1' shows *DriveForward* is active, a '2' shows *GELwayDriver* is active and a '3' shows

DetectObstacle is active. Fig 3.5 shows us that the behavioural system is functioning as it should be since the higher level behaviours are suppressing the lower level behaviours when triggered.

3.5 Chapter Conclusion

This chapter provided two major milestones in the path to designing a scalable platform for reconfigurable robots. The first milestone was the behavioural system which has given the robot the capability to make its own decisions based on its environmental inputs and the second functionality was the ability to run multiple systems in parallel, namely the balancing system, the communication system and the behavioural system. The methodology described in this chapter brings the systems presented in Chapters 2, 4 and 5 together. This was done by modifying shared balancing parameters from the balancing system to allow it to implement behaviours which were capable of moving the robot in any direction whilst sustaining it in the upright position. It also used a designed behaviour, triggered by Bluetooth inputs sent by the user from the communication system, covered in Chapter 4, to allow the operator to control the movements of the robot and drive it around via a mobile phone or laptop. Lastly, it provided a means to implement the distance controller, covered in Chapter 5, to allow multiple robots to follow each other at a set distance in a string formation. With the concept of the behavioural system proved the next task for future students would be to design behaviours that provide multiple robots the ability to work together as a group and solve problems jointly. This can be achieved by utilising the communication systems developed in the next chapter which provides the ability to share information with other devices and robots, which means behaviours could be triggered by joint-sensory information.

Chapter 4

Communication Between Heterogeneous Robots using Bluetooth

This chapter will focus on the communication between multiple robots using Bluetooth connections. These communication links need to be developed so that we are able to control multiple robots simultaneously as well as have the robots communicate with each other. This will allow us to explore how robots interact with each other and the limitations that can apply. The key stages of this communication system are connecting the GELway to an external Bluetooth device, connecting the GELway to another balancing robot and running these two communication systems in parallel.

This project has used a master/slave model for its communications system as it allows an operator to control a group of robots with only one communications link required to the master robot. Section 4.1 introduces master/slave communication in multiple robotic systems, covering its advantages and describing its function in robot platoons and modular robotic system. The master/slave communications system required two types of Bluetooth connections to be made: The first being one with an external Bluetooth device to allow an operator to have remote-control over the master robot, and the second connection between two robots to allow the master to control the slave robot. The first connection between the GELway and an external Bluetooth device is covered in section 4.2 and details how the GELway was connected to a mobile phone and laptop PC. Reprogramming an application for the mobile phone so that it could connect with the GELway was the most difficult part of this chapter as the original Bluetooth mobile application used was poorly documented. The modified mobile application in this project has been fully documented so as to reduce

the learning curve for future work on external Bluetooth connections. The remote operator connection was used to trigger the driving behaviour, mentioned in Chapter 3, to allow an operator to control the GELway's movements.

Section 4.3 covers the connection of two NXT systems together over a two-way Bluetooth communication link allowing the two robots to send and receive information. This connection process is well documented and was successfully implemented without any difficulties. The final part of this chapter looks at the communication of multiple robots, specifically the sharing of sensory information between two robots, the relaying of commands from the master to the slave robot and the difficulties of multiple robot communication systems. This is covered in section 4.4.

The sharing of sensor information between the two robots was one of the key functionalities that this project set to achieve and investigate. By sharing information the robots are able to get a better overall picture of their environment since they can effectively double the amount of sensor inputs into their system. Each robot is able to use different sensors, giving themselves their own specific role within a group of robots, such as a compass sensor being attached to a group's navigator.

4.1 Master/Slave Communication in Multiple Robotic Systems

The master/slave communication model in terms of robotic systems is where one system, the master, has unidirectional control over one or more systems, the slaves. This type of configuration allows a master robot to delegate tasks to the slave robot. Rather than completing the tasks as individuals, the master is able to determine an optimised solution to a given problem, assign specific tasks to the slaves and provide the capability of group problem solving [32]. The master/slave communication model is applicable to many multi-robot systems such as robot platoons and modular reconfigurable robots.

Robot platoons can benefit from a master/slave communication model as the designated master robot can take in the sensory inputs of the group providing a better overall view the group's operational environment. This can allow the master robot to make more informed decisions, optimising the group's performance, rather than individuals. In this project we have provided the robots with the capability to share and send sensory information between them. By using the behavioural system, together with the master/slave communication, the designated master robot can initiate behaviours which are based off joint sensory information, allowing the robots to adapt to environmental changes as a group, rather than individuals.

In modular robotic systems, master/slave communication can be used to reconfigure the complete system to adapt to its current operational need. The individual modules can relay their sensory information to the master to allow it to optimise the robots current configuration. Master/slave communication in modular systems also allows the master to delegate movement commands to individual slave models in order to generate movement of the system as whole. For instance an operator controlled modular robot would only have commands sent to the master, where the master can further delegate these tasks to the slaves. This simplifies the communication system since only one active communication link is required to control the movements of the modular robot, rather than connecting to each module individually.

The master/slave model was selected for the communications system for this project as provided us with the ability to control multiple robots whilst only being connected to the master robot. Using the master/slave model we can control the master robot whilst the slave follows behind autonomously. This means we are only concerned with commanding the master robot, where the master can handle the control over the slaves. This would become increasingly important when numerous robots are involved, since controlling each individual robot would be impractical.

4.1.1 How the Master/Slave Communication Works

The master/slave model for this project requires two types of connections:

1. A connection from the master robot to a remote-control device.
2. A connection from the master robot to the slave robot.

The master/slave communication system can function on any connection type, whether it is physical or wireless. This project omitted physical connections as a choice since it required our robots to be mobile and not restricted by wire length. Since the NXT system came with Bluetooth functionality it was chosen as the communication type. Fig 4.1 depicts a master/slave connection with an external remote-control operator connection.

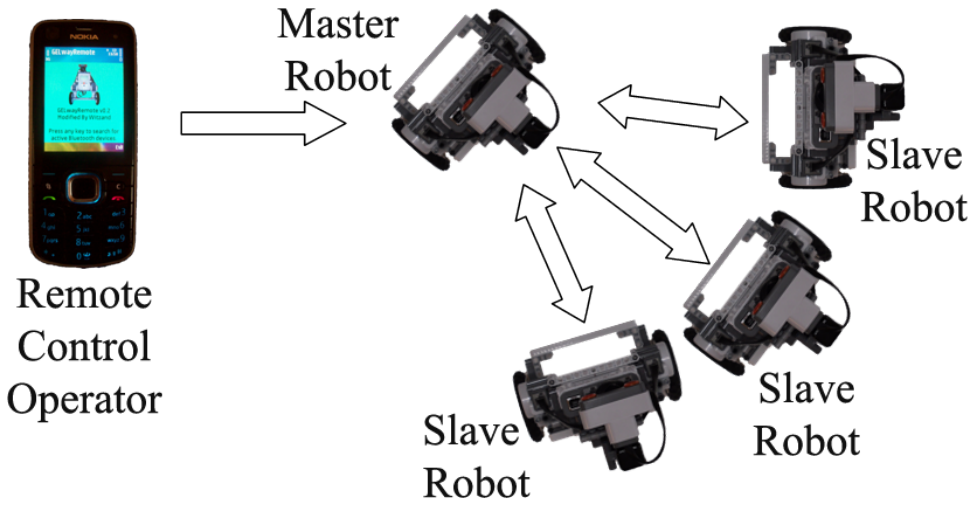


Figure 4.1: Master/slave connection with external remote-control operator.

As we see in Fig 4.1 there is only one connection made to the master robot but multiple connections made from the master to the slave. The master/slave connection allows an operator to control a group of robots by only communicating with the master, enabling it to take an operator command and relay it to the slaves. The connection between the master and slave also allows the master to receive sensor data from all the slave robots, allowing the robots to have an overall view of the group environment rather than just their own sensor

inputs. These two connection types will now be covered in more detail to describe how they were achieved using the LEGO Mindstorms Bluetooth connection. Section 4.2 details the remote-control to master connection and section 4.3 covers the master and slave connection.

4.2 Establishing the Remote-Control Connection

Establishing a Bluetooth connection with the NXT allows us to control functions of the GELway without the need of the USB cable, or having to manually press the buttons on the robot.

The Bluetooth feature of the NXT gives programmers more freedom to control their robots from a distance, an important feature in driving and communicating with multiple robots. The two connection types that will be made with the NXT to allow us to trigger actions within the robot are a mobile phone to NXT connection, and a laptop PC to NXT connection.

Before any Bluetooth communication links can be established the GELway first had to be programmed to wait for a connection to be made, set up the necessary connection stream and start a Bluetooth listener thread.

4.2.1 Bluetooth Receiver Thread

In order to process commands sent to the GELway there needed to be a method of handling the Bluetooth connection and storing any information sent. Since the GELway is constantly running a thread in order to keep itself balanced it was decided to run another thread in parallel which would handle the Bluetooth communication. The thread itself would have a lower priority compared to the balancing thread, since keeping the robot upright is the most important priority, but would still help reduce the loss of any commands sent over Bluetooth since it is constantly waiting for new information.

Connecting to the NXT

The NXT uses the *BTConnection* classes provided in the leJOS firmware package to establish Bluetooth connections. This makes the connection process far simpler provided the right methods are used when establishing a connection. The first step in programming the connection is to make the robot wait for an active Bluetooth connection and establish the connection type. This is done with the following commands:

```
LCD.drawString("Waiting...", 2, 1);
BTConnection conn = Bluetooth.waitForConnection();
// Used when a PC/NXT connection is made
conn.setIOMode(0);
// Used when a phone connection is made
//conn.setIOMode(NXTConnection.RAW);
LCD.drawString("Connected", 6, 1);
```

The *Bluetooth.waitForConnection()* effectively pauses the program and as such had to be called before any of the threads had been initiated to prevent the robot from moving around before the Bluetooth connection had been established.

Initiating the Receiver Thread

Once the Bluetooth connection had been established the *BTConnection* object is sent into the *BluetoothReader*. This thread opens the data input and output streams in its constructor and once initiated runs an infinite loop which constantly waits for sent integers. A delay had to be added after reading in an integer since the leJOS Bluetooth connection firmware is only capable of raw communications speeds of approximately $40ms$ [33].

The thread consists of three methods:

1. *getDir()* - this method is used to get a sent integer command.
2. *getNewDir()* - this method checks that a new command has been sent. Without this it would be difficult to determine if the same command had been sent twice.

3. *sendCommand(int command)* - which is used to send an integer command over the Bluetooth channel. This is only used during master/slave communication.

As mentioned before, this thread will run in parallel at a lower priority to the balancing thread and is initiated once a Bluetooth connection has been established.

4.2.2 Mobile Bluetooth Connection

The connection with a mobile phone was the first Bluetooth connection established with the NXT. It has the advantages of being a highly portable remote-control but lacks in being able to process received data from the NXT, such as plotting received sensor information. Another disadvantage is that the initial connection is slower than the laptop as the phone has to search for all available Bluetooth devices rather than just directly connecting to the NXT's Bluetooth address.

One of the main differences that separate the mobile phone from the Laptop PC and NXT is the input and output mode for the Bluetooth connection. Since both the Laptop PC and NXT use the same Bluetooth class files they can be connected in a stream mode, where no header files are sent between the devices. However, the mobile phone has its own classes, which send two header messages with each transfer, and as such, the NXT has to change its input/output mode to a RAW connection. This is done with the following line:

```
conn.setIOMode(NXTConnection.RAW);
```

Note that *NXTConnection.RAW* simply returns an integer value of 2 i.e. there are two header messages sent when connecting with Bluetooth devices that do not use the leJOS classes.

Mobile Hardware Specifications

The mobile phone specifications are provided in Table 4.1 so that this communication link can be recreated by other people wishing to communicate with their NXT robot [34].

Mobile Phone	Nokia 6220 Classic
Operating System	Symbian Version 9.3
User Interface	S60 3rd Edition Feature Pack 2
Connectivity	Bluetooth 2.0 Interface

Table 4.1: Bluetooth mobile specifications

The software written for NXT Bluetooth communication was designed for phones which run on Symbian software. Tests have been conducted using only the Nokia 6220 Classic, however other phones which run on a Symbian operating system should be able to install and run the communication program.

Mobile Phone Software

The software used to communicate the Nokia 6220 to the GELway is based off Pedro Miguel's *NXTSymbian* [35]. The program establishes a Bluetooth connection with an NXT device and then once connected, sends different byte codes to the NXT depending on the mobile key pressed. Receiving the byte codes on the NXT was difficult however, as each byte code was a different length, and sometimes only half the information was received. Since Miguel had provided the source code for his program, it was decided to reprogram and edit the *NXTSymbian* mobile application.

The first step to reprogramming Miguel's mobile application was to find an IDE capable of creating an install file for the mobile phone. It was decided to go with Mobile Processing, an open source programming environment that can be used to write applications for Symbian devices [36].

Mobile Processing IDE is an easy to use compiler which uses Java as its programming language. This was a major factor in selecting it as an IDE since the programming language used on GELway was also Java, thus reducing the required learning time needed to use it.

The *NXTSymbian* source code was loaded into the Mobile Processing IDE in order to gain a better understanding into how it functions, and how best to alter the code. Unfortunately the original programmer did not provide any documentation on how the program functions, or any commenting within the source code. Since this mobile application is an important part of this project, each method in the program was properly documented, which will hopefully reduce the learning curve needed for future users of the program.

The major change needed for the *NXTSymbian* application was to make it send integers to the NXT rather than byte codes. This was achieved by altering the method:

```
void nxtkey(int akeyCode, int akey);
```

The method implemented a switch to determine what key was pressed in the phone, and then sent the corresponding byte code for that key. The method was modified to instead send the corresponding integer to the key pressed on the phone, which could then be received by the GELway. The Bluetooth receiver program on the NXT was then changed to receive integers instead of bytes, and a successful communication was able to be established.

The program was renamed to *GELwayRemote*, and images were added to the program to make it unique to the connection of the GELway. The source code for this program can be found in Appendix A.16 - GELwayRemote Source Code.

GELwayRemote Mobile Phone Commands

Each button on the mobile phone keypad is used to send an integer command to the GELway. Fig 4.2 shows each function the mobile keys perform when pressed. Note that the help and exit buttons are *GELwayRemote* specific commands and are not transmitted over the Bluetooth connection.

Figure 4.2: *GELwayRemote* commands for mobile phone.

The keypad buttons 2, 4, 6 and 8 are used to control the direction of the GELway robot whilst the 1 key resets the robot balance mechanism. The central joystick on the phone can also be used to drive the robot around. The unused keypad buttons still send their respective integer number however they have not been programmed on the GELway. This allows for future functionality to be added.

4.2.3 Laptop Bluetooth Connection

Connecting the laptop PC and the NXT via Bluetooth was much simpler than the mobile phone connection as they both use the same Java class files. The other advantage is that leJOS provides a whole tutorial chapter dedicated to Bluetooth connections between the computer and the NXT on their website [21]. The major advantage in using the laptop PC is that it is able to process data received from the NXT and plot the results, which allows for further analysis. However the laptop lacks in portability, and is more or less fixed to the range of the Bluetooth connection. The laptop Bluetooth connection will be used to control the master GELway.

As mentioned previously the Bluetooth connection, when both devices use the leJOS classes, can be done in stream mode. This means that no header files are sent between the devices and the input output connection is set up with the command:

```
conn.setIOMode(0);
```

Note that the zero implies that no header files are sent, which differs from the mobile connection which uses two header files.

Laptop Hardware Specifications

The laptop specifications are provided so that this communication link can be recreated by other people wishing to communicate with their NXT robot [37].

Laptop	Dell XPS M1530
Operating System	Windows Vista Home Premium
Java Version	Java(TM) 6 Update 13
Bluetooth Type	Dell Truemobile 355 Bluetooth
Bluetooth Version	6.1.6002.18005
Connectivity	Bluetooth 2.0 Interface

Table 4.2: Laptop specifications

Note that most computers that are able to establish Bluetooth connections with other devices should be able to connect with the NXT.

Laptop GUI Software

An interface was designed on the laptop to allow for commands to be sent to the NXT. The GUI is a simple design that uses a key listener to send an integer to the NXT depending on the key which is pressed. The GUI is shown in Fig 4.3 and is comprised of four simple sections.

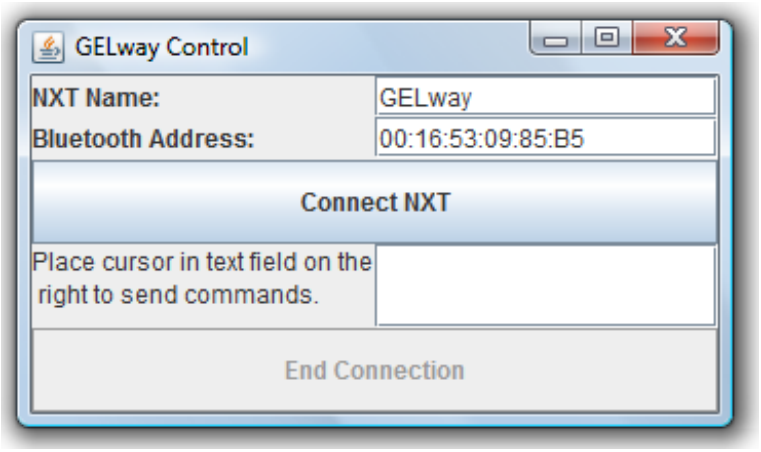


Figure 4.3: Laptop GUI controller.

The first section allows the user to enter in the name of the NXT device and its Bluetooth address with the second section providing a connection button to the NXT. The third section has a text field which has a keyboard listener that sends an integer to the NXT depending on the key that is pressed. The integers sent by the laptop GUI were designed to correspond with the *GELwayRemote* mobile application so that no extra programming was required on the NXT. The fourth section closes the Bluetooth connection with the NXT. The source code for this GUI can be found in Appendix A.17 PCController Source Code.

Laptop GUI Commands

The following keys are used to control GELway after a Bluetooth connection is made.

Keyboard Key	Action
Left arrow	Turn GELway left
Right arrow	Turn GELway right
Up arrow	Move GELway forwards
Down arrow	Move GELway backwards
Enter	Reset GELway balance

Table 4.3: Laptop GUI keyboard commands

4.3 Establishing the Master/Slave Connection

The final Bluetooth connection required for master/slave communication is the NXT to NXT connection. Having a communication link between the robots is a vital part of creating a scalable platform for reconfigurable robots. The link will allow the master robot to relay messages to the slave robot as well as allow the robots to share sensors.

Since both NXTs will be using the same Bluetooth class files, the connection mode is the same as the previously mentioned laptop to NXT connection. However, the process of connecting the two devices is slightly different. First, the master robot needs to have already been paired with the slave robot and have its name and Bluetooth address stored. The master can then connect to the slave robot directly by using the commands:

```
String name = "GELwayJR";
LCD.drawString("Connecting...", 0, 0);
LCD.refresh();
RemoteDevice btrd = Bluetooth.getKnownDevice(name);
BTConnection btc = Bluetooth.connect(btrd);
LCD.drawString("Connected␣Slave", 0, 0);
```

Note that the Bluetooth receiving methods for the slave are the same as previously mentioned in Section 4.2.1.

Since the master requires two active Bluetooth connections, one to listen to commands from the mobile and laptop, and one to communicate with the slave robot, another *BluetoothReader* thread is initiated.

4.4 Communication of Multiple Robots

In sections 4.2 and 4.3 we presented the two important communication links we require to control multiple robots simultaneously. In this section we discuss the process of connecting the user and master/slave connections to show the steps involved in connecting two active Bluetooth connections.

4.4.1 Process of Connecting the Bluetooth Connections

In order to connect the two active Bluetooth connections required for the GELway, the first being the remote-control connection with the user and the second the master/slave connection, the following process shown in Fig 4.4 must be followed.

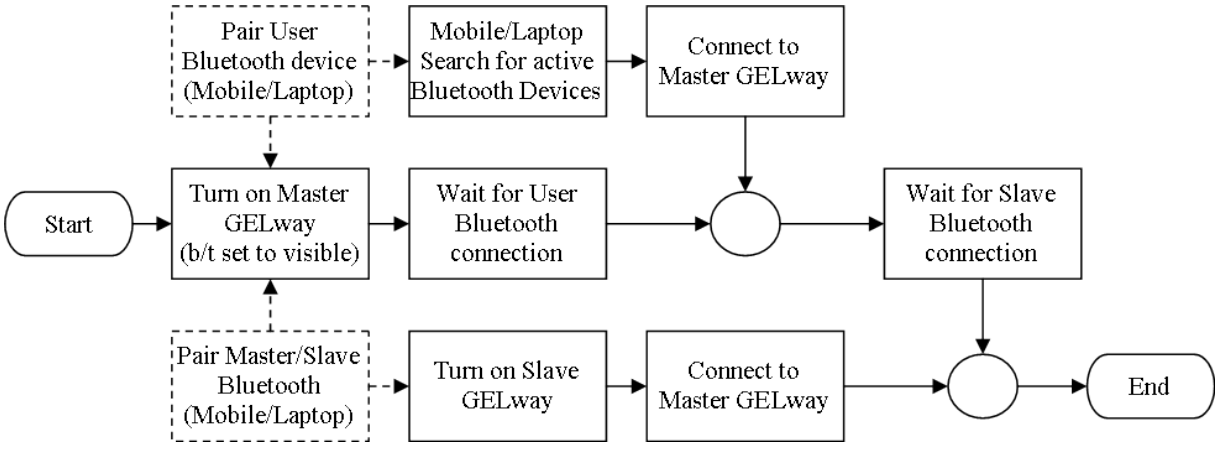


Figure 4.4: Establishing the GELway Bluetooth connections.

The first thing to note about this process is the steps shown in the dashed lines only have to be done the first time the connection is attempted. After the first time the Bluetooth

devices can be paired and the addresses and names of the connecting device are stored. The first stage is to turn on the master GELway, making sure the Bluetooth connection on the robot is visible. At this stage the mobile/laptop and slave robot can be turned on, but are not required to be. The master GELway is then put into waiting mode, which pauses the program and waits for a Bluetooth connection. The mobile/laptop can now search for active Bluetooth devices. Note that the laptop is capable of directly connecting to the Bluetooth device without searching since the GUI was built with this functionality. Once the master GELway connects to the user remote-control it goes back into wait mode for the slave connection. The slave GELway is now able to connect to the master and all the Bluetooth devices are now connected together. The Bluetooth connections are handled by separate threads in the master GELway.

4.5 Testing the Communications System

To test the communications system in the robot we devised two tests that will allow us first to test the master and slave to communication and the second to test the operator's ability to control both robots. These will now be covered in detail, describing why the tests are important first and following that a description on how the robots performed.

4.5.1 Sharing Sensory Information

With a two-way communication link established between the master and slave robot, they are now able to send sensory information to each other. Sharing information between the robots is one of the key functionalities that this project set out to achieve and investigate. This functionality provides the robots with the ability to get a better overall picture of their environment since the amount of available sensor ports effectively doubles. By attaching different sensors to a robot it can take on its own specific role within a group of robots, for example a robot with an attached compass sensor could become the navigator of the group. The sharing of sensor information between the robots is also a very practical when using LEGO Mindstorms technology since the NXT is limited to only four sensor inputs, with two

sensor ports already used up by the gyro and EOPD sensor.

The sharing of sensor information was tested by using an ultrasonic sensor on the slave robot to control the obstacle avoidance behaviour in the master robot. This test checks the ability of the master to recieve information from the slave. The results of the test showed that the master was able to successfully trigger its obstacle avoidance behaviour by using the ultrasonic sensor attached to the slave. Whilst this test is basic, it shows that the master is able to use sensor information from the slave to make decisions. This can be further developed by adding more slaves to the systems and testing the master's ability to handle multiple slave sensor input.

4.5.2 Controlling the Slave Robot

The Bluetooth communications have been designed in such a way that we are able to send commands only to the master robot. The idea behind doing this is so that you only have to communicate with one robot rather than trying to control multiple robots at once. Therefore the slave was designed to be autonomous but would listen for commands from the master robot. Technically we can still control the slave robot indirectly by relaying information from the master to the slave robot. This is a typical scenario arising in robot platoons, such as strings of autonomous vehicles. In string formations, the leading vehicle passes the information to the vehicle right behind it, then from that vehicle the information is passed on to the next one [38]. Chapter 5 explores the synchronisation of two robots in more detail, having one robot follow the other.

To test the ability of the master to send information to the slave a relay mechanism was programmed into the driving behaviour of the master which would take a movement command sent by an operator and relay this command to the slave, whilst also performing the movement action as well. This test allowed us to see how information was sent between the robots and also the delays that may occur between the two systems. The outcome of the test showed the robots were both able to perform the movement commands sent by the operator. Over time, however, the robots ended up in different orientations and positions, partly due to the different motor characteristics, but mostly due to the slight delay in the

slave receiving the relay command from the master. A possible solution to this problem could be to have the slave send back to the master a confirmation command informing it that he had received the command and is ready to execute the action.

4.6 Chapter Conclusion

This chapter has covered a detailed explanation of adding the communicative functionality to the GELway to allow it to be controlled by an operator using an external Bluetooth device and to send and receive information with another robot. The communication system has been successfully implemented into the GELway by running two threads in parallel with the balancing thread, one thread that has a sole purpose to listen to commands sent by the user and another thread which has a purpose to send and receive information with another robot. These communicative systems are an important step towards creating a scalable platform for reconfigurable robots because these systems place a large focus on the communication with other robots in their vicinity. This also paves the way for joint decision making with multiple robots since they are able to use joint sensory information to get a better overall picture of their environment. The communication system provides one of the key requirements to allowing robots to work together as groups rather than as individuals. The other important requirement is the need to synchronise the movements of multiple robots to allow them to move as a formed body. This leads to the next chapter on synchronous movement of heterogeneous robots.

Chapter 5

Synchronous Movement of Heterogeneous Robots

This chapter addresses the problem of synchronising the movement of two balancing robots. The overall goal to be achieved here is to have the two robots, master and slave, keep at a set distance from each whilst they move around autonomously. Synchronisation of multiple robots is important to this project since modular reconfigurable robots utilise the synchronised movement of all modules to generate movement in the system as a whole. It will also allow the robots to work together more effectively as a team rather than individuals.

There are many facets of robot synchronisation, such as moving the robots in swarms or platoons whilst keeping a desired formation. In this project we chose to work with a string of robots that follow each other. As a step towards creating a string of robots we chose the particular problem of master/slave synchronisation. The synchronisation of the master and slave robot was one of the more difficult tasks taken on in the project and involves getting the master and the slave robot to follow each other in a straight line, and move around autonomously, without operator intervention. This is an example of a basic coordinated behaviour, chosen for the simplicity of its objective, but is known to be a challenging task. The distance controller was included in the behavioural architecture covered in Chapter 3 which allowed it to be suppressed when we desired the follower robot to autonomous and not follow a set path.

The choice to have multiple robots in a string formation also had a practical application for the modular reconfigurable robotic design as it would allow multiple robots to close in on each other to connect up into longer modules. Since two-wheeled, balancing robots are constantly moving, the control system used to keep the robots at a set distance would allow

them to gradually connect together reducing the chance of undesired collisions. The distance control system will increase the portability of this project into the modular reconfigurable world.

The first stage of the master/slave synchronisation involved designing of the control system which would maintain the following GELway at a set distance from the leading robot. This is covered in section 5.1 and describes how the equations of motion were first derived for the controller and then built on Simulink. The controller used proportional gain to prevent overshoot and keep the following robot at a set distance. After the distance controller was built it was then implemented into the robot as part of the behavioural system and is covered in section 5.2 of this thesis. Using the distance controller as a lower level behaviour in the robot allowed it to be suppressed by higher level behaviours and give the following robot the freedom to move around when required. The major issues with the robot synchronisation were the fact that the following robot accelerated too fast at large distances, causing it to become unstable, and also the robot's tendency to rotate over time. To limit the following robot's acceleration its speed was capped so that it would move at a constant maximum relative speed at large distances and the rotation was accounted for by using the compass sensor to keep the robot orientated in the same direction.

5.1 Designing the Robot's Synchronous Movement

In this section the problem associated with synchronous movement of the master and slave robot will be discussed. The synchronisation of multiple robots is important for reconfigurable systems as it allows the complete system to control and manoeuvre its modules to enable it to form new shapes and a range of different movements. This allows reconfigurable robots to adapt to changes not only in their physical environment but also in their physical structure. Allowing the robots to move together synchronously allows us to treat them as a group rather than individuals, making joint problem solving tasks far more effective.

In a simplistic world the problem would be as simple as measuring the distance to the leading

robot and changing the motor speed of the following robot. Fig 5.1 shows a flowchart of a simplified follow the leader type setup.

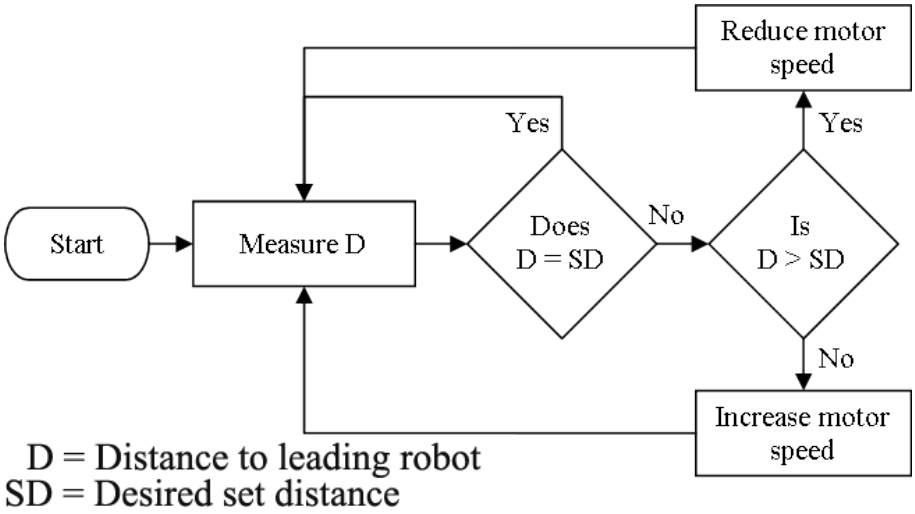


Figure 5.1: Simplified flow diagram for follower robot in synchronous movement

In Fig 5.1 the D represents the distance from the leading robot and SD represents the desired distance between the robots. Note that this setup assumes that power is being sent to the motors at all times to the following robot. Fig 5.1 shows a basic feedback loop where the follower robot measures the distance to the leading robot, works out if it is either at the desired distance, too close or too far away and adjusts its speed accordingly. For instance if the distance was greater than the desired distance it would increase power to the motors. This set up is fine for most robotic systems since their systems are usually stable. But for balancing robots the problem becomes a lot more complicated as the distance between the two balancing robots is constantly changing due to the fact they are always moving to remain upright. This leads us to conclude that motion of the follower robot and the feedback mechanism for correcting the distance must be described in terms of differential equations or transfer functions, and the task of controlling the distance requires a feedback controller.

5.1.1 Problems with Synchronous Movement of Balancing Robots

There are several additional difficulties associated with trying to have two balancing robots follow each other in a straight line that need to be taken into account when trying to implement a follow the leader type movement synchronisation. They are as follows:

1. Simply increasing or decreasing the power sent to the motors to move the follower robot to set distance will not work using balancing robots since the nature of their control system is to remove any constant disturbance. This is because the system proposed in Fig 5.1 is just a very general conceptual diagram. Within a feedback loop, the control signal will not be constant as the position error will vary with time.

2. Applying multiple positive and negative disturbances to the balancing system can cause it to become unstable. To better understand this think, of when the robots are at the desired distance from each other, if it moves back slightly it will be made to move forward, it if overshoots a little it is told to move back, then forwards again. The PID controller will struggle to remove these disturbances causing the balancing robot to oscillate and eventually fall over.

3. During the numerous tests conducted on the robots it was noted that over time they had a tendency to slowly rotate. This is due to the slight differences in the characteristics of the motors controlling the left and right wheels. This is a problem since we need the robots to be facing in the same direction in order to have one robot follow one another.

These problems need to be taken into account when designing a system which is capable of keeping two balancing robots at a set distance from each other. Developing the distance control system was done in phases, with the first phase simply trying to design a system that could keep the robot at a set distance from a large object.

5.1.2 Distance Controller Design

Taking into account the first two problems raised in section 5.1.1, a controller was designed which ran outside of the existing PID control system used to keep the robot upright, and would keep the robot at a set distance from a target. The control system uses the relative velocity between the robots as its control command u and the distance to the object as the output x. The is equation is shown in Eq 5.1:

$$\dot{x} = u \qquad or \qquad x = \frac{1}{s}u \tag{5.1}$$

Effectively what we require our distance control system to do is to move the robot more quickly if it is a large distance away but slow the robot down as it approaches the set distance. This will assist the balancing system by not applying large amount of disturbance when the robot is balancing around the desired set distance. A proportional gain controller was selected as it provided no overshoot and produced the desired step response. The distance control system was built on Simulink and is shown in Fig 5.2.

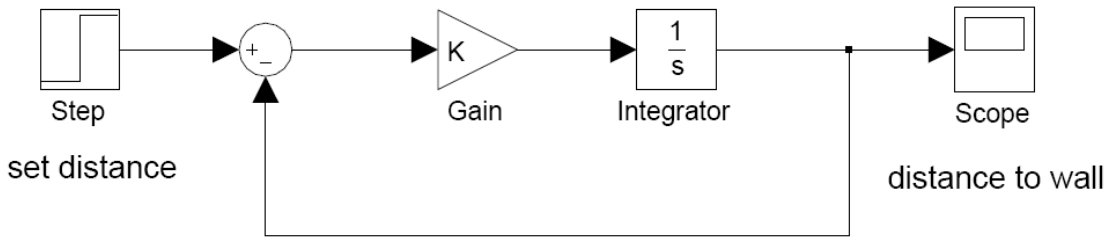


Figure 5.2: Simulink model of distance control system

The closed loop system of this model is $K/(S+K)$, with a single pole at $s = -K$.

With the control system now selected the next step is to choose an appropriate value for K. In selecting K we want to ensure that it does not approach the desired set distance too rapidly otherwise we can cause the robot to overshoot. The simulated model will not show this, but in practice it is difficult to stop a balancing robot instantly since the PID controller will oscillate about the stationary set point for some time. With this in mind several K

values were simulated using Matlab. The results are shown in Fig 5.3. Note that during the testing purposes the desired set distance was 20 cm.

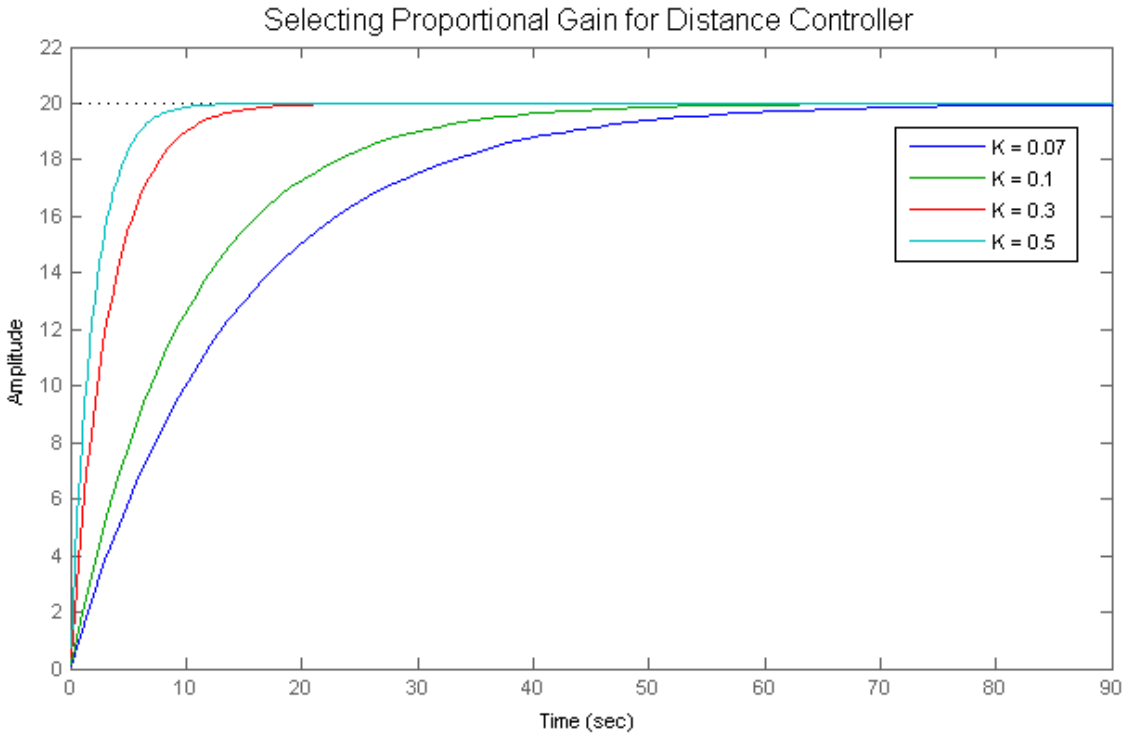


Figure 5.3: Step response of distance controller with varying gains

The ideal gain to meet our specifications would be $K = 0.1$ since it approaches the desired set distance gradually as it gets closer over a desired amount of time. With the control parameters for the distance controller now set the next step is to simulate it with the GELway balance controller presented in Fig 2.6 of Chapter 2. The Simulink model for the distance and balance controller is shown in Fig 5.4, noting that the balance controller has replaced the integrator in the distance controller. The balance controller has been added as a subsystem to show how the distance controller fits into the complete system.

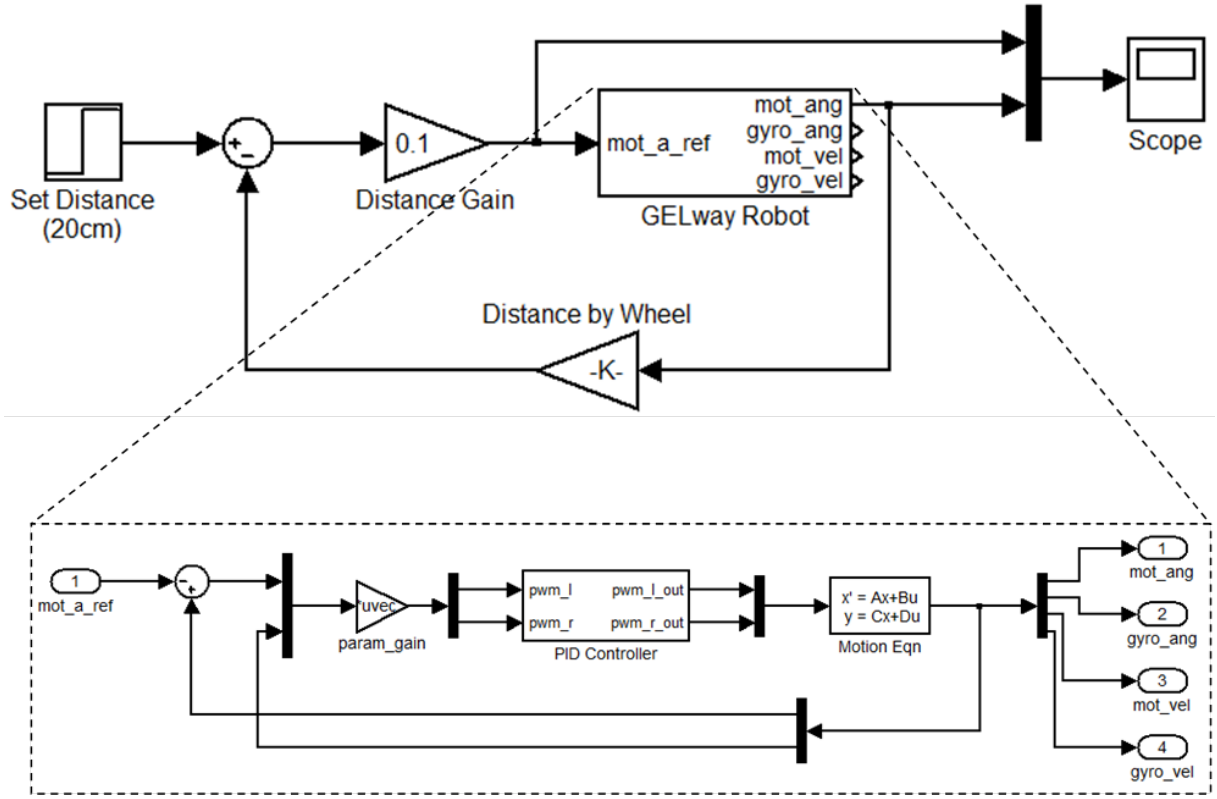


Figure 5.4: Simulink block diagram of distance and balance controller.

The feedback of the distance controller in Fig 5.4 used the motor angle, multiplied by the wheel circumference, to get the actual distance travelled by the robot. Note that when the actual distance controller was implemented in the robot, the ultrasonic sensor was used to measure the distance from the desired set point. When a distance step input of 20 is applied to the robot it will move to correct this reference by applying an offset to the body pitch angle until the desired distance is reached. Fig 5.5 shows the results of the Simulink model in Fig 5.4 when a unit step of 20 is applied.

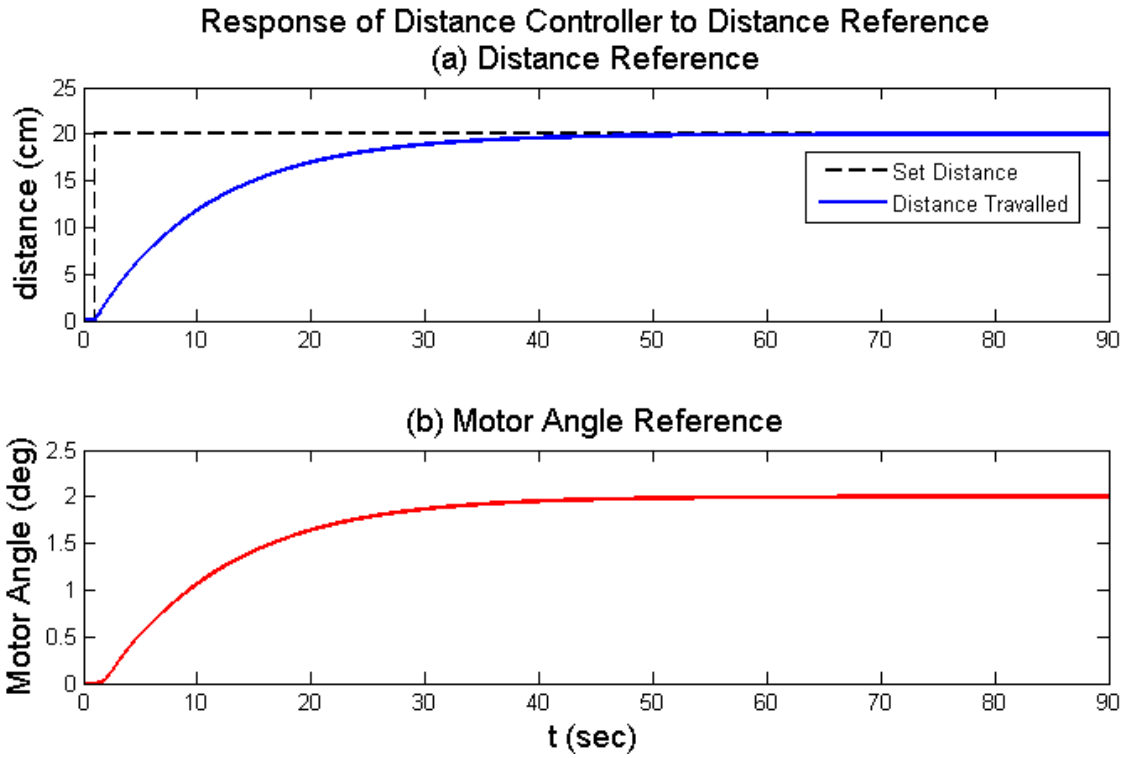


Figure 5.5: Simulink of the response of the distance controller to a distance reference of 20cm.

The results in Fig 5.5 gave us the desired result of having the robot gradually move until the desired distance set point was reached. Since the robot slows down as it approaches the desired set distance we have also prevented overshoot in the system which was a key trait we required for our distance controller.

Before the distance control system was implemented modifications to the follower GELway were made. The modifications involved a repositioning of the ultrasonic sensor of the GELway so that it would sit lower on the body point at the back of leading GELway. This is because this is the greatest surface area for the GELway and therefore has the greatest chance of allowing the follower robot to get a constant distance reading. Pictures and building instructions for the follower robot can be found in Appendix A.12.

5.2 Programming the Robot's Synchronous Movement

This section of the thesis focuses on implementing the distance control system into the follower GELway. It will cover how the distance controller was programmed into the GELway and how it performed once implemented. The section will also cover a suggested solution to account for the robot's tendency to rotate slowly over time.

5.2.1 How the Distance Controller was Programmed

Since the main role of the follower robot during the testing phase was to follow the leading robot it did not require its driving behaviour. This meant that we could replace the lowest level behaviour of driving the robot to a behaviour which would run the distance controller constantly to keep the follower robot at a desired set distance. It makes sense to have it as the lowest behaviour since it can be suppressed by higher behaviours which could allow the follower robot to move around freely without having it keep a set distance from the leader robot all the time. Due to distance controller being the main action in the behaviour it is said to be a servo behaviour as it utilises a feedback control loop as its control component.

To implement the distance control system we put the robot in place of the $1/s$ integrator box from Fig 5.2. From there we take a measured ultrasonic sensor reading and subtract the desired set distance which was set at 20 cm. This was then multiplied by the controller gain and applied as a reference motor angle signal to the follower robot. This process is repeated with the motor angle offset constantly increasing or decreasing resulting with a constant forwards or backwards movement respectively. This is shown in the segment of code below.

```
// Measured Distance
double dist = (double)us.getDistance();
// error = (measured - set) distance
error = dist - setDist;
// motor angle tilt power
double tiltPower = (K * error)
ctrl.setTiltAngle(tiltPower)
```

This designed controller worked perfectly when the GELway was started at a distance close to the desired set point and was able to maintain a constant distance. However if the GELway started at distances greater then approximately 5 to 10 cm away from the desired set point the disturbance sent to the balance controller was too great and it caused the GELway to fall over. To rectify this problem a tilt power offset cap was set at a distance of 5 cm so that no matter how far away the GELway was from the desired set distance it would keep a constant maximum relative speed. The source code for the *GELwayFollower* behaviour can be found in Appendix A.9.

With the cap in place the GELway was able to successfully keep a distance from an object. This was briefly tested by holding a piece of paper in front of the ultrasonic sensor and gradually moving it forwards and backwards to test if the GELway would respond accordingly. After we were satisfied that the GELway was maintaining a constant distance its distance controller parameters were data logged for further analysis. This will now be discussed in more depth in section 5.3.

5.2.2 Accounting for the GELway's Rotation

As addressed before, the rotation in the GELway over time makes it difficult for the robots to follow each other. One solution to this, which has been trialled and found to be effective, is to use the compass sensor to keep the GELway orientated in the same direction. During the testing phase of the solution it was implemented as a behaviour and worked by measuring the orientation of the GELway. The GELway would then be given a desired orientation and would rotate slowly to this position. This could be merged with the lowest level behaviour of the GELway, one that is constantly active. It would work by gradually rotating one motor faster than the other causing a slow rotation. This could be merged with the *GELwayFollower* behaviour anywhere during its *action()* method. The processes is simple: measure the orientation of the GELway and if it is out by a factor of approximately 3 degrees rotate the robot in the direction which will return it to the set orientation the fastest. It is simple to implement however could not be fully tested due to the limitation of only having one compass sensor.

5.3 Testing the Distance Controller

To test if the *GELwayFollower* behaviour was functioning correctly a test was set up which would place the GELway approximately 100 cm from the wall. This would test to see if the distance controller was capable of moving the GELway forwards towards the desired distance of 20cm without causing too much disturbance making the GELway fall over. What we should expect to see is the GELway slow down as it approaches the set distance and maintain this distance of 20cm. The results of the test were recorded using the leJOS Datalogger and can been seen in Fig 5.6.

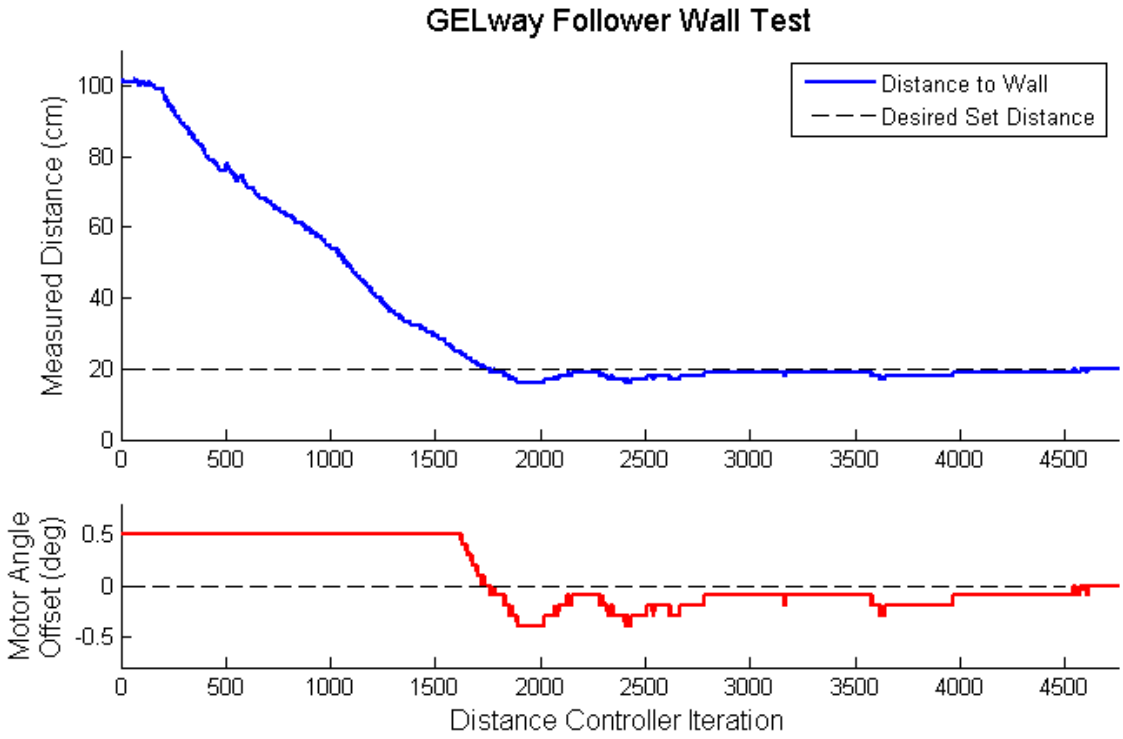


Figure 5.6: Testing the distance controller at a desired distance of 20cm

The top plot in Fig 5.6 shows the distance read by the ultrasonic sensor in centimetres over each distance controller iteration. The plot shows that once the GELway starts moving forwards it maintains a constant distance until it approaches the desired set distance of 20cm. Once the GELway reaches 20cm it maintains a constant distance with an error of approximately ± 2cm which can be accounted for by the constant forwards and backwards

movement whilst trying to maintain its upright position. The bottom plot in Fig 5.6 shows the motor angle offset for each distance controller iteration. We see that the offset has been capped at 0.5 degrees whilst the GELway is at a large distance away from the wall. As it approaches the wall the offset decreases allowing the GELway to keep at the desired set distance of 20cm. This test concludes that the distance controller is functioning correctly.

5.4 Chapter Conclusion

This chapter has covered the design and implementation of a distance control system that will allow two balancing robots to follow each other in a straight line. Due to the difficulty of this task and a large number of options to investigate complete synchronisation was not achieved, however the foundation has been laid for further improvements to made to the existing system. These include replacing the ultrasonic sensor with a sensor capable of differentiating the leading robot from other obstacles, such as the infrared sensor/beacon combination, and using compass sensors in both robots to keep them orientated in the same direction. This will be covered in more detail in the future recommendations for this project in section 6.2. This project has proven that the distance controller designed for synchronising a string of robots works by keeping the follower robot at a set distance from a given object. With new distance sensors it should be possible to have robots synchronise their movements in a string formation.

Chapter 6

Conclusion and Recommendations

The intention of this project was to design a scalable software platform for reconfigurable robots and carry out initial research into such robots using LEGO Mindstorms technology. To achieve this goal the project was separated into several key areas, each with their own contribution to reaching this software platform. First, developing the balancing system allowed us to test the capabilities of having robots run multiple tasks simultaneously, a trait which is common in most reconfigurable robots. Secondly, the behavioural system allowed us to bring all the other systems together and enabled the robots to be controlled, move around autonomously and most importantly to make their own decisions based on the information they gathered from their sensors. Thirdly, the communicative system was developed which paved the way for allowing robots to share information and control each other without operator intervention. Lastly, we laid the groundwork for the synchronisation of multiple robots by allowing the robots to move around as a group rather than individuals, improving their ability to solve tasks and make decisions based on collaborative information. Each of the chapters presented in this thesis will first be discussed, covering the final conclusions about each of the systems developed for the robot in this project. Following this, recommendations for future work will be provided, suggesting possible avenues to develop the work described in this thesis.

6.1 Conclusions

As discussed in Chapter 2 the balancing robot was selected as it was a well researched design, reducing the potential risks and allowing us to focus on the main aims of the project. By improving on the documentation and explanations on previous two-wheeled, balancing robotic designs this project will hopefully reduce the learning curve required when future work is carried out on similar designs. Throughout the course of the year we have tweaked and optimised the balancing system, and provided new functionality such as self calibrating methods for the gyroscope sensor, a self reset mechanism and movement controls. This has provided us with a unique balancing robot with a higher level of portability in its software design.

The behavioural system developed in Chapter 3 provided the robots with the means of making their own decisions based on the environmental input. The behavioural system was an essential step to developing the scalable platform for reconfigurable robots as one of the main features in these robots is their ability to adapt to change. By using the behavioural system with sensor inputs as the behaviour triggers we can program the robot to act differently depending on its environment. With the ability to share sensors between multiple robots the option is there to allow for decisions to be made jointly, thus improving the group dynamics of the robots. The key goal of the behavioural system was to make it simple for future functionality to be added or removed from the robot with relative ease, which was successfully achieved by implementing the subsumption architecture style arbitration system.

The communications system, covered in Chapter 4, provided us with the means to control the movements of the robot, but more specifically the capability to have multiple robots work together co-operatively and share information between themselves. The communication system is crucial in the project since without it we effectively have independent robots incapable of working together synchronously, a vital trait of reconfigurable robots. The communications system, as discussed in Chapter 4, was successfully implemented by using the Bluetooth communication feature on the NXT allowing for connections to external Bluetooth devices and other NXT systems. The communications feature allowed us to

conduct further testing in the robots such as synchronised movements and joint-triggered behaviours in the robot.

The final stage of the project was to design a control system to allow multiple balancing robots to follow each other as the leader robot moved around, synchronising their movements. Chapter 5 provided an analysis of the feasibility of this task and covered the groundwork that had been laid to successfully fulfil this goal. Allowing the robots to synchronise their movements will further improve the ability of multiple autonomous robots to work cooperatively to solve problems. The project has developed the principle solution to this problem, and investigated the ways to further perfect this solution. In particular, recommendations are made as to enhancing the hardware equipment.

6.2 Recommendations

There are numerous possibilities to develop the scalable software designed for reconfigurable robots in this project. This section will provide several avenues that could be explored for this project detailing suggestions on how they could be approached.

6.2.1 Communications System

The communication system can be expanded further to allow for more robots to be added to the system, each with their own specific sensors providing a more detailed picture of the robot's surroundings. This will allow more research to be conducted into how the master robot handles multiple slaves and how it processes all the incoming information. With more active communications streams between the robots there will be a larger focus on data control, and as such, the communication system will need to ensure it has procedures in place for the master to manage information sent from the slaves.

6.2.2 Behavioural System

With the behavioural system programmers will be able to take the existing design developed in this project and add their own functionality to the robot. This could open up the possibility of allowing the robots to solve more complex tasks by using a larger set of behaviours. With the option of using multiple robots, future projects could involve programming behaviours that allow for joint search and rescue operations or mapping out rooms as a group of robots.

6.2.3 Distance Control System

Whilst the implemented distance control system was capable of keeping the follower robot at a set distance from a large object it was limited in achieving the ultimate goal of following the leading GELway. This is not so much a problem with the distance controller itself, but more to do with the limited resources available in the laboratory. In ideal circumstances the following recommendations are advised to better improve the synchronisation on two balancing robots.

Firstly have both robots fitted with a compass sensor which will allow them to be facing in the same direction at all times. During the testing only one robot was able to have a compass sensor attached. Note that with both robots having a compass sensor fitted the master robot can orientate the slave robot to face the same direction even after turning of moving around.

The major issue with the testing that was carried out on the distance controller was the sensor on the follower used to measure the distance from the leader robot. The ultrasonic sensor is an effective distance measuring device, but only when facing directly at a target. Due to the small space on the leader robot,s back even the slightest turn in either robot made it extremely difficult to measure a constant distance. This is why most of the testing was done using large surface areas such as a piece of paper or a wall. Whilst it has not been tested during this project a suggested extension would be to replace the ultrasonic sensor with HiTechnic's NXT IRSeeker V2 for the follower robot and an attached IR Beacon on

the back of the leader robot, pictured in Fig 6.1 respectively [39, 40].

Figure 6.1: HiTechnics (a) IRSeeker V2 and (b) IR Beacon [39, 40]

The reason why the IR Seeker is a better replacement is that it not only returns the signal strength but also returns the signal direction as well, up to a range of 240 degrees [39]. This would help limit the problem of the robots slowly rotating over time, and also means the follower robots do not have to directly face the leading robot. The suggested solution would involve powering the IR Beacon on the back of the leading GELway to output a constant infrared signal and have the follower robot use the IR Seeker to find and follow the IR Beacon's signal at a set distance. Whilst it has not been tested, this solution has the possibility of eliminating a lot of the problems associated with using the ultrasonic sensor.

Another issue with using the ultrasonic sensor is its inability to decipher between the leading robot and another obstacle. The follower robot could get locked onto another target keeping a set distance from that if it loses track of the leading robot. This problem can be fixed by again using the IRSeeker/IR Beacon combination since in ideal circumstances the only infrared source in the room will be the one on the back of the leader robot.

Note that these suggestions involve using more sensor ports which are already limited in number. HiTechnic have recently produced a Sensor Multiplexer which will allow up to four sensors to be attached to one sensor port on the NXT [41]. This would not only allow more sensors to be attached but could also help improve the behavioural system since it would allow more sensor triggered behaviours.

Taking these suggestions into account and using the existing distance control system the

balancing robots should be able to follow each other in a line autonomously without operator intervention.

6.2.4 Reconfigurable Modular Robots

A possible future direction for this project involves taking the robot into the modular robotic world where balancing robots are capable of connecting with each other to form new systems. By developing a two-wheeled, balancing robot into a modular system it would be possible to connect robots into four or more wheeled systems that are capable of achieving different tasks such as moving heavier loads, traversing rocky terrain and multiple processing of data.

6.2.5 Robot Platoons

Once the communication and distance control systems have been developed to include numerous robots capable of communicating and moving in set formations a possible future for the two-wheeled, balancing robots could be robot platooning. Each robot would have a variety of different sensors attached, giving them there own unique role within the platoon. In the robot platoon the master robot would effectively be the commanding officer, detailing specific commands for the slaves to execute.

All of these future research areas are possible due to the scalable nature of this project. In conclusion this project topic of Coordinated LEGO Segways has set out and achieved the development of a scalable software architecture for reconfigurable robots. The programmable robot used in this project has shown itself to be capable of running multiple tasks simultaneously, communicating and synchronising with other systems and adapt to changes in its environment through the use of it behavioural system. With future work to develop the current system it could be possible to reach the ultimate goal of designing a complete, reconfigurable, autonomous robot.

Appendix A

Appendices

A.1 Balancing Dynamics

A.1.1 Modelling GELway - Two Wheeled Inverted Pendulum

As the GELway behaves similarly to an inverted pendulum it can be modelled as one. This is depicted in Fig A.1.

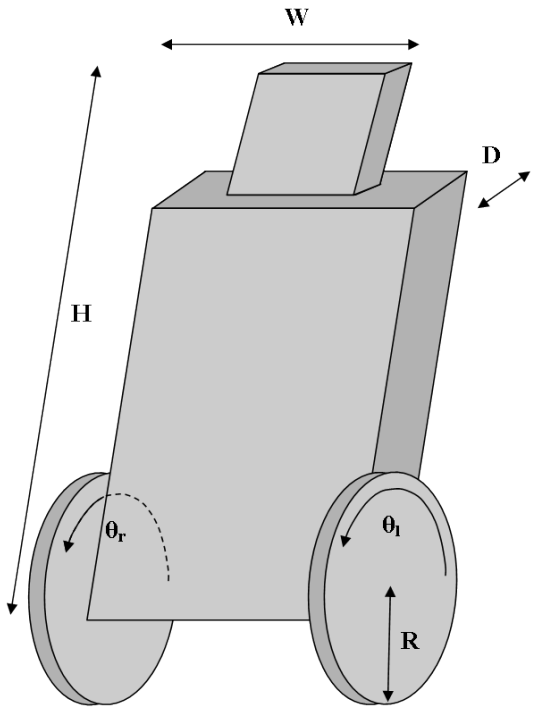


Figure A.1: Two-wheeled inverted pendulum [3].

The coordinate system used to model the GELway can be seen in Fig A.2. The motion equations derived for the GELway will be based off these coordinates.

Before the state-space and coordinate equations are presented a table detailing of all the relevant parameters will be given and their respective units. This has been provided in Table A.1.

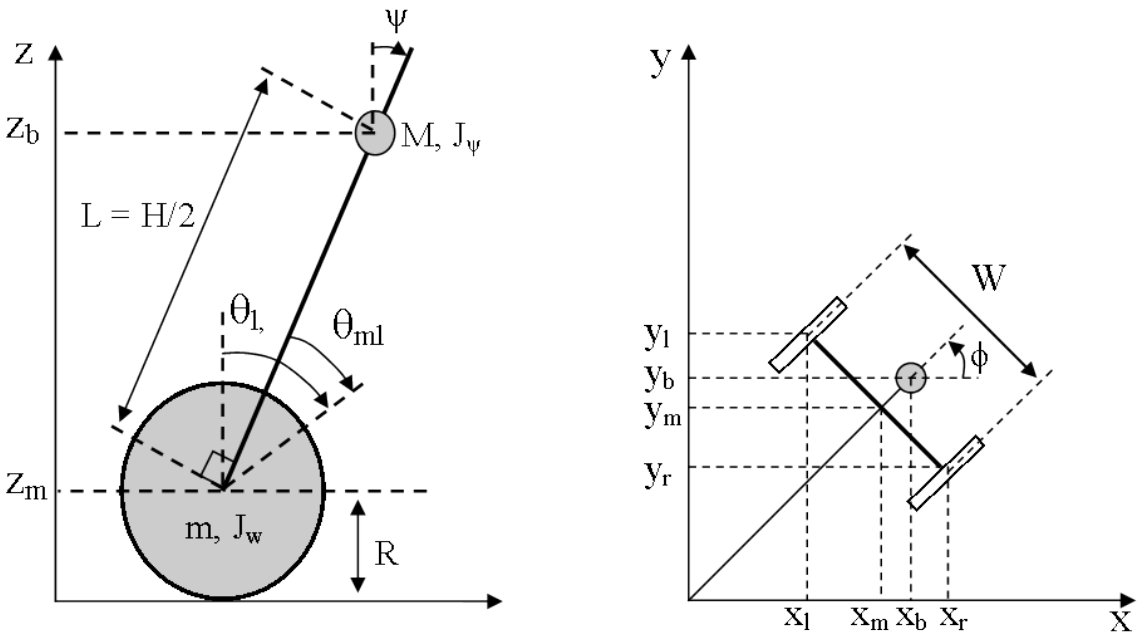


Figure A.2: Coordinate system for equations of motion for the GELway [3].

Parameter	**Unit**	**Description**
θ	[deg]	Average angle of wheels
ψ	[deg]	Body pitch angle
ϕ	[deg]	Body yaw angle
θ_m	[deg]	DC Motor angle
g	$[m/sec^2]$	Gravity acceleration
m	[kg]	Wheel weight
R	[m]	Wheel radius
J_w	$[kgm^2]$	Wheel inertial moment
M	[kg]	Body weight
W	[m]	Body width
D	[m]	Body deptth
H	[m]	Body height
L	[m]	Body length
J	$[kgm^2]$	Inertia moment
J_ψ	$[kgm^2]$	Body pitch inertial moment
J_ϕ	$[kgm^2]$	Body yaw inertial moment
J_m	$[kgm^2]$	DC motor inertial moment
n		Gear ratio
K_t	[Nm/A]	DC motor torque constant
K_b	[V sec/rad]	DC motor back EMF constant
f_m		Friction coefficient between body and DC motor
f_w		Friction coefficient between wheel and floor.

Table A.1: State-equation parameters for the GELway

A.1.2 Deriving GELway Motion Equations

Using the coordinate system presented in Fig A.2 and the Lagrangian method the motion equations for a two–wheeled inverted pendulum can be derived. If we assume the direction of the two–wheeled inverted pendulum is in the positive x–axis and time $t = 0$, the following coordinates are given.

The following variables are used in the proceeding equations:

θ : Angle of left and right wheel (averaged)
ψ : Angle of body pitch
ϕ : Angle of body yaw)

$$(\theta,\ \phi) = \left(\frac{1}{2}(\theta_l + \theta_r), \frac{R}{W}(\theta_l - \theta_r)\right) \tag{A.1}$$

$$(x_m,\ y_m,\ z_m) = \left(\int \dot{x}_m \,\mathrm{d}t, \int \dot{y}_m \,\mathrm{d}t, R\right) \tag{A.2}$$

$$(\dot{x}_m,\ \dot{y}_m) = \left(R\dot{\theta}\cos\phi,\ R\dot{\theta}\sin\phi\right) \tag{A.3}$$

$$(x_l,\ y_l,\ z_l) = \left(x_m\ -\ \frac{W}{2}\sin\phi,\ y_m + \frac{W}{2}\cos\phi,\ z_m\right) \tag{A.4}$$

$$(x_r,\ y_r,\ z_r) = \left(x_m + \frac{W}{2}\sin\phi,\ y_m - \frac{W}{2}\cos\phi,\ z_m\right) \tag{A.5}$$

$$(x_b,\ y_b,\ z_b) = (x_m + L\sin\psi\cos\phi,\ y_m + L\sin\psi\sin\phi,\ z_m + L\cos\psi) \tag{A.6}$$

Using Eqs. (A.1) to (A.6) the translational kinetic energy $T-1$, the rotational kinetic energy T_2, and the potential energy U can be found.

$$\begin{aligned} T_1 &= \frac{1}{2}m\left(\dot{x_l}^2 + \dot{y_l}^2 + \dot{z_l}^2\right) + \frac{1}{2}m\left(\dot{x_r}^2 + \dot{y_r}^2 + \dot{z_r}^2\right) \\ &\quad + \frac{1}{2}M\left(\dot{x_b}^2 + \dot{y_b}^2 + \dot{z_b}^2\right) \end{aligned} \tag{A.7}$$

$$\begin{aligned} T_2 &= \frac{1}{2}J_w\dot{\theta_l}^2 + \frac{1}{2}J_w\dot{\theta_r}^2 + \frac{1}{2}J_\psi\dot{\psi}^2 + \frac{1}{2}J_\phi\dot{\phi}^2 + \frac{1}{2}n^2J_m\left(\dot{\theta}_l - \dot{\psi}\right)^2 \\ &\quad - \frac{1}{2}n^2J_m\left(\dot{\theta}_r - \dot{\psi}\right)^2 \end{aligned} \tag{A.8}$$

$$U = mgz_l + mgz_r + Mgz_b \tag{A.9}$$

The Lagrangian method consists of the following expression:

$$L = T_1 + T_2 - U \tag{A.10}$$

With the Lagrange equations being the following:

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = F_\theta \tag{A.11}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{\psi}}\right) - \frac{\partial L}{\partial \psi} = F_\psi \tag{A.12}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{\phi}}\right) - \frac{\partial L}{\partial \phi} = F_\phi \tag{A.13}$$

By substituting Eqs. (A.7) – (A.10) in Eqs. (A.11), (A.12) and (A.13) we get the following equations:

$$\begin{aligned} \left((2m+M)R^2 + 2J_w + 2n^2J_m\right)\ddot{\theta} + \left(MLR\cos\psi - 2n^2J_m\right)\ddot{\psi} & \\ -MLR\dot{\psi}^2\sin\psi &= F_\theta \end{aligned} \tag{A.14}$$

$$\begin{aligned} \left(MLR\cos\psi - 2n^2J_m\right)\ddot{\theta} + \left(ML^2 + J_\psi + 2n^2J_m\right)\ddot{\psi} & \\ -MgL\sin\psi - ML^2\dot{\phi}^2\sin\psi\cos\psi &= F_\psi \end{aligned} \tag{A.15}$$

$$\begin{aligned} \left(\frac{1}{2}mW^2 + J_\phi + \frac{W^2}{2R^2}\left(J_w + n^2J_m\right) + ML^2\sin^2\psi\right)\ddot{\phi} & \\ +2ML^2\dot{\psi}\dot{\phi}\sin\psi\cos\psi &= F_\phi \end{aligned} \tag{A.16}$$

Using the DC motor torque and viscous friction, the generalised forces equate to the following:

$$(F_\theta,\ F_\psi,\ F_\phi) = \left(F_l + F_r, F_\psi,\ \frac{W}{2R}(F_r - F_l)\right) \tag{A.17}$$

$$F_l = nK_ti_l + f_m(\dot{\psi} - \dot{\theta}_l) - f_w\dot{\theta}_l \tag{A.18}$$

$$F_r = nK_ti_r + f_m(\dot{\psi} - \dot{\theta}_r) - f_w\dot{\theta}_r \tag{A.19}$$

$$F_\psi = -nK_ti_l - nK_ti_r - f_m(\dot{\psi} - \dot{\theta}_l) - f_m(\dot{\psi} - \dot{\theta}_r) \tag{A.20}$$

where $i_{l,r}$ is DC motor current.

The DC motor current cannot be used directly since the LEGO Mindstorms servo motor reads the motor voltage in PWM [42]. Thus the relation between current $i_{l,r}$ and voltage $v_{l,r}$ needs to be evalutated using the DC motor equation. The general DC motor equation is as follows:

$$L_m \dot{i_{l,r}} = v_{l,r} + K_b(\dot{\psi} - \dot{\theta_{l,r}}) - R_m i_{l,r} \tag{A.21}$$

We assume that the motor inductance is neglible and approximate $i_{l,r}$ as the following

$$i_{l,r} = \frac{v_{l,r} + K_b(\dot{\psi} - \dot{\theta_{l,r}})}{R_m} \tag{A.22}$$

Using Eq. (A.22), the generalised force expressions can be found in terms of the motor voltage:

$$F_\theta = \alpha(v_l + v_r) - 2(\beta + f_w)\dot{\theta} + 2\beta\dot{\psi} \tag{A.23}$$

$$F_\psi = -\alpha(v_l + v_r) + 2\beta\dot{\theta} - 2\beta\dot{\psi} \tag{A.24}$$

$$F_\phi = \frac{W}{2R}\alpha(v_r - v_l) - \frac{W^2}{2R^2}(\beta + f_w)\dot{\phi} \tag{A.25}$$

$$\text{where} \quad \alpha = \frac{nK_t}{R_m}, \quad \beta = \frac{nK_tK_b}{R_m} + f_m$$

A.1.3 State Equations for the GELway

To simplify the motion Eqs. (A.14) – (A.16) we linearise them about the GELway's balance point, i.e. $\psi \to 0$. This means we can simplify the equations by substituting $\sin\psi \to \psi$ and $\cos\psi \to 1$ and removing second order terms like $\dot{\psi}^2$ since they will be significantly smaller compared to other terms.

Therefore the motion Eqs (A.14) – (A.16) are approximated as the following:

$$\left((2m + M)R^2 + 2J_w + 2n^2J_m\right)\ddot{\theta} + \left(MLR - 2n^2J_m\right)\ddot{\psi} = F_\theta \tag{A.26}$$

$$\left(MLR - 2n^2J_m\right)\ddot{\theta} + \left(ML^2 + J_\psi + 2n^2J_m\right)\ddot{\psi} - MgL\psi = F_\psi \tag{A.27}$$

$$\left(\frac{1}{2}mW^2 + J_\phi + \frac{W^2}{2R^2}\left(J_w + n^2J_m\right)\right)\ddot{\phi} = F_\phi \tag{A.28}$$

Eq. (A.26) and (A.27) can be grouped together as they both consist of angles θ and ψ. Eq. (A.28) has only the yaw angle ϕ. The first two equations can now be expressed as:

$$S\begin{bmatrix}\ddot{\theta}\\ \ddot{\psi}\end{bmatrix} + T\begin{bmatrix}\dot{\theta}\\ \dot{\psi}\end{bmatrix} + U\begin{bmatrix}\theta\\ \psi\end{bmatrix} = V\begin{bmatrix}v_l\\ v_r\end{bmatrix} \tag{A.29}$$

where

$$\begin{aligned}
S &= \begin{bmatrix} (2m+M)R^2 + 2J_w + 2n^2J_m & MLR - 2n^2J_m \\ MLR - 2n^2J_m & ML^2 + J_\psi + 2n^2J_m \end{bmatrix} \\
T &= 2\begin{bmatrix} \beta + f_w & -\beta \\ -\beta & \beta \end{bmatrix} \\
U &= 2\begin{bmatrix} 0 & 0 \\ 0 & -MgL \end{bmatrix} \\
V &= 2\begin{bmatrix} \alpha & \alpha \\ -\alpha & -\alpha \end{bmatrix}
\end{aligned}$$

Furthermore, Eq. (A.28)can be written in the form:

$$I\ddot{\phi} + J\dot{\phi} = K(v_r - v_l) \tag{A.30}$$

where

$$\begin{aligned}
I &= \frac{1}{2}mW^2 + J_\phi + \frac{W^2}{2R^2}\left(J_w + n^2J_m\right) \\
J &= \frac{W^2}{2R^2}(\beta + f_w) \\
K &= \frac{W}{2R}\alpha
\end{aligned}$$

The next step is to write the state space equations. Using the following variables x_1, x_2 as the state variables for Eq. (A.29) and (A.30) respectively, and u is the input variable:

$$x_1 = \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \quad x_2 = \begin{bmatrix} \phi \\ \dot{\phi} \end{bmatrix} \quad u = \begin{bmatrix} v_l \\ v_r \end{bmatrix} \tag{A.31}$$

we can rewrite Eq. (A.29) and (A.30) in the state space form as follows:

$$\dot{x_1} = A_1 x_1 + B_1 u \tag{A.32}$$

$$\dot{x_2} = A_2 x_2 + B_2 u \tag{A.33}$$

where

$$A_1 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & A_1(3,2) & A_1(3,3) & A_1(3,4) \\ 0 & A_1(4,2) & A_1(4,3) & A_1(4,4) \end{bmatrix}, \quad B_1 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ B_1(3) & B_1(3) \\ B_1(4) & B_1(4) \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 1 \\ 0 & -J/I \end{bmatrix}, \quad B_2 = \begin{bmatrix} 0 & 0 \\ -K/I & K/I \end{bmatrix}$$

$$\begin{aligned} A_1(3,2) &= -gMLE(1,2)/det(E) \\ A_1(4,2) &= gMLE(1,1)/det(E) \\ A_1(3,3) &= -2[(\beta + f_w)E(2,2) + \beta E(1,2)]/det(E) \\ A_1(4,3) &= 2[(\beta + f_w)E(1,2) + \beta E(1,1)]/det(E) \\ A_1(3,4) &= 2\beta[E(2,2) + E(1,2)]/det(E) \\ A_1(4,4) &= -2\beta[E(1,1) + E(1,2)]/det(E) \\ B_1(3) &= \alpha[E(2,2) + E(1,2)]/det(E) \\ B_1(3) &= -\alpha[E(2,2) + E(1,2)]/det(E) \\ det(E) &= E(1,1)E(2,2) - E(1,2)^2 \end{aligned}$$

A.2 GELway.java Source Code

```
/* -*- tab-width: 2; indent-tabs-mode: nil; c-basic-offset: 2 -*- */
import lejos.nxt.*;
import lejos.nxt.comm.BTConnection;
import lejos.nxt.comm.Bluetooth;
import lejos.nxt.comm.NXTConnection;
import javax.bluetooth.RemoteDevice;
import lejos.subsumption.*;
import lejos.util.Datalogger;
/**
 * This is the GELways main program. It initiates the Bluetooth connection and establishes
 * the balancing and behavioural threads. Note this program has been based off Marvin the
 * Balancing robot and has been modified by myself. As such, they should be properly
 * referenced if you intend on modifying this code. The creators of Marvin are Bent Bisballe
 * Nyeng, Kasper Sohn and Johnny Rieper
 *
 * @author Steven Jan Witzand
 * @version August 2009
 */
class GELway extends Thread
{
   static CtrlParam ctrl;
   static BluetoothReader slave;
   static BluetoothReader br;
   public static void main(String[] args)
   {
      ctrl = new CtrlParam();
      // loadMaster(); // Load the Master GELway settings
      loadSlave(); // Load the Slave GELway settings
      // loadNormal(); // Load the normal GELway settings
   }
   /**
    * Loads the master setting for the robot. Establishes the remote control and slave
    * Bluetooth connections.
    */
   public static void loadMaster() {
      LCD.drawString("Waiting...", 2, 1);
      BTConnection conn = Bluetooth.waitForConnection();
      conn.setIOMode(0); // Used when a pc connection is made
      // conn.setIOMode(NXTConnection.RAW); // Used when a phone connection is made
      LCD.clear();
      // Start Bluetooth reader thread
      br = new BluetoothReader(conn);
      br.start();
      connectMasterSlave();
      // Start Balance control thread
      BalanceController bc = new BalanceController(ctrl);
      bc.start();
      Behavior b1 = new GELwayDriver(ctrl, br); // Needed to drive robot and slave
      Behavior b2 = new DetectObstacle(ctrl); // Needed to avoid obstacles
      Behavior[] bArray = { b1, b2 };
      Arbitrator arby = new Arbitrator(bArray);
      arby.start();
   }
   /**
    * Loads the slave setting for the robot. Has the follower behaviour to keep GELway at a
    * set distance
    */
   public static void loadSlave() {
      LCD.drawString("Waiting...", 2, 1);
      BTConnection conn = Bluetooth.waitForConnection();
      conn.setIOMode(0); // Used when a pc connection is made
      // conn.setIOMode(NXTConnection.RAW); // Used when a phone connection is made
      LCD.clear();
      // Start Bluetooth reader thread
      br = new BluetoothReader(conn);
      br.start();
      // Start Balance control thread
```

```
        BalanceController bc = new BalanceController(ctrl);
        bc.start();
        // Behavior b1 = new GELwayDriver(ctrl, br); // Needed to drive robot and slave
        Behavior b1 = new GELwayFollower(ctrl); // Needed to avoid obstacles
        // Behavior b2 = new SlaveObstacle(ctrl, br); // Needed to avoid obstacles
        Behavior[] bArray = { b1 };
        Arbitrator arby = new Arbitrator(bArray);
        arby.start();
    }
    /**
     * Loads operator settings for GELway when only one robot is used
     */
    public static void loadNormal() {
        LCD.drawString("Waiting...", 2, 1);
        BTConnection conn = Bluetooth.waitForConnection();
        // conn.setIOMode(0); // Used when a pc connection is made
        conn.setIOMode(NXTConnection.RAW); // Used when a phone connection is made
        LCD.clear();
        // Start Bluetooth reader thread
        br = new BluetoothReader(conn);
        br.start();
        // Start Balance control thread
        // BalanceController bc = new BalanceController(ctrl);
        BalanceController bc = new BalanceController(ctrl);
        bc.start();
        LCD.clear();
        Behavior b1 = new GELwayDriver(ctrl, br); // Needed to drive robot and slave
        // Behavior b1 = new GELwayFollower(ctrl);
        Behavior b2 = new DetectObstacle(ctrl); // Needed to avoid obstacles
        // Behavior b3 = new KeepStraight(ctrl); // Needed to keep straight
        Behavior[] bArray = { b1, b2 };
        Arbitrator arby = new Arbitrator(bArray);
        arby.start();
    }
    /**
     * This method is used to connect the Master GELway to the Slave GELway.
     */
    public static void connectMasterSlave() {
        LCD.drawString("Master Connected", 0, 4);
        LCD.drawString("Press Enter to", 1, 6);
        LCD.drawString("Connect Slave", 2, 7);
        try {Button.ENTER.waitForPressAndRelease();} catch (InterruptedException e1) {}
        LCD.clear();
        String name = "GELwayJR";
        LCD.drawString("Connecting...", 0, 0);
        LCD.refresh();
        RemoteDevice btrd = Bluetooth.getKnownDevice(name);
        if (btrd == null) {
            LCD.clear();
            LCD.drawString("No such device", 0, 0);
            LCD.refresh();
            try {Thread.sleep(2000);} catch (InterruptedException e) {}
            System.exit(1);
        }
        BTConnection btc = Bluetooth.connect(btrd);
        // btc.setIOMode(0);
        if (btc == null) {
            LCD.clear();
            LCD.drawString("Connect fail", 0, 0);
            LCD.refresh();
            try {Thread.sleep(2000);} catch (InterruptedException e) {}
            System.exit(1);
        }
        LCD.clear();
        LCD.drawString("Connected Slave", 0, 0);
        LCD.refresh();
        slave = new BluetoothReader(btc);
        slave.start();
    }
}
```

A.3 BalanceController.java Source Code

```
import lejos.nxt.*;
import lejos.nxt.addon.EOPD;
import lejos.util.Datalogger;
/**
 * This class contains the parameters needed to keep the GELway balanced. It contains a PID
 * controller which reads in the angles and the angle velocities from the gyro sensor and
 * the left and right motor, and uses these (weighted) values to calculate the motor output
 * required to keep the GELway balanced.
 *
 * @author Steven Witzand
 * @version March 2009
 */
public class BalanceController extends Thread
{
   // The PID control parameters
   private final double Kp = 1.2;
   private final double Ki = 0.25;
   private final double Kd = 0.1;
   private final int eopdThresh = 1022;
   double num = 0.0;
   int startLog = 0;
   static double damp = 0.1;
   private static final int stationary = 0; // left
   private static final int forwards = 1; // up
   private static final int backwards = 2; // down
   static boolean upright = true;
   // Teseing error contributions.
   private final double K_psi = 44.257035; // Gyro angle weight
   private final double K_phi = 0.806876; // Motor angle weight
   private final double K_psidot = 0.620882; // Gyro angle velocity weight
   private final double K_phidot = 0.039711;// Motor angle velocity weight
   // Original Balance numbers
   // private final double K_psi = 34.189581; // Gyro angle weight
   // private final double K_phi = 0.835082; // Motor angle weight
   // private final double K_psidot = 0.646772; // Gyro angle velocity weight
   // private final double K_phidot = 0.028141; // Motor angle velocity weight
   private static CtrlParam ctrl;
   public boolean offsetDone = false;
   /**
    * BalanceController constructor.
    *
    * @param ctrl The motor control parameters.
    */
   public BalanceController(CtrlParam ctrl)
   {
      this.ctrl = ctrl;
      setDaemon(true);
   }

   /**
    * The BalanceController thread which constantly runs to keep the GELway upright
    */
   public void run()
   {
      MotorController motors = new MotorController(Motor.C, Motor.A);
      GyroscopeSensor gyro = new GyroscopeSensor(SensorPort.S3);
      EOPD eopd = new EOPD(SensorPort.S2);
      eopd.setModeLong();
      double int_error = 0.0;
      double prev_error = 0.0;
      while (true) {
         // Start balancing provided GELway is upright and the EOPD sensor can sense the
         // ground
         while (eopd.readRawValue() < eopdThresh && upright) {
            ctrl.setUpright(true);
            runDriveState();
            double Psi = gyro.getAngle();
```

```
            double PsiDot = gyro.getAngleVelocity();
            // ctrl.tiltAngle() is used to drive the robot forwards and backwards
            double Phi = motors.getAngle() - ctrl.tiltAngle();
            double PhiDot = motors.getAngleVelocity();
            // Proportional Error
            double error = Psi * K_psi + Phi * K_phi + PsiDot * K_psidot + PhiDot
                  * K_phidot;
            // Integral Error
            int_error += error;
            // Derivative Error
            double deriv_error = error - prev_error;
            prev_error = error;
            // Power sent to the motors
            double pw = (error * Kp + deriv_error * Kd + int_error * Ki) * 1;
            motors.setPower(pw + ctrl.leftMotorOffset(), pw + ctrl.rightMotorOffset());
            // Delay used to stop Gyro being read to quickly. May need to be increase or
            // decreased depending on leJOS version.
            delay(6);
         }
         startLog = 0;
         motors.stop();
         upright = false;
         ctrl.setUpright(false);
         while (eopd.readRawValue() > eopdThresh) {}
         // Restart the robot after the third beep. Reset balance parameters
         if (eopd.readRawValue() < eopdThresh) {
            for (int i = 0; i < 3; i++) {
               if (eopd.readRawValue() > eopdThresh) break;
               Sound.setVolume(50 + i * 25);
               Sound.beep();
               delay(700);
            }
            gyro.resetGyro();
            motors.resetMotors();
            ctrl.resetTiltAngle();
            int_error = 0.0;
            prev_error = 0.0;
            ctrl.setDriveState(stationary);
            upright = true;
         } else { motors.stop(); }
      }
   }

   public static void delay(int time)
   {
      try {Thread.sleep(time);} catch (Exception e) {}
   }

   /**
    * Returns the current upright state of the GELway. TRUE = Upright
    *
    * @return upright current upright state of the GELway
    */
   public static boolean getUpright() { return upright; }

   /**
    * Run drive state is used to keep the GELway in a constant state of forwards or
    * backwards motion. It can be set to keep the GELway stationary as well.
    */
   public static void runDriveState()
   {
      if (ctrl.getDriveState() == forwards)
         ctrl.setTiltAngle(3 - 3 * Math.exp(-ctrl.getDamp()));
      else if (ctrl.getDriveState() == backwards)
         ctrl.setTiltAngle(-5 + 3 * Math.exp(-ctrl.getDamp()));
      else
         ctrl.setTiltAngle(0);
      ctrl.setDamp(0.1);
   }
}
```

A.4 MotorController.java Source Code

```
/* -*- tab-width: 2; indent-tabs-mode: nil; c-basic-offset: 2 -*- */
import lejos.nxt.*;

/**
 * A class used to handle controlling the motors. Has methods to set the motors speed and
 * get the motors angle and velocity. Based off original programmers of Marvin Bent Bisballe
 * Nyeng, Kasper Sohn and Johnny Rieper
 *
 * @author Steven Jan Witzand
 * @version August 2009
 */
class MotorController
{

   private Motor leftMotor;
   private Motor rightMotor;
   // Sinusoidal parameters used to smooth motors
   private double sin_x = 0.0;
   private final double sin_speed = 0.1;
   private final double sin_amp = 20.0;

   /**
    * MotorController constructor.
    *
    * @param leftMotor
    *           The GELways left motor.
    * @param rightMotor
    *           The GELways right motor.
    */
   public MotorController(Motor leftMotor, Motor rightMotor)
   {
      this.leftMotor = leftMotor;
      this.leftMotor.resetTachoCount();

      this.rightMotor = rightMotor;
      this.rightMotor.resetTachoCount();
   }

   /**
    * Method is used to set the power level to the motors required to keep it upright. A
    * dampened sinusoidal curve is applied to the motors to reduce the rotation of the
    * motors over time from moving forwards and backwards constantly.
    *
    * @param leftPower
    *           A double used to set the power of the left motor. Maximum value depends on
    *           battery level but is approximately 815. A negative value results in motors
    *           reversing.
    * @param rightPower
    *           A double used to set the power of the right motor. Maximum value depends on
    *           battery level but is approximately 815. A negative value results in motors
    *           reversing.
    */
   public void setPower(double leftPower, double rightPower)
   {
      sin_x += sin_speed;
      int pwl = (int) (leftPower + Math.sin(sin_x) * sin_amp);
      int pwr = (int) (rightPower - Math.sin(sin_x) * sin_amp);

      leftMotor.setSpeed(pwl);
      if (pwl < 0) {
         leftMotor.backward();
      } else if (pwl > 0) {
         leftMotor.forward();
      } else {
         leftMotor.stop();
      }
```

```
        rightMotor.setSpeed(pwr);
        if (pwr < 0) {
            rightMotor.backward();
        } else if (pwr > 0) {
            rightMotor.forward();
        } else {
            rightMotor.stop();
        }
    }

    /**
     * getAngle returns the average motor angle of the left and right motors
     *
     * @return A double of the average motor angle of the left and right motors in degrees.
     */
    public double getAngle()
    {
        return ((double) leftMotor.getTachoCount() +
                (double) rightMotor.getTachoCount()) / 2.0;
    }

    /**
     * getAngle returns the average motor velocity of the left and right motors
     *
     * @return a double of the average motor velocity of the left and right motors in
     *         degrees.
     */
    public double getAngleVelocity()
    {
        return ((double) leftMotor.getActualSpeed() +
                (double) rightMotor.getActualSpeed()) / 2.0;
    }

    /**
     * reset the motors tacho count
     */
    public void resetMotors()
    {
        leftMotor.resetTachoCount();
        rightMotor.resetTachoCount();
    }

    /**
     * stop both motors from rotating
     */
    public void stop()
    {
        leftMotor.stop();
        rightMotor.stop();
    }
}
```

A.5 GyroscopeSensor.java Source Code

```
/* -*- tab-width: 2; indent-tabs-mode: nil; c-basic-offset: 2 -*- */
import lejos.nxt.*;

/**
 * This class is designed to work with the HiTechnic Gyrosensor. It has methods to calculate
 * the gyro offset, and find the gyro's angle and velocity. It has been modified by the
 * original programmers Bent Bisballe Nyeng, Kasper Sohn and Johnny Rieper
 *
 * @author Steven Jan Witzand
 * @version August 2009
 */
public class GyroscopeSensor
{
   private SensorPort port;
   private double angle = 0.0;
   private int lastGetAngleTime = 0;
   private double lastOffset = 0;
   private final double a = 0.999999;// Weight of older offset values.

   /**
    * The GyroscopeSensor constructor.
    *
    * @param port The NXT Sensor port the gyro sensor is connected to.
    */
   public GyroscopeSensor(SensorPort port)
   {
      this.port = port;
      calcOffset();
   }

   /**
    * Calculates the offset specific to the HiTechnic gyro sensor. Needs to read the gyro
    * sensor in a stationary position.
    */
   public void calcOffset()
   {
      lastOffset = 0;
      double offsetTotal = 0;
      LCD.drawString("Calibrating␣Gyro", 0, 2);
      for (int i = 0; i < 50; i++) {
         offsetTotal += (double) port.readValue();
         try {
            Thread.sleep(4);
         } catch (InterruptedException e) {
         }
      }
      while (!Button.ENTER.isPressed()) {
         lastOffset = Math.ceil(offsetTotal / 50) + 1;
         LCD.drawString("Calibration␣Done", 0, 4);
         LCD.drawString("offset:␣" + lastOffset, 2, 5);
         LCD.drawString("Press␣Enter", 1, 6);
      }
      try {
         Thread.sleep(500);
      } catch (InterruptedException e) {
      }
   }

   /**
    * Gets the angle offset of the gyro. Uses a recursive filter to attempt to account for
    * the gyro drift.
    *
    * @return A double containing the current gyro offset value.
    */
   private double getAngleOffset()
   {
      double offset = lastOffset * a + (1.0 - a) * (double) port.readValue();
```

```
        lastOffset = offset;
        return offset;
    }

    /**
     * Get the angle velocity of the gyro sensor.
     *
     * @return A double containing the angular velocity of the gyro sensor in degrees per
     *         second
     */
    public double getAngleVelocity()
    {
        double offset = getAngleOffset();
        return (double) port.readValue() - offset;
    }

    /**
     * Get the calculated gyro angle (angular velocity integrated over time).
     *
     * @return The angle in degrees.
     */
    public double getAngle()
    {
        int now = (int) System.currentTimeMillis();
        int delta_t = now - lastGetAngleTime;

        // Make sure we only add to the sum when there has actually
        // been a previous call (delta_t == now if its the first call).
        if (delta_t != now) {
            angle += getAngleVelocity() * ((double) delta_t / 1000.0);
        }
        lastGetAngleTime = now;

        return angle;
    }

    /**
     * Reset the gyro angle
     */
    public void resetGyro()
    {
        angle = 0.0;
        lastGetAngleTime = 0;
    }

}
```

A.6 CtrlParam.java Source Code

```
/**
 * This class is utilised to set and get common parameters between the Balance and
 * Behavioural threads. It is used to offset the motors power levels and the tilt offset of
 * the robot.
 *
 * @author Steven Witzand
 * @version August 2009
 */
class CtrlParam
{
   private double offsetLeft = 0.0;
   private double offsetRight = 0.0;
   private double tiltAngle = 0.0;
   private int driveState = 0;
   private double damp = 0.0;
   private boolean upright = false;

   /**
    * Set the GELways LEFT motor offset.
    *
    * @param offset
    *           a double used to set the left motor offset in degrees per second.
    */
   public synchronized void setLeftMotorOffset(double offset)
   {
      this.offsetLeft = offset;
   }

   /**
    * Set the GELways RIGHT motor offset.
    *
    * @param offset
    *           a double used to set the right motor offset in degrees per second.
    */
   public synchronized void setRightMotorOffset(double offset)
   {
      this.offsetRight = offset;
   }

   /**
    * Set the GELways LEFT motor offset.
    *
    * @return A double of the left motor offset in degrees per second.
    */
   public synchronized double leftMotorOffset()
   {
      return this.offsetLeft;
   }

   /**
    * Set the GELways RIGHT motor offset.
    *
    * @return A double of the right motor offset in degrees per second.
    */
   public synchronized double rightMotorOffset()
   {
      return this.offsetRight;
   }

   /**
    * Set the tilt motor angle offset. The method works incrementally since it is the nature
    * off the PID controller to remove the offset.
    *
    * @param angle
    *           A double of the tilt angle offset in degrees.
    */
   public synchronized void setTiltAngle(double angle)
```

```
    {
        this.tiltAngle += angle;
    }

    /**
     * Get the tilt motor angle offset.
     *
     * @return A double of the tilt angle offset in degrees.
     */
    public synchronized double tiltAngle()
    {
        return this.tiltAngle;
    }

    /**
     * Resets the tilt motor angle back to zero.
     */
    public void resetTiltAngle()
    {
        this.tiltAngle = 0.0;
    }

    /**
     * Set the current drive state of the GELway. 0 = Forwards, 1 = Backwards, 2 = Stationary
     *
     * @param state
     *            current drive state of the GELway
     */
    public void setDriveState(int state)
    {
        this.driveState = state;
    }
    /**
     * Returns the current drive state of the GELway
     *
     * @return current drive state of the GELway
     */
    public synchronized int getDriveState()
    {
        return this.driveState;
    }
    /**
     * Returns the current upright state of the GELway
     *
     * @return current upright state of the GELway
     */
    public void setUpright(boolean state)
    {
        this.upright = state;
    }
    public synchronized boolean getUpright()
    {
        return this.upright;
    }
    /**
     * Testing methods used to slow the initial accelleration of the GELway
     */
    public void setDamp(double weight)
    {
        this.damp += weight;
    }
    public void resetDamp()
    {
        this.damp = 0.0;
    }
    public synchronized double getDamp()
    {
        return this.damp;
    }
}
```

A.7 MotorDirection.java Source Code

```
/**
 * A class which handles controlling the GELway directional movements. Works by altering
 * parameters in the CtrlParam class which are called by the BalanceController thread.
 *
 * @author Steven Jan Witzand
 * @version August 2009
 */
public class MotorDirection
{
   private CtrlParam ctrl;
   private final double turnPower = 200; // Speed at which the GELway rotates
   private final double tiltPower = 3; // Speed of moving forwards and backwards

   /**
    * MotorDirection constructor.
    *
    * @param ctrl
    *           The motor control parameters.
    */
   public MotorDirection(CtrlParam ctrl)
   {
      this.ctrl = ctrl;
   }

   /**
    * Moves the GELway forward for a specified period of time. Works by incrementally
    * increasing the motor tilt angle offset.
    *
    * @param period
    *           An integer number specifying the length to move the GELway forward.
    */
   public void forward(int period)
   {
      ctrl.setLeftMotorOffset(0);
      ctrl.setRightMotorOffset(0);
      for (int time = 0; time < period; time += 10) {
         try {
            Thread.sleep(10);
         } catch (InterruptedException e) {
         }
         ctrl.setTiltAngle(tiltPower);
      }
   }

   /**
    * Moves the GELway backward for a specified period of time. Works by incrementally
    * decreasing the motor tilt angle offset.
    *
    * @param period
    *           An integer number specifying the length to move the GELway backward.
    */
   public void backward(int period)
   {
      ctrl.setLeftMotorOffset(0);
      ctrl.setRightMotorOffset(0);

      for (int time = 0; time < period; time += 10) {
         try {
            Thread.sleep(10);
         } catch (InterruptedException e) {
         }
         ctrl.setTiltAngle(-1.5 * tiltPower);
      }
   }

   /**
    * Moves the GELway in a direction for a specified period of time and power. Works by
```

```
     * incrementally increasing the motor tilt angle offset.
     *
     * @param period
     *           An integer number specifying the length to move the GELway forward.
     */
    public void move(int period, double tiltDir)
    {
        ctrl.setLeftMotorOffset(0);
        ctrl.setRightMotorOffset(0);
        for (int time = 0; time < period; time += 10) {
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
            }
            ctrl.setTiltAngle(tiltDir);
        }
    }

    /**
     * Rotate the GELway left indefinitely until a forward or backward movement is given.
     *
     * @param period
     *           An integer to delay other commands being sent to the GELway
     */
    public void left(int period)
    {
        ctrl.setLeftMotorOffset(-turnPower);
        ctrl.setRightMotorOffset(turnPower);
        delay(period);
    }

    /**
     * Rotate the GELway right indefinitely until a forward or backward movement is given.
     *
     * @param period
     *           An integer to delay other commands being sent to the GELway
     */
    public void right(int period)
    {
        ctrl.setLeftMotorOffset(turnPower);
        ctrl.setRightMotorOffset(-turnPower);
        delay(period);
    }

    /**
     * Instruct GELway to stop, and stay stopped for a given period of time.
     *
     * @param period
     *           An integer containing the number of millisecond to pause before returning.
     */
    public void stop(int period)
    {
        ctrl.setLeftMotorOffset(0);
        ctrl.setRightMotorOffset(0);
        delay(period);
    }

    /**
     * Make the programming sleep for a specified period of time.
     *
     * @param time
     *           An integer containing the number of milliseconds to sleep.
     */
    public void delay(int time)
    {
        try {
            Thread.sleep(time);
        } catch (Exception e) {}
    }
}
```

A.8 GELwayDriver.java Source Code

```
import lejos.subsumption.*;
import lejos.nxt.*;

/**
 * A behaviour which allows the GELway to receive Bluetooth commands to turn them into
 * actions in the robot. Note that this behaviour requires an active Bluetooth connection.
 *
 * @author Steven Jan Witzand
 * @version August 2009
 */
public class GELwayDriver implements Behavior
{

   // KeyCodes:
   private static final int directionLeft = 4; // left
   private static final int directionRight = 6; // right
   private static final int directionForward = 2; // up
   private static final int directionBackward = 8; // down
   private static final int holdPosition = 5; // down

   BluetoothReader br;
   MotorDirection mv;
   CtrlParam ctrl;

   /**
    * GELwayDriver constructor.
    *
    * @param ctrl
    *           The motor control parameters.
    * @param br
    *           An active Bluetooth connection.
    */
   public GELwayDriver(CtrlParam ctrl, BluetoothReader br)
   {
      this.ctrl = ctrl;
      mv = new MotorDirection(ctrl);
      this.br = br;
   }

   /**
    * Trigger for the Behaviour. Note that this always reutrns true since it is the lowest
    * level behaviour.
    */
   public boolean takeControl()
   {
      return true;
   }
   /**
    * No suppression required.
    */
   public void suppress()
   {

   }
   /**
    * Action method which converts a sent command into a movement action in the robot.
    */
   public void action()
   {
      if (br.getNewDir()) {
         LCD.clear();
         int number = br.getDir();
         switch (number)
         {
            case directionLeft:
               LCD.drawString("LEFT", 5, 5);
               mv.left(200);
```

```
                break;

            case directionRight:
                LCD.drawString("RIGHT", 5, 5);
                mv.right(200);
                break;

            case directionForward:
                LCD.drawString("FORWARD", 3, 5);
                ctrl.setLeftMotorOffset(0);
                ctrl.setRightMotorOffset(0);
                ctrl.resetDamp();
                mv.forward(100);
                // ctrl.setDriveState(1);
                break;

            case directionBackward:
                LCD.drawString("BACKWARD", 3, 5);
                ctrl.setLeftMotorOffset(0);
                ctrl.setRightMotorOffset(0);
                ctrl.resetDamp();
                mv.backward(100);
                // ctrl.setDriveState(2);
                break;

            case holdPosition:
                LCD.drawString("Stay", 3, 5);
                // mv.stop(200);
                ctrl.setDriveState(0);
                break;

            default:
                break;
        }
    }
  }
}
```

A.9 GELwayFollower.java Source Code

```
import lejos.subsumption.*;
import lejos.util.Datalogger;
import lejos.nxt.*;
import lejos.nxt.addon.EOPD;
/**
 * A behaviour which uses a distance controller to keep the GELway at a set distance from a
 * given obstacle.
 *
 * @author Steven Jan Witzand
 * @version August 2009
 */
public class GELwayFollower implements Behavior
{
   // KeyCodes used the drive the GELway:
   private static final int directionLeft = 4; // left
   private static final int directionRight = 6; // right
   private static final int directionForward = 2; // up
   private static final int directionBackward = 8; // down
   private static final int holdPosition = 5; // down
   MotorDirection mv;
   CtrlParam ctrl;
   UltrasonicSensor us = new UltrasonicSensor(SensorPort.S1);
   private final double K = 0.1;
   double error = 0.0;
   double setDist = 20.0;

   /**
    * GELwayFollower constructor.
    *
    * @param ctrl
    *           The motor control parameters.
    */
   public GELwayFollower(CtrlParam ctrl)
   {
      this.ctrl = ctrl;
      mv = new MotorDirection(ctrl);
   }

   /**
    * The takeControl method always returns true as this is the lowest behaviour
    */
   public boolean takeControl() { return true }
   /**
    * No suppression required.
    */
   public void suppress(){}
   /**
    * Action method which checks the current distance from a given object and corrects this
    * by sending a motor angle reference to the controller.
    */
   public void action()
   {
      if (ctrl.getUpright()) {
         // Measured Distance
         double dist = (double) us.getDistance();
         error = dist - setDist;
         // Power sent to the motors
         double tiltPower = (K * error);
         // Ctrl Offset Caps
         if (tiltPower > 0.5) tiltPower = 0.5;
         if (tiltPower < -0.5) tiltPower = -0.5;
         ctrl.setTiltAngle(tiltPower);
      }
      mv.delay(1);
   }
}
```

A.10 DetectObstacle.java Source Code

```
import lejos.subsumption.*;
import lejos.subsumption.Behavior;
import lejos.nxt.*;

/**
 * A behaviour which allows the GELway to avoid obstacles using the Ultrasonic Sensor.
 *
 * @author Steven Jan Witzand
 * @version August 2009
 */
public class DetectObstacle implements Behavior
{
   CtrlParam ctrl = new CtrlParam();
   MotorDirection mv;
   UltrasonicSensor us = new UltrasonicSensor(SensorPort.S1);

   /**
    * DetectObstacle constructor.
    *
    * @param ctrl
    *           The motor control parameters.
    */
   public DetectObstacle(CtrlParam ctrl)
   {
      this.ctrl = ctrl;
      mv = new MotorDirection(ctrl);
   }

   /**
    * Trigger for the Behaviour. This trigger is actioned when a distance less than 25cm is
    * detected.
    */
   public boolean takeControl()
   {
      return (us.getDistance() < 25);
   }

   /**
    * No suppression required.
    */
   public void suppress()
   {
   }

   /**
    * Action method which stops the robots current movements, reverses the robot and turns
    * 180 degrees.
    */
   public void action()
   {
      ctrl.setLeftMotorOffset(0);
      ctrl.setRightMotorOffset(0);
      ctrl.setDriveState(0);
      mv.backward(200);
      mv.right(1800);
      mv.stop(1000);
      mv.backward(100);
   }

}
```

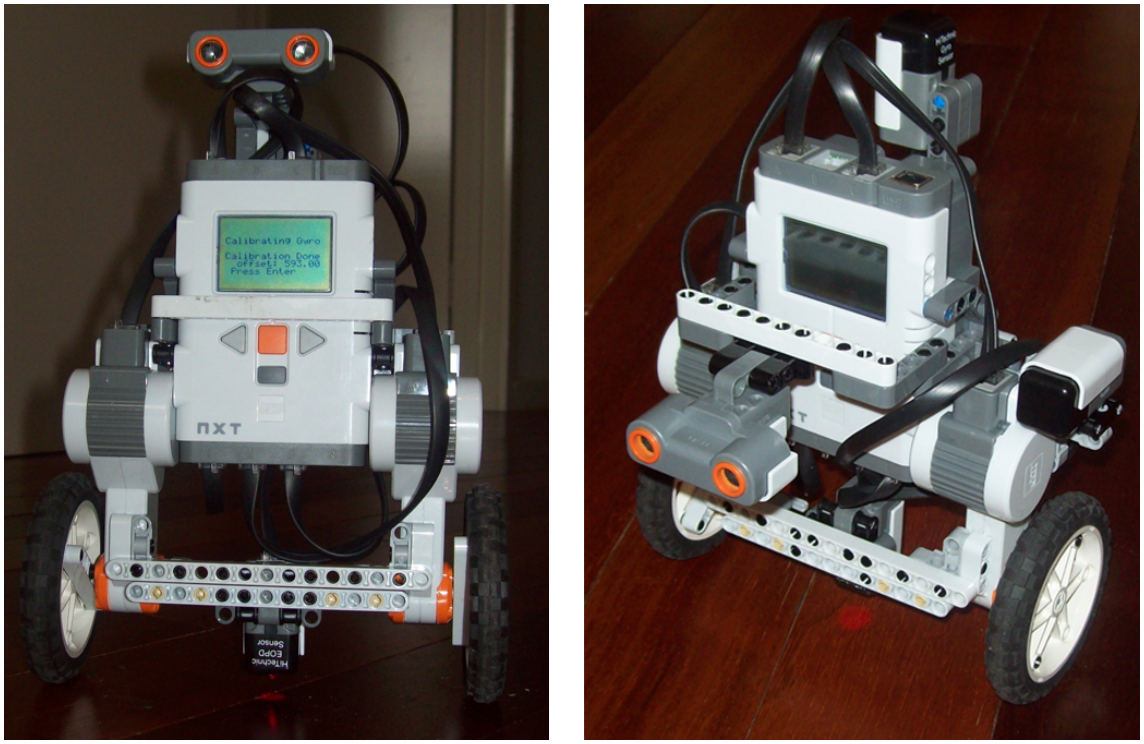
A.11 BluetoothReader.java Source Code

```
import java.io.*;
import lejos.nxt.comm.*;
/**
 * This thread constantly listens to Bluetooth commands sent to the GELway
 *
 * @author Steven Jan Witzand
 * @version August 2009
 */
public class BluetoothReader extends Thread
{
   DataInputStream istream;
   DataOutputStream ostream;
   private int dir;
   private boolean newDir = true;
   /**
    * BluetoothReader constructor.
    *
    * @param conn The Bluetooth Connection set up with the GELway
    */
   public BluetoothReader(BTConnection conn)
   {
      istream = conn.openDataInputStream();
      ostream = conn.openDataOutputStream();
   }
   /**
    * BluetoothReader thread which constantly runs waiting for new commands
    */
   public void run()
   {
      dir = 0;
      while (true) {
         while (true) {
            try {
               newDir = false;
               dir = istream.readInt();
               newDir = true;
               try {Thread.sleep(100);} catch (InterruptedException e) {}
               if (dir == -1) break;
            } catch (IOException e) {}
         }
      }
   }
   /**
    * Reutrns the current commands send over Bluetooth
    * @return the current command sent over Bluetooth
    */
   public int getDir() { return dir;}
   /**
    * Used to check if a new commands has been sent.
    *
    * @return is a current commands has been sent over Bluetooth
    */
   public boolean getNewDir(){return newDir;}

   /**
    * Used to send a commands to the Bluetooth device the robot is connected with.
    *
    * @param command command to be sent to connected Bluetooth device
    */
   public void sendCommand(int command)
   {
      try {
         ostream.writeInt(command);
         ostream.flush();
      } catch (IOException e) {}
   }
}
```

A.12 GELway Building Instructions

This document contains the building instructions for the GELway Follower and GELway Leader robots. Note that the build design is largely based off Yorihisa Yamamoto's NXTway-GS building designs with modifications made to suit this project [43]. All parts used come from the LEGO Mindstorms standard kit with the exception of the Gyroscope and EOPD sensor which come from HiTechnic.

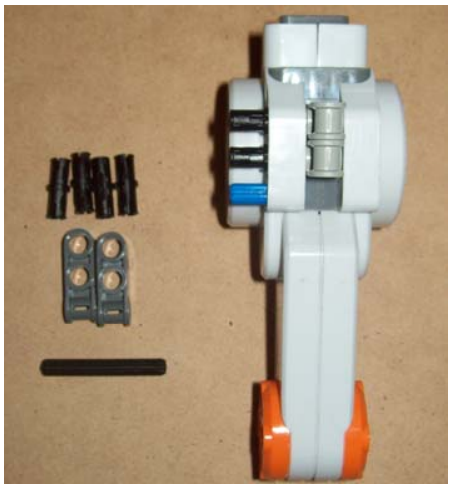


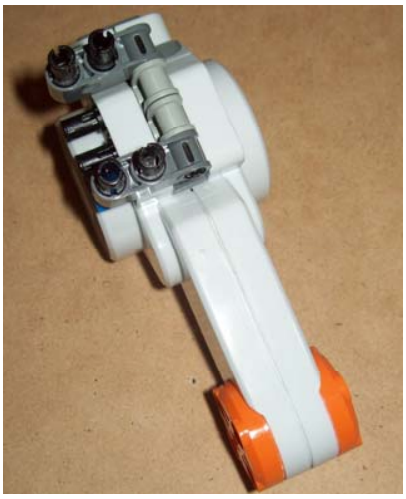
GELway Leader pictured (left) and the GELway Follower Robot pictured (right)

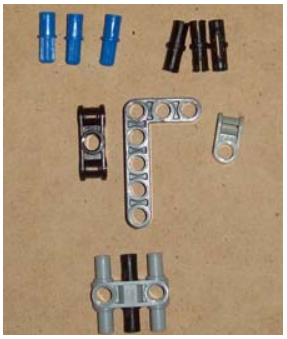
GELway Building Instructions

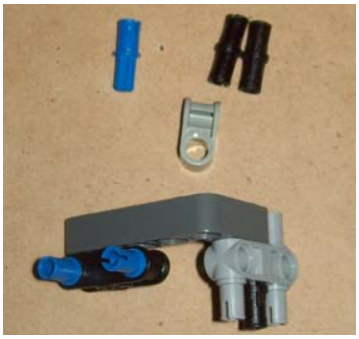
Modified version of the NXTway-GS building instructions

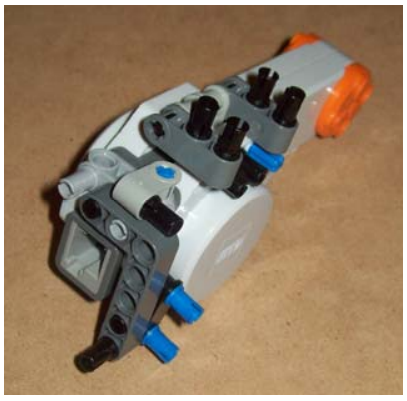
Step 1—Building the Motor Connection

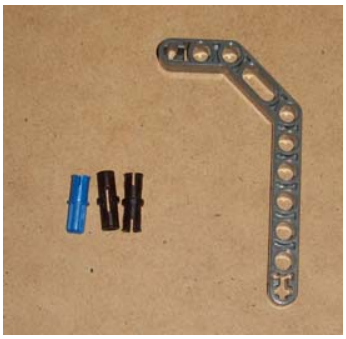


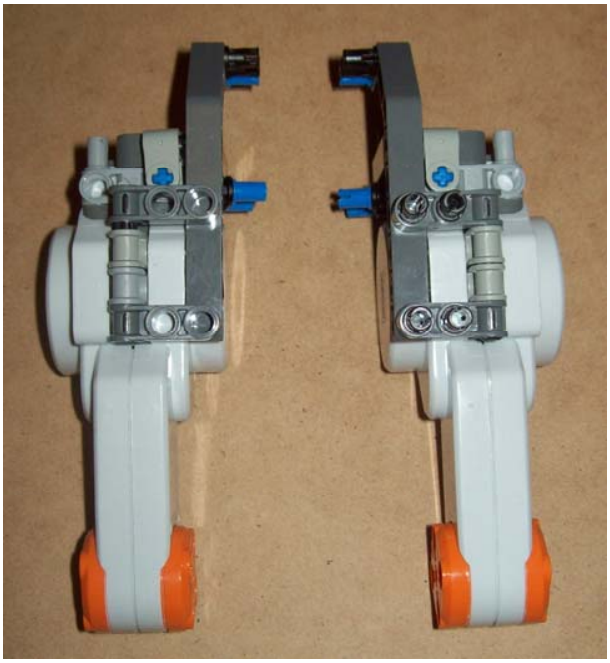




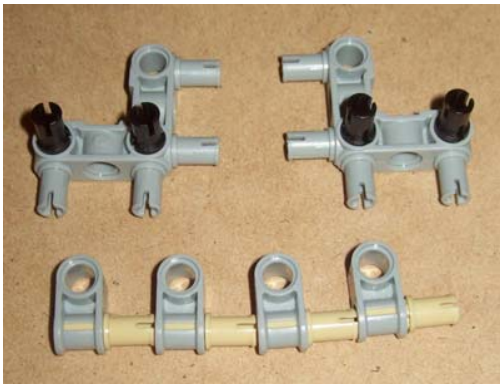


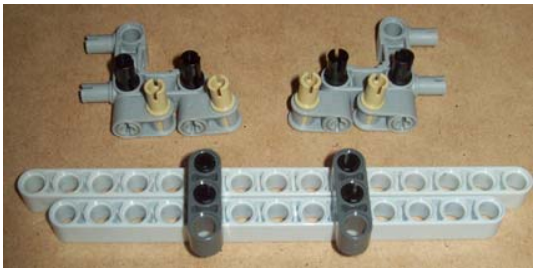


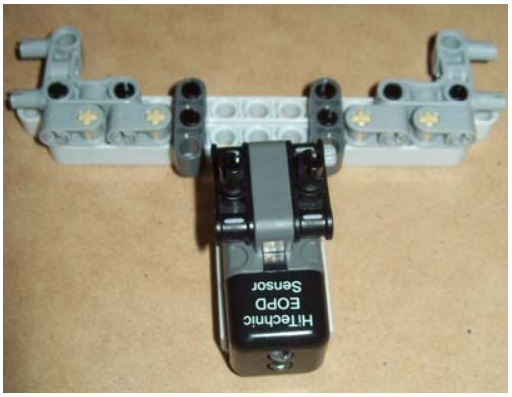


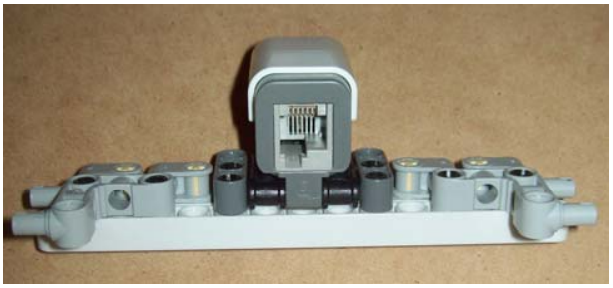


Step 3—Building the Upright Detector Frame



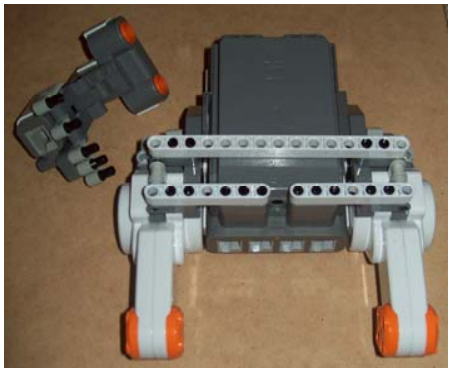


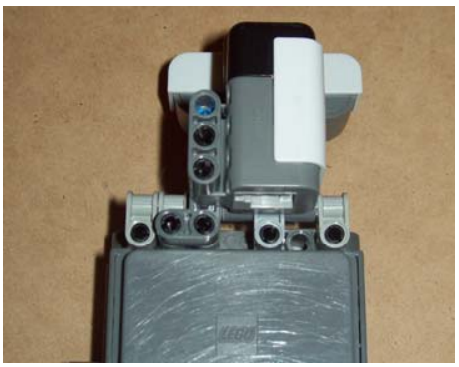




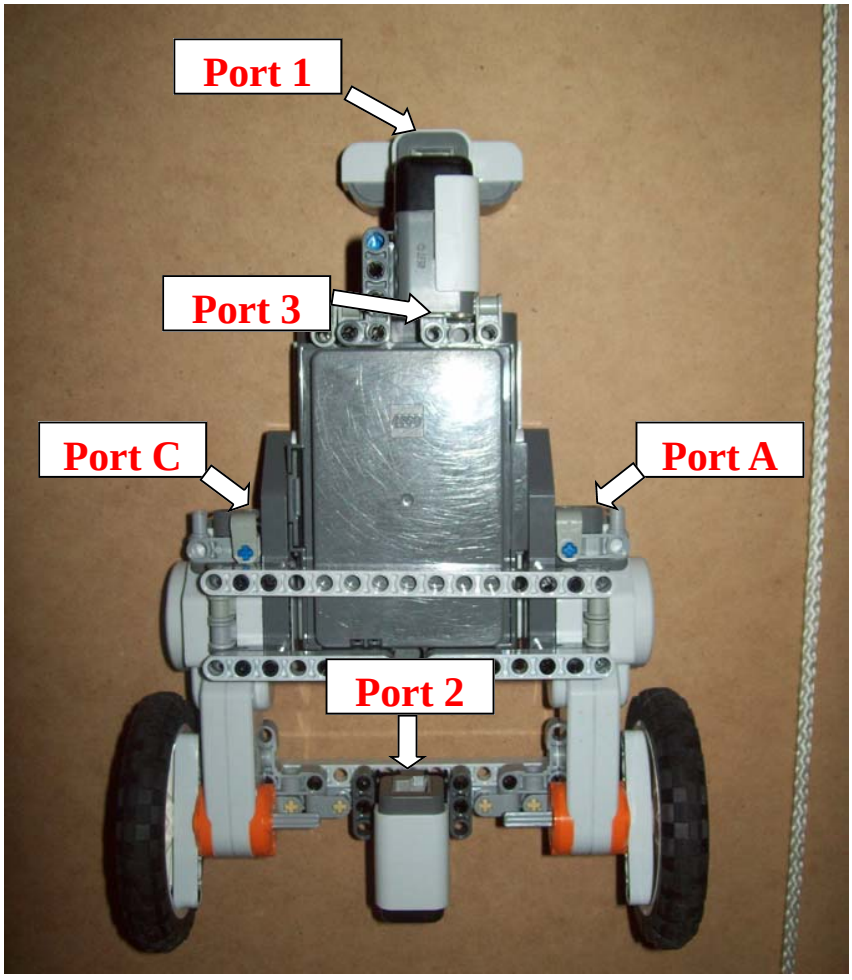
Step 3—Connecting the Motors

Leading Robot—Head Connection

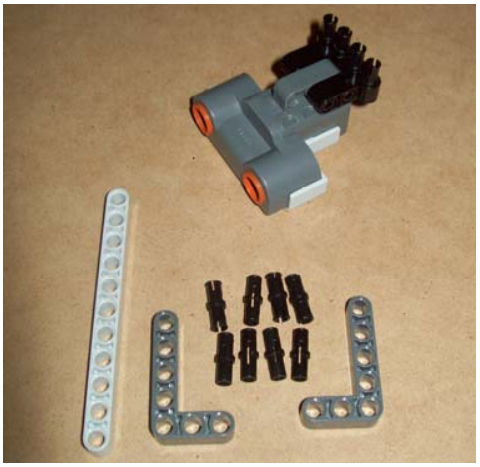


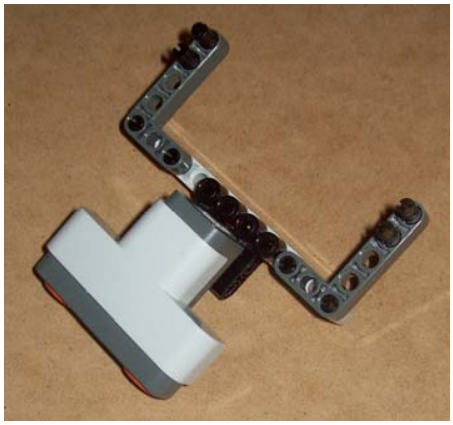


Leading Robot—Complete with Port Connections

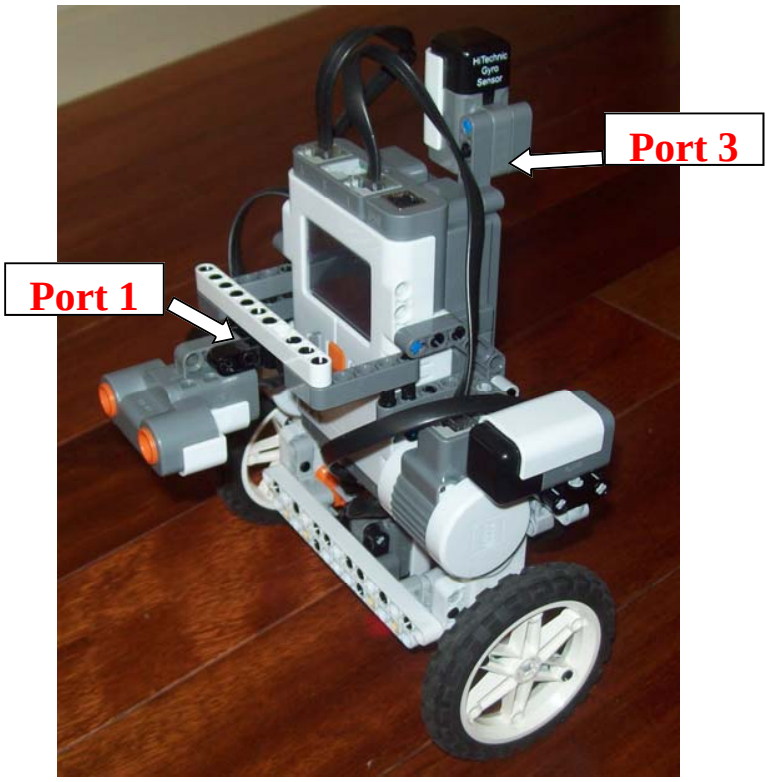


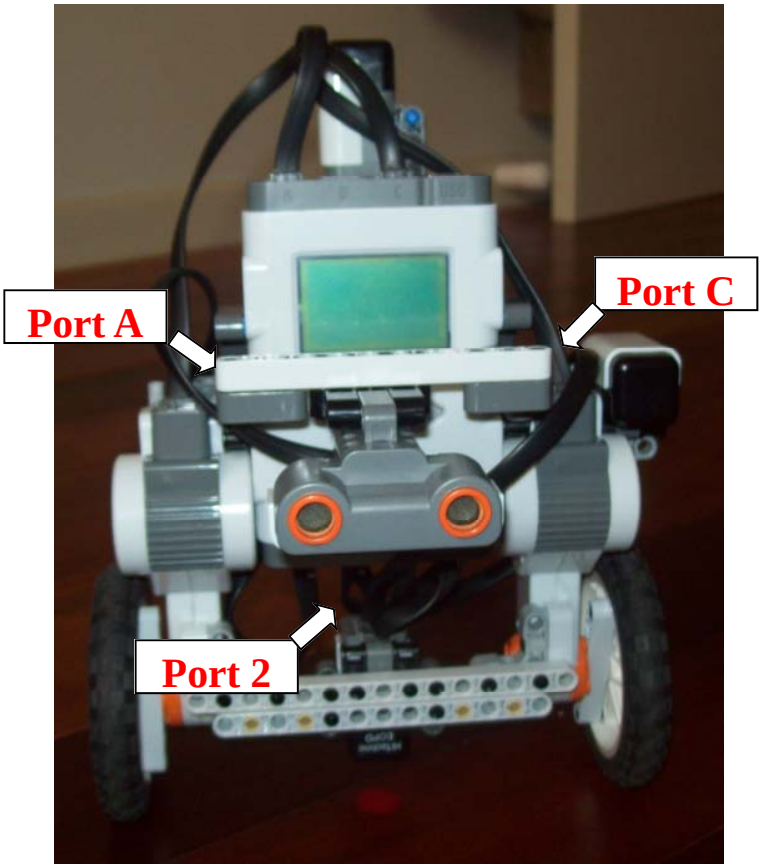
Follower Robot—Head Connection



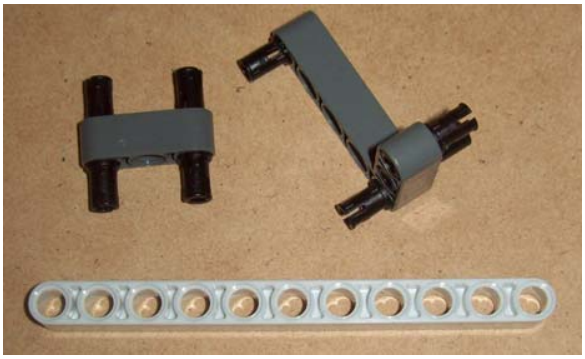


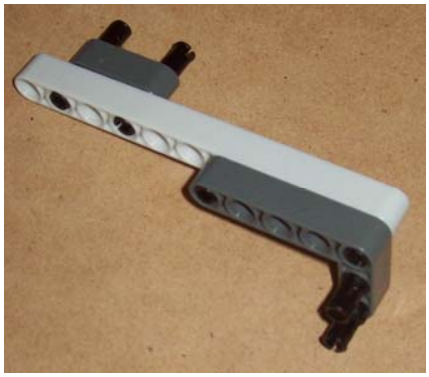
Follower Robot—Complete with Port Connections

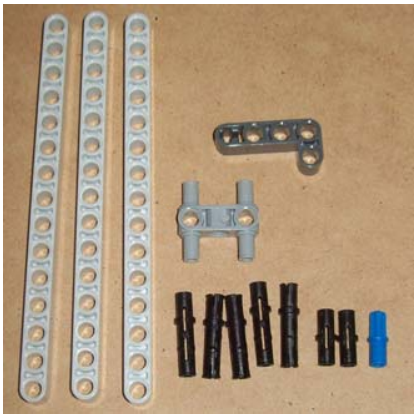


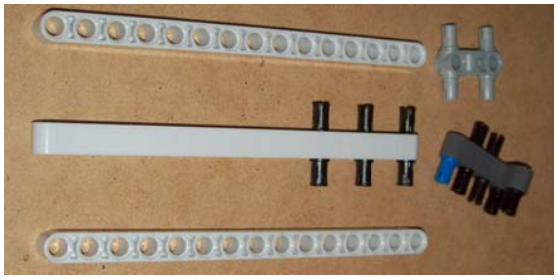


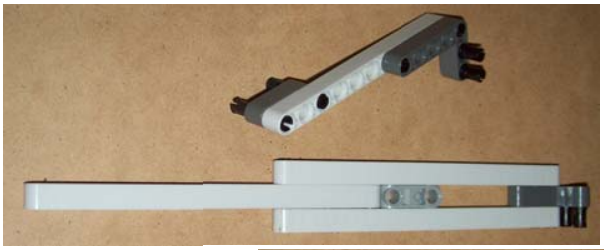
Weight Carrying Attachment—Sturdy Arms












NXT


Includes link to ONLINE ROBOT SIMULATOR
Robot Programming
A Practical Guide to Behavior-Based Robotics
JOSEPH L. JONES
Robotic Simulator by Daniel Roth
TAB

Additional Sensor Attachments

A.13 GELwayController.m Source Code

```
% Modified by Steven Witzand  UNSW@ADFA 2009

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Program: GELway_Controller.m
% Written for Coordinated LEGO Segway Thesis
% This program is a modified version of the Matlab files provided by the
% creaters of the NXTway-GS balancing robot. This file is used to establish
% the plant for the simulink models.
%
% SBLT Steven Witand 2009
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% GELway Parameters and State-Space Matrix Calculation
% Modified version of the NXTway-GS Controller files

% Physical Constant
g = 9.81;                       % gravity acceleration [m/sec^2]

% GELway Parameters
m = 0.03;                       % wheel weight [kg]
R = 0.04;                       % wheel radius [m]
Jw = m * R^2 / 2;               % wheel inertia moment [kgm^2]
M = 0.6;                        % body weight [kg]
W = 0.14;                       % body width [m]
D = 0.04;                       % body depth [m]
H = 0.144;                      % body height [m]
L = H / 2;                      % distance of the CoM from the wheel axle [m]
Jpsi = M * L^2 / 3;             % body pitch inertia moment [kgm^2]
Jphi = M * (W^2 + D^2) / 12;    % body yaw inertia moment [kgm^2]
fm = 0.0022;                    % friction coefficient between body & DC motor
fw = 0;                         % friction coefficient between wheel & floor

% DC Motor Parameters
Jm = 1e-5;                      % DC motor inertia moment [kgm^2]
Rm = 6.69;                      % DC motor resistance [ ]
Kb = 0.468;                     % DC motor back EMF constant [Vsec/rad]
Kt = 0.317;                     % DC motor torque constant [Nm/A]
n = 1;                          % gear ratio

% GELway State-Space Matrix Calculation
alpha = n * Kt / Rm;
beta = n * Kt * Kb / Rm + fm;
tmp = beta + fw;

E_11 = (2 * m + M) * R^2 + 2 * Jw + 2 * n^2 * Jm;
E_12 = M * L * R - 2 * n^2 * Jm;
E_22 = M * L^2 + Jpsi + 2 * n^2 * Jm;
detE = E_11 * E_22 - E_12^2;

A1_32 = -g * M * L * E_12 / detE;
A1_42 = g * M * L * E_11 / detE;
A1_33 = -2 * (tmp * E_22 + beta * E_12) / detE;
A1_43 = 2 * (tmp * E_12 + beta * E_11) / detE;
A1_34 = 2 * beta * (E_22 + E_12) / detE;
A1_44 = -2 * beta * (E_11 + E_12) / detE;
B1_3 = alpha * (E_22 + E_12) / detE;
B1_4 = -alpha * (E_11 + E_12) / detE;
A1 = [
        0 0 1 0
        0 0 0 1
        0 A1_32 A1_33 A1_34
        0 A1_42 A1_43 A1_44
        ];
B1 = [
        0 0
        0 0
        B1_3 B1_3
```

```
        B1_4 B1_4
        ];
C1 = eye(4);
D1 = zeros(4, 2);

% Controller Parameters

% Servo Gain Calculation using Optimal Regulator
A_BAR = [A1, zeros(4, 1); C1(1, :), 0];
B_BAR = [B1; 0, 0];
QQ = [
        1, 0,   0, 0, 0
        0, 6e5, 0, 0, 0
        0, 0,   1, 0, 0
        0, 0,   0, 1, 0
        0, 0,   0, 0, 4e2
        ];
RR = 1e3 * eye(2);
KK = lqr(A_BAR, B_BAR, QQ, RR);
k_f = KK(1, 1:4);                    % feedback gain
k_i = KK(1, 5);                      % integral gain

% suppress velocity gain because it fluctuates GELway
k_f(3) = k_f(3) * 0.85;
C_phi=[1 0];
C_theta=[1 0 0 0];
```

A.14 GELwayRun.m Source Code

```
% Written by Steven Witzand  UNSW@ADFA 2009

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Program: GELway.m
% Written for Coordinated LEGO Segway Thesis
% This program is used to simulate both the Marvin the Balancing robot and
% the NXTway-GS control systems. The MATLAB file creates the plants for the
% controller then calls the respective simulink models for each robot.
% Using the data from Simulink the maximum overshoot, delay time, rise time
% and settling time is calculated and displayed in the form of a plot.
%
% SBLT Steven Witand 2009
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clf, clc

% Set up the plant parameters
GELway_Controller;
% PID Parameters
P = 1;
I = 0.2;
D = 0.2;

% Simulation 1
sim('Marvin'); ptime = 20; % Marvin Simulation
%sim('NXTwayGS'); ptime = 10; NXTway-GS Simulation
t = GELout.time;
u = GELin.signals.values;
y = GELout.signals.values;
hold on
plot_char(ptime, t, u, y, 1);

% Simulation 2
P = 1.15;
I = 0.25;
D = 0.2;
sim('Marvin'); ptime = 20;
t = GELout.time;
u = GELin.signals.values;
y = GELout.signals.values;
plot_char(ptime, t, u, y, 2);

% Label Plot
xlabel('t (sec)','fontsize',14)
ylabel('Motor Angle (deg)','fontsize',14)
title('Response of Controller to Motor Angle Disturbance','fontsize',16)
```

A.15 GELwayTransferFunction.m Source Code

```
% Written by Steven Witzand UNSW@ADFA 2009

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Program: GELway.m
% Written for Coordinated LEGO Segway Thesis
% This program builds the transfer function of the GELway. It is used to
% create Bode and Nyquist plots
%
% SBLT Steven Witand 2009
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clf
% Set up the state-space variables for the GELway
GELway_Controller;
% Get the state-space model for only one wheel (both inputs are the same
sys=ss(A1,2*B1(:,1),eye(4),zeros(4,1))
s = tf('s');

% Marvin's PID Parameters
P = 1;
I = 0.2;
D = 0.2;
pid = P + I/s + D*s/(1/1000*s+1);
sys1 = sys*pid;
sys2 = feedback(sys1,k_f);
sys3 = C_theta*(sys2) * k_f(1);
% Loop-gain transfer function
lgain2 = k_f*sys1;
hold on
%subplot(2,1,1)
%margin(lgain)
nyquist(lgain2)
pole(lgain2)
zero(lgain2)

% GELway's PID Parameters
P = 1.15;
I = 0.25;
D = 0.2;
pid = P + I/s + D*s/(1/1000*s+1);
%subplot(2,1,2)
sys1 = sys*pid;
sys2 = feedback(sys1,k_f);
sys3 = C_theta*(sys2) * k_f(1);
% Loop-gain transfer function
lgain = k_f*sys1;
%margin(lgain)
nyquist(lgain)
pole(lgain)
axis([-3 15 -10 10])
zero(lgain)
legend('Marvin','GELway')
```

A.16 GELwayRemote.java Source Code

```
import processing.core.*;
import processing.bluetooth.*;
public class GELwayRemote extends PMIDlet{
/*
 * GELwayRemote
 * This program is used to connect a mobile device to the GELway robot. The mobile
 * is used as a remote control and controls the GELway's movements. Each button on
 * the mobile phone sends a integer command which can is received by the GELway. The
 * mobile phone does not actually drive the GELway around, it just sends an integer
 * value. The GELway uses the received integers to handle events such as moving
 * around and resetting the robot.
 *
 * @Author Steven Witzand
 * @Version 0.2
 * @Note This program is based off Pedro Miguel's NXTSymbian v0.1 and has been
 * modified to suit the needs of the GELway robot.
 */


final String SOFTKEY_HELP = "Help";
final String SOFTKEY_BAT   = "Battery";
final String SOFTKEY_BACK   = "Back";

final int STATE_START = 0;
final int STATE_FIND = 1;
final int STATE_CONNECTED = 2;
final int STATE_HELP = 3;

int state;

Bluetooth bt;
Service[] services;
Client cl;

String msg;
PFont font;

/**
 * Setup method used load the font and the type of Bluetooth connection
 */
public void setup()
{
  font = loadFont();
  textFont(font);
  bt = new Bluetooth(this, Bluetooth.UUID_SERIALPORT);
  state = STATE_START;
}

/**
 * Called when the program is exiting, stopping the Bluetooth connection.
 */
public void destroy()
{
  background(0,226,167);
  fill(0);
  text("Exiting...", 2, 2, width - 4, height - 4);
  bt.stop();
  pause(2000);
}

/**
 * Setup method used load the font and the type of Bluetooth connection
 */
public void draw()
{
  background(0,226,167); // background colour of the program
  // start up screen of the program
```

```
  if (state == STATE_START)
  {
    fill(0);
    PImage b;
    b = loadImage("GELway2.png");
    image(b, (width/2)-(b.width/2), 5);
    textAlign(CENTER);
    text("GELwayRemote␣v0.2\nModified␣By␣Witzand\n\nPress␣any␣key␣to␣search␣for...
␣␣␣␣␣␣␣␣␣␣...active␣Bluetooth␣devices.", 2, b.height + 10, width - 4, height - 4);
  }
  // state used to search for active Bluetooth devices. Also used to connect to
  // establish a Bluetooth connection.
  else if (state == STATE_FIND)
  {
    fill(0);
    textAlign(LEFT);
    if (services == null) {text("Looking␣for␣NXT...\n\n" + msg, 2, 2,...
          ...width-4, height-4);}
    else
    {
      String msg_aux = "Choose␣NXT␣port:\n";
      for (int i = 0; i < length(services); i++)
        msg_aux += i + ".␣" + services[i].device.name + "\n";
      text(msg_aux, 2, 2, width-4, height-4);
    }
  }
  // Screen used to display what current command has been sent to the GELway.
  else if (state == STATE_CONNECTED)
  {
    noFill();
    textAlign(CENTER);
    text("GELway␣Remote\n\nPress␣left␣softkey␣for␣Commands\n\n\n" + msg, 2, 2,...
           ...width-4, height-4);
  }
  // State which describes what each mobile phone button does.
  else if (state == STATE_HELP)
  {
    fill(0);
    textAlign(CENTER);
    text("Commands:\n2:␣Forward\n4:␣Left\n6:␣Right\n8:␣Reverse\n\n1:...
␣␣␣␣␣␣␣␣␣␣...Reset␣GELway\n\nPress␣any␣key␣to␣return", 2, 2, width-4, height-4);
  }
}

/**
 * This method handle searching for, connection to, and handling of the Bluetooth
 * connection.
 * @Param library type of library sent to the method, expecting to be Bluetooth
 *        library.
 * @Param event type of Bluetooth event
 * @Param data the Bluetooth datatype, holding information such as Bluetooth name
 *        and address
 */
public void libraryEvent(Object library, int event, Object data) {
  if (library == bt) {
    switch (event) {
    case Bluetooth.EVENT_DISCOVER_DEVICE:
      msg = "Device␣found:␣" + ((Device) data).address;
      break;
    case Bluetooth.EVENT_DISCOVER_DEVICE_COMPLETED:
      msg = "Found␣" + length((Device[]) data) + "␣devices...
␣␣␣␣␣␣␣␣␣␣␣␣␣␣...\nSearching␣for␣serial␣port␣service...";
      break;
    case Bluetooth.EVENT_DISCOVER_SERVICE:
      msg = "Found␣serial␣port␣on␣" + ((Service[]) data)[0].device.address;
      break;
    case Bluetooth.EVENT_DISCOVER_SERVICE_COMPLETED:
      services = (Service[]) data;
      msg = "Search␣complete.␣Pick␣one.";
      break;
```

```
    case Bluetooth.EVENT_CLIENT_CONNECTED:
      cl = (Client) data;
      msg = "Client Connected!?";
      break;
    }
  }
}

/**
 * This method brings up the help screen and returns back to the main screen.
 * @Param label string which is used to select states for the program.
 */
public void softkeyPressed(String label)
{
  if (label.equals(SOFTKEY_HELP))
  {
    state = STATE_HELP;
    softkey(SOFTKEY_BACK);
  }
  else if(label.equals(SOFTKEY_BAT)) {}
    else if(label.equals(SOFTKEY_BACK))
    {
    state = STATE_CONNECTED;
    softkey(SOFTKEY_HELP);
  }
}

/**
 * This method handles key events in the programs and alters the state according to
 * which buttons are pressed.
 */
public void keyPressed()
{
  if (state == STATE_START)
  {
    services = null;
    bt.find();
    state = STATE_FIND;
    msg = "";
  }
  else if(state == STATE_HELP)
  {
    state = STATE_CONNECTED;
  }
  else if (state == STATE_FIND)
  {
    if (services != null)
    {
      if ((key >= '0') && (key <= '9'))
      {
        int i = key - '0';
        if (i < length(services))
        {
          msg = "Connecting...";
          cl = services[i].connect();
          state = STATE_CONNECTED;
          softkey(SOFTKEY_HELP);
          msg = "Connected";
        }
      }
    }
  }
  else if (state == STATE_CONNECTED)
  {
    nxtkey(keyCode,key);
  }
}

/**
 * This method is called when a button is pressed by the mobile and sends the
```

```
 * corresponding integer
 * corresponding to what button was pressed.
 * @Param aKeyCode what joystick button was pressed
 * @Param akey what mobile button is pressed
 */
public void nxtkey(int akeyCode, int akey)
{
  // joystick buttons
  switch(akeyCode)
  {
    case UP:
      msg = "FWD";
      //nxt_send(fwd);
      nxt_send(2);
      break;
    case DOWN:
      msg = "RWD";
      nxt_send(8);
      break;
    case LEFT:
       msg = "Left";
      nxt_send(4);
      break;
    case RIGHT:
      msg = "Right";
      nxt_send(6);
      break;
    // mobile buttons pressed
    default:
      switch(akey)
      {
      case '1':
        msg = "Reset";
        nxt_send(1);
        break;
      case '2':
        msg = "FWD";
        nxt_send(2);
        break;
      case '3':
        msg = "...";
        nxt_send(3);
        break;
      case '4':
        msg = "Left";
        nxt_send(4);
        break;
      case '5':
        msg = "Stop";
        nxt_send(5);
        break;
      case '6':
        msg = "Right";
        nxt_send(6);
        break;
      case '7':
        nxt_send(7);
        break;
      case '8':
        msg = "RWD";
        nxt_send(8);
        break;
      case '9':
        msg = "Stop␣remotec";
        nxt_send(9);
        break;
      case '*':
        msg = "Trigger1";
        nxt_send(10);
        break;
```

```
      case '0':
        msg = "Trigger2";
        nxt_send(11);
        break;
      case '#':
        msg = "Trigger3";
        nxt_send(12);
        break;
      }
    }
}

/**
 * Delays the programs for a specified period of milliseconds
 * @Param millis how long the program is delayed, in milliseconds.
 */
public void pause(long millis) {
  try { Thread.sleep(millis);  }  catch( Exception e) { }
}

/**
 * This method sends and flushes commands to the GELway over Bluetooth
 * @Param command integer to be sent to the GELway
 */
public void nxt_send(int command){
  cl.writeInt(command);
  cl.flush();
}

/**
 * This method receives information from the GELway. Note this is currently not
 * used in the program, but was kept for future work.
 */
public byte[] nxt_rcv() {
  byte[] buffer = null;
  int length = -1;
  do {
    length = cl.read();
  } while (length < 0);
  int len_aux = cl.read();
  length = (0xFF & length) | ((0xFF & len_aux) << 8);
  buffer = new byte[length];
  cl.readBytes(buffer);
  return buffer;
}

}
```

A.17 PCController.java Source Code

```
/* -*- tab-width: 2; indent-tabs-mode: nil; c-basic-offset: 2 -*- */
import lejos.pc.comm.*;
import java.io.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/**
 * A simple PC GUI used to connect and send commands to the GELway.
 *
 * @author Steven Jan Witzand
 * @version August 2009
 */
class PCController extends JPanel
{
   // Defined movement commands
   private static final int directionLeft = 37; // left
   private static final int directionRight = 39; // right
   private static final int directionForward = 38; // up
   private static final int directionBackward = 40; // down
   private static final int reset = 10; // down

   static DataOutputStream dos;
   static NXTComm nxtComm;
   static JButton connect;
   static JButton endConnect;
   static JTextField nxt_name;
   static JTextField bluetooth;
   static JTextField text;

   public static void main(String[] args)
   {
      // Initiate bluetooth communication.
      // Create window

      JFrame frame = new JFrame("GELway Control");
      JPanel name = new JPanel();
      JPanel send = new JPanel();
      name.setLayout(new GridLayout(2, 2));
      send.setLayout(new GridLayout(1, 2));
      frame.setLayout(new GridLayout(4, 1));

      connect = new JButton("Connect NXT");
      endConnect = new JButton("End Connection");
      endConnect.setEnabled(false);
      text = new JTextField();
      // Master GELway
      nxt_name = new JTextField("GELway");
      bluetooth = new JTextField("00:16:53:09:85:B5");
      // Slave GELway
      // nxt_name = new JTextField("GELwayJR");
      // bluetooth = new JTextField("00:16:53:03:6A:A8");
      JLabel nxt_name1 = new JLabel("NXT Name:");
      JLabel blue_add = new JLabel("Bluetooth Address:");
      JTextArea help = new JTextArea(
            "Place cursor in text field on the right to send commands.");
      help.setEditable(false);
      help.setLineWrap(true);
      help.setAlignmentX(CENTER_ALIGNMENT);
      help.setBackground(blue_add.getBackground());

      name.add(nxt_name1);
      name.add(nxt_name);
      name.add(blue_add);
      name.add(bluetooth);
      send.add(help);
      send.add(text);
```

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setSize(new Dimension(350, 200));
frame.setVisible(true);
frame.add(name);
frame.add(connect);
frame.add(send);
frame.add(endConnect);

// Put an action command for the Connect Button
connect.addActionListener(new ActionListener()
{
   public void actionPerformed(ActionEvent ae)
   {
      if (ae.getSource() == connect) {
         connectNXT();
      }
   }
});
// Put an action command for the End Connection Button
endConnect.addActionListener(new ActionListener()
{
   public void actionPerformed(ActionEvent ae)
   {
      if (ae.getSource() == endConnect) {
         try {
               dos.close();
               nxtComm.close();
         } catch (IOException e) {
         }
         connect.setEnabled(true);
         endConnect.setEnabled(false);
      }
   }
});

// add a key listener to the text filed to send commands on Bluetooth connection
text.addKeyListener(new KeyListener()
{
   public void keyReleased(KeyEvent e)
   {
      try {
         int keyPressed = e.getKeyCode();
         int direction;
         switch (keyPressed)
         {
            case directionLeft:
               direction = 4;
               break;
            case directionRight:
               direction = 6;
               break;
            case directionForward:
               direction = 2;
               break;
            case directionBackward:
               direction = 8;
               break;
            case reset:
               direction = 1;
               break;
            default:
               direction = 5;
               break;
         }
            dos.writeInt(direction);
            dos.flush();
      } catch (java.io.IOException e2) {
         System.out.println("IOException");
         return;
```

```
            }
        }

        public void keyPressed(KeyEvent e){}
        public void keyTyped(KeyEvent e){}
    });
    // add Enter listeners to easily connect to NXT
    nxt_name.addKeyListener(new KeyListener()
    {
        public void keyReleased(KeyEvent e)
        {
            int keyPressed = e.getKeyCode();
            if (keyPressed == KeyEvent.VK_ENTER) {
                connectNXT();
            }
        }

        public void keyPressed(KeyEvent e){}
        public void keyTyped(KeyEvent e){}
    });

    bluetooth.addKeyListener(new KeyListener()
    {
        public void keyReleased(KeyEvent e)
        {
            int keyPressed = e.getKeyCode();
            if (keyPressed == KeyEvent.VK_ENTER) {
                connectNXT();
            }
        }

        public void keyPressed(KeyEvent e){}
        public void keyTyped(KeyEvent e){}
    });
}

/**
 * Connect to an NXT device and open input and output streams.
 */
public static void connectNXT()
{
    System.out.println("Connecting to GELway...");
    nxtComm = null;
    try {
        nxtComm = NXTCommFactory.createNXTComm(NXTCommFactory.BLUETOOTH);
        NXTInfo nxtInfo = new NXTInfo(NXTCommFactory.BLUETOOTH, nxt_name.getText(),
              bluetooth.getText());
        nxtComm.open(nxtInfo);
    } catch (lejos.pc.comm.NXTCommException e1) {
        System.out.println("NXTCommException " + e1.getMessage());
        return;
    }
    System.out.println("Connected.");

    // Attach output stream to the bluetooth connection
    OutputStream os = nxtComm.getOutputStream();
    dos = new DataOutputStream(os);
    connect.setEnabled(false);
    endConnect.setEnabled(true);
}
}
```

Bibliography

[1] David G Duff, Mark Yim, and Kimon Roufas. Evolution of PolyBot: A modular reconfigurable robot. In *Harmonic Drive International Symposium 2001; 2001 November 20-21; Nagano, Japan and COE/Super-Mechano-Systems Workshop*, Tokyo, Japan, November 2001.

[2] LEGO Mindstorms. The NXT. `http://mindstorms.lego.com/Overview/The_NXT.aspx` Accessed On: 5 Feb 2009, February 2009.

[3] Yorihisa Yamamoto. *NXTway-GS Model-Based Design-Control of self-balancing two-wheeled robot built with LEGO Mindstorms NXT*. CYBERNET SYSTEMS CO., LTD., 2008.

[4] Albert Ko, H.Y.K. Lau, and T.L. Lau. SOHO Security with Mini Self-Balancing Robots. *Industrial Robot: An International Journal*, 32(6):492–498, 2005.

[5] Segway. Technology Innovation: The Science of Segway. `http://www.segway.com/about-segway/science-of-segway.php` Accessed On: 2 Feb 2009, February 2009.

[6] Haruhisa Kurokawa, Eiichi Yoshida, Kohji Tomita, Akiya Kamimura, and Shigeru Kokaji. M-TRAN II: Metamorphosis from a Four-Legged Walker to a Caterpillar. In *Intl. Conference on Intelligent Robots and Systems*, pages 2454–2459, Las Vegas, Nevada, October 2003. IEEE/RSJ.

[7] Haruhisa Kurokawa, Kohji Tomita, Akiya Kamimura, and Takashi Hasuo. What's M-TRAN. `http://unit.aist.go.jp/is/frrg/dsysd/mtran3/what.htm` Accessed On: 15 Oct 2009, August 2008.

[8] Haruhisa Kurokawa, Eiichi Yoshida, Kohji Tomita, Akiya Kamimura, Satoshi Murata, and Shigeru Kokaji. Deformable Multi M-TRAN Structure Works as Walker Generator. In *Intl. Conference on Intelligent Robots and Systems*, pages 746–753, Amsterdam, 2004.

[9] Bill Steele. Cornell robot discovers itself and adapts to injury when it loses one of its limbs. `http://www.news.cornell.edu/stories/nov06/resilientrobot.ws.html` Accessed On: 15 Oct 2009, November 2006.

[10] Josh Bongard, Victor Zykov, and Hod Lipson. Resilient machines through continuous self-modelling. *Science*, 314(5802):1118–1121, November 2006.

[11] Josh Bongard, Victor Zykov, and Hod Lipson. Self Modeling Robotics: Movies and Pictures. Cornell University, `http://ccsl.mae.cornell.edu/research/`

selfmodels/pictures/Starfish_Standing.JPG Accessed On: 15 Oct 2009, November 2006.

[12] Reinforcement Learning: A Tutorial. *NXT Gyro Sensor.* Wright Laboratory and Wright State University, 1997.

[13] Segway. About Segway: Segway Milestones. http://www.segway.com/about-segway/segway-milestones.php Accessed On: 2 Feb 2009, February 2009.

[14] Phil Ament. Segway HT. http://www.ideafinder.com/history/inventions/segway.htm Accessed On: 2 Feb 2009, February 2005.

[15] Tom Harris. How Segways Work. http://science.howstuffworks.com/gyroscope.htm Accessed On: 2 Feb 2009, February 2001.

[16] Steve Hassenplug. Steve's LegWay. http://www.teamhassenplug.org/robots/legway/ Accessed On: 5 Feb 2009, February 2003.

[17] HiTechnic. NXT EOPD. http://www.hitechnic.com/contents/en-us/d28.html Accessed On: 5 Feb 2009, February 2009.

[18] Philippe E. Hurbain. Ultrasonic Sensor. http://www.philohome.com/nxtway/nxtway.htm Accessed On: 5 Feb 2009, February 2009.

[19] Ryo Watanabe. *Motion Control of NXTway(LEGO Segway).* Waseda University, 2007.

[20] Joseph L. Jones and Daniel Roth. *Robot Programming - A Practical Guide to Behavior-Based Robotics.* McGraw-Hill Companies, 2003.

[21] Jose Solorzano Dell, Juan Antonio Brea Moral. *The leJOS NXJ Tutorial.* Esmeta, 2009.

[22] Graz University of Technology. RoboCup 2009. http://www.robocup2009.org/1-0-home.html Accessed On: 15 Oct 2009, October 2009.

[23] Floris Mantz and Pieter Jonker. *Behavior-Based Perception for Soccer Robots.* Delft University of Technology, The Netherlands, 2006.

[24] ZDNet. Images: Aibo, Gort join Robot Hall of Fame. http://content.zdnet.com/2346-9595_22-10728.html Accessed On: 15 Oct 2009, October 2009.

[25] S Russell and P Norvig. *Artificial Intelligence a Modern Approach.* Prentice Hall, New Jersey, 1995.

[26] Patrick Lin, George Bekey, and Keith Abney. *Autonomous Military Robotics: Risk, Ethics, and Design.* California Polytechnic State University, San Luis Obispo, 2008.

[27] Steven Hassenplug. NXT Programming Software. http://www.teamhassenplug.org/NXT/NXTSoftware.html Accessed On: 12 Jun 2009, October 2008.

[28] Johnny Rieper, Bent Bisballe Nyeng, and Kasper Sohn. *Marvin - The Balancing Robot*. Aarhus University, 2009.

[29] Farid Golnaraghi Benjamin C. Kuo. *Automatic Control Systems*. John Wiley and Sons, Inc., 2003.

[30] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems 10th Edition*. Pearson Education, Inc., 2005.

[31] HiTechnic. *NXT Gyro Sensor Manual*. Dataport Systems, Inc, 2009.

[32] S. Ramachandran and I-Ming Chen. Distributed Agent Based Design of Modular Reconfigurable Robots. In *Proceedings of the 5th International Conference on Computer Integrated Manufacturing*, pages 447–458, 2000.

[33] Andy Shaw. *leJOS Forum - Upload speed using bluetooth*. leJOS, `http://lejos.sourceforge.net/forum/viewtopic.php?p=5946&sid=1c5c4621d236a752e05f90e0c35c7192` Accessed On: 30 Sep 2009, 2009.

[34] Nokia. Nokia 6220 classic. `http://www.nokia.com.au/NOKIA_ASIA_2/Product_Catalogue/Products/Consumer_Phones/6000-series/6220_classic/pdf/nokia_6220classic_factsheet.pdf` Accessed On: 2 Jun 2009, June 2009.

[35] Pedro Miguel. NXT-Symbian. `http://nxt-symbian.sourceforge.net/` Accessed On: 30 Sep 2009, September 2006.

[36] Francis Li. Mobile Processing. `http://mobile.processing.org/` Accessed On: 30 Sep 2009, September 2004.

[37] Dell. *Dell Studio XPS 15 Laptop*. `www.dell.com.au` Accessed On: 1 Feb 2009, 2008.

[38] Y. Sun and P. Ioannou. *A Handbook for Inter-Vehicle Spacing in Vehicle Following*. Research Reports. University of Southern California, 1995.

[39] HiTechnic. NXT Programming Software. `http://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=NSK1042` Accessed On: 17 Oct 2009, October 2009.

[40] HiTechnic. IR Beacon. `http://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=FTCBCN` Accessed On: 17 Oct 2009, October 2009.

[41] HiTechnic. Sensor Multiplexer. `http://www.hitechnic.com/cgi-bin/commerce.cgi?preadd=action&key=NSX2020` Accessed On: 17 Oct 2009, October 2009.

[42] LEGO Mindstorms. Servo Motors. `http://mindstorms.lego.com/Overview/Interactive_Servo_Motors.aspx` Accessed On: 5 Feb 2009, February 2009.

[43] Yorihisa Yamamoto. *LEGO MINDSTORMS NXTway-GS Building Instructions*. CYBERNET SYSTEMS CO., LTD., 2008.