THE UNIVERSITY OF QUEENSLAND

School of Information Technology and Electrical Engineering

ACTIVE BALANCE FOR A HUMANOID ROBOT

By

Toby Daniel Low

The School of Information Technology and
Electrical Engineering
The University of Queensland

Submitted for the degree of Bachelor of Engineering (Honours)
in the division of Computer System Engineering
29th October 2003

Daniel Toby Low

Lab 311, University of Queensland

St. Lucia, Brisbane

QLD 4067, Australia.

29th October 2003

Head

School of Information Technology and Electrical Engineering,

The University of Queensland

St. Lucia QLD 4067

Dear Professor Kaplan,

In accordance with the requirements of the Degree of Bachelor of Engineering (Honours) in the School of Information Technology and Electrical Engineering, I would like to submit the following thesis entitled

"Active Balance of a Humanoid Robot"

This thesis was performed under the supervision of Dr Gordon Wyeth. I declare that this work submitted in this thesis is that of my own, except work that has been acknowledged in text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours Sincerely,

_____

Daniel Toby Low

TABLE OF CONTENTS

LIST OF FIGURES

PAGE

LIST OF TABLES

PAGE

LIST OF EQUATIONS

PAGE

ABSTRACT

Conventional robots have helped the world tremendously from going where no man has gone before to improving the output of factory operations. The only problem with many of these robots is that they operate in an environment built for them. As so much of our infrastructures and home/work environments have been moulded to suit the human physical makeup, these robots cannot help humans in these environments. This is where humanoid robots will revolutionise the way we humans work and live providing endless possibilities and uses.

The main problem in the development of humanoid robots is in implementing or mimicking the Central Nervous System (CNS), brain and the sensors of the human body. With these tools, humans are able to balance and perform movements subconsciously. In order for a humanoid robot to perform these trivial tasks, many control loops must be formulated to keep the robot stable and perform the desired tasks correctly.

The scope and goal of this thesis is to design and implement an active balance system that will enable the robot to remain upright despite adverse disturbances. These disturbances were classified into 3 categories, small, medium and large disturbances. Large disturbances were classified those forces where the robot is required to take a step to remain upright and due to the complexity involved in countering large disturbances, the scope of the thesis was limited to only countering small to medium disturbances.

To counter these small to medium disturbances, an inertial measurement unit or IMU sensor system was implemented to detect disturbances through change in inclinations and a two-mass inverted pendulum model of the GuRoo robot was formed. A control system to keep the posture on the robot upright was designed using this model and various tests performed in simulation as well as on the real robot using the GuRoo source code and *Dynamechs* package. Examining the simulation results, it was found that the robot would not be able to counter disturbances without some form of ankle control included. Thus due to ankle balance system part of the overall balance system not being completed in time; a crude substitute ankle system was used.

With this new ankle system, the designed active balance system was implemented successfully on the humanoid robot keeping the robot's overall angle close to the desired angle and thus allowing the robot to withstand small to medium impact and sustained pushes. The system could also balance the crouching motion of the robot but was unsuccessful for the previously developed GAWalk due to the large changes in inclination produced by the walk. Although there was no success in balancing the walk, the newly developed walk is much smoother showing great promise. This combined with the currently developing ankle compliance balance system shows even greater promise for overall balance of the entire robot against a wide range of disturbances.

ACKNOWLEDGEMENTS

I would like to acknowledge and thank the following people for their contributions and help towards the completion of this thesis:

- My Family for their support and understanding throughout the year.
- My supervisor, Dr Gordon Wyeth for allowing me to develop my research skills, passing on his valuable knowledge and guiding me when needed.
- Damien Kee for his time and assistance in helping me understand the GuRoo project, maintaining the robot and conducting thesis tests.
- The GuRoo 2003 team – Tim Pike, Anthony Peters, Simon Matthews and Nick Underly for all their support and humor.
- Other robotic lab members for helping me out in times of need.
- My Friends for helping through this year and providing me with important thesis information.

CHAPTER 1

INTRODUCTION

In this chapter, the background of humanoid robots will be introduced and details of the RoboCup competition will be outlined. In addition, the motivation for the development of humanoid robots will be discussed as well as the problems encountered with the balancing of humanoid robots. Finally, an outline of the subsequent chapters is given.

1.1 WHY HUMANOID ROBOTS EXIST?

Humanoid robots are currently a key development area for many institutions in the area of robotics. Many conventional robots operate within fields isolated from the human society such as factories, mines and under the sea [5]. These robots operate in environments that are too dangerous for humans to enter helping to explore and do tasks beyond human capability. These robots applications are certainly unique however the coming generations of robots are expected to exist and operate within the human environment and society that has been created. The push for robots to perform trivial human tasks is being more apparent as they develop. Nonetheless, these tasks may seem trivial for us but the complexity, intellect and body control that humans perform unconsciously is not so trivial for robots. The integrated sensor, processor and reaction systems of the human body are very advanced compared to that of a robot. This is where the current research of humanoid robots is and with continuing research, future robots shall soon coexist within human society such as schools, homes and offices.

1.2 ROBOCUP COMPETITION

The RoboCup competition is an international research and education initiative. Its goal is to promote artificial intelligence and robotics research by providing a competition in which a wide range of technologies and concepts can be developed and implemented. The RoboCup competition began in July 1997 with a single soccer league and has now expanded into eight (8) different leagues/competitions. The vision of the RoboCup competition is “By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champions” [6]

To help realise this vision, a solo division of the RoboCup competition is held in order to help develop the required intellect, balance and speed. In 2002, the robot competed in the following challenges:

1. Walking – The robot must be able to walk 6 metres, walk around a marker and walk back to where it started.
2. Stand on One Leg – The robot must be able to stay stable for 1 minute whilst one foot is off the ground.
3. Penalty Kick – The robot must be able to kick a soccer ball into an open goal.
4. Free Style – The robot can perform a 5-minute demonstration of the team's choice.

1.3 ROBOT BALANCING

In traditional legged robots, stability is maintained by having at least three (3) contact points with the ground surface at all times [7]. With biped machines, only two (2) points are in contact with the ground surface and as a result alternative methods for maintaining balance and stability must be implemented.

To help gain an insight of the difficulties involved in balancing biped robots, the balancing behaviour of human beings can be explored. The complexity of the nervous system in the human body allows for feedback of various sensor systems information [8]. Using this information consisting of foot pressures, external forces applied, visuals and inertias, the human brain can compute the various joint and structural movements to maintain stability. The integration of sensors, balance and a control system to match the human body is extremely difficult. The design of the perfect balance system involves an enormous amount of computation and control. Thus to help the process of achieving humanoid balancing, simplified models have been developed and will be discussed in later chapters.

1.4 CHAPTER OUTLINE

The remainder of this document is divided into nine (9) chapters. A description of each chapter is outlined below:

Chapter 2: Literature Review – Introduces background information and key concepts within the field of study. Additionally the "GuRoo" project will be introduced and examined along with other humanoid robot projects. Finally the approach that will be taken in this thesis paper will be outlined.

Chapter 3: Specifications – Defines the topic, purpose and goals for the thesis as well as how the project will be planned. Additionally, project constraints and methodologies will be defined.

Chapter 4: Modelling the Humanoid Robot – This chapter will compare and determine the balance model that will be used to create a control system for. The dynamic and state space equations will be developed in this chapter as well.

Chapter 5: Balance Control System – The balance control system design will be outlined in this chapter. First the conversion from a MIMO to a SISO system is conducted and the control system designed using root locus techniques. The MATLAB simulated results are then compared and additional refinements to the control system are described.

Chapter 6: Attitude Behaviours – Introduces and describes two (2) attitude behaviours that will be implemented in the active balance system. These two behaviours are posture and COG control.

Chapter 7: Software Design and Implementation – Describes the software design flow of the simulation system and of the balance software system.

Chapter 8: Simulation Results and Refinements – This chapter will present simulation results obtained from the GuRoo simulator. Additional refinements required at this stage will also be outlined with the final simulation results that will be compared to real hardware results presented.

Chapter 9: Hardware Implementation – Describes the IMU hardware and software implementation required to conduct tests on the real robot. In addition, this chapter presents and explains the real hardware results and compares the results with simulation.

Chapter 10: Discussion – This chapter discusses and evaluates the active balance system. It will also draw conclusions about the active balance system and identify future developments that can be made to the active balance system.

CHAPTER 2

LITERATURE REVIEW

This chapter will introduce relevant background information required in order to understand the rest of the report. Key concepts and terminology on balancing and control will be given. Furthermore existing humanoid balance and control systems including the "GuRoo" robot will be analysed. Finally, the approach that will be taken in the project will be outlined.

2.1 BACKGROUND INFORMATION

2.1.1 KEY CONCEPTS AND TERMINLOGY

To define balance and stability models and criteria, some key concepts and terminology must be defined. The most commonly used terminology [7] is detailed below:

- Centre of Gravity (C.o.G) – a position within the robot in which the robot can be considered as a point mass at that location.
- Zero Moment Point (Z.M.P) – a point on the contact surface where the total forces and moments acting on the robot is equal to zero.
- Foot Rotation Indication (F.R.I) – a point on the contact surface where a net ground reaction force is required in order to keep the robot stationary.
- Support Polygon – the convex hull of contact points of the robot. In the single foot support phase, the support polygon is defined by the outline of that support foot. During double support phase, the wrapped area in which the feet cover is considered the support polygon. This is shown in Figure 1 below.

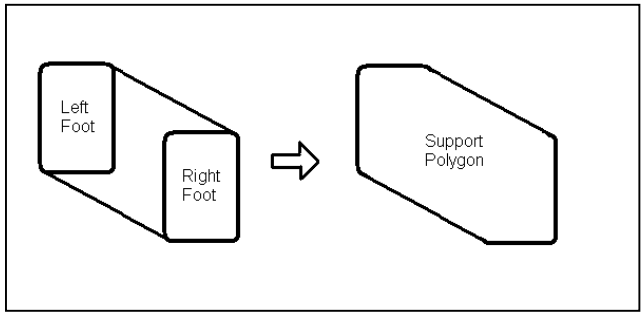


Figure 1: Top View of Double Support Phase Polygon

- Centre of Pressure (C.o.P) – a point on the contact surface where the net ground reaction force acts through.
- Stability Margin – the distance between the nearest support polygon boundary and the *Zero Moment Point* (Z.M.P). This is shown below in Figure 2.

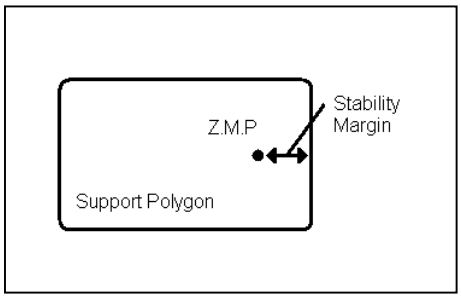


Figure 2: Stability Margin [1]

2.1.2 TYPES OF WALKING

A humanoid robot walk can be classified into two (2) types, static and dynamic. *Static walking* involves the controlling of the ground projection of the C.o.G to remain within the support polygon of the robot at all times [9]. Static walking assumes that all dynamics forces produced by the motion of the robot's limbs are small compared to the gravitational forces of the robot. Static walking is usually very slow, as the robot must minimise the dynamic forces produced by the motion of the limbs. With static walkers, the robot can be stopped at any time in its gait and remain in a stable position.

Dynamic walkers on the other hand adjust their gait according to the situation the robot is in. The stability of dynamic walkers requires the calculation and manipulation of the Zero Moment Point (Z.M.P). Unlike static walkers, the C.o.G does not need to remain in the support polygon; only the Z.M.P must remain in the support polygon in order for the robot to stay stable. Vukobratovic [10] introduced the Z.M.P concept and now the most widely used concept for defining stability in the field of walking robots. The Z.M.P takes into account dynamic moments due to the motions of the robot limbs and any external forces. When the robot is in a dynamically stable walk, the Z.M.P, F.R.I and C.o.P are all coincident [2].

2.1.3 BALANCE MODELS

Of course the concept of the Z.M.P may seem easy but the processing power involved calculating the exact Z.M.P of the robot is quite large. Thus robots have been modelled into simpler forms, which are easier to implement. Common models are listed below:

1. Inverted Pendulum Model [4] – considers the robot to be a point mass at the end of a mass-less telescopic leg extending from the centre of the supporting base to the C.o.G of the robot. This model is shown in Figure 3 and significantly simplifies the analysis and calculation of dynamic motion. Although this model simplifies analysis, the trade-offs are performance limitation and undershoot . Due to the model being a non-minimum phase system, unavoidable undershoot of the control will occur. Also, the assumption a mass-less telescopic leg is not applicable, as the legs of many robots comprise about 50% of the robots total mass. The torso and head regions of the robot are also quite heavy in many robots and must be taken into consideration.

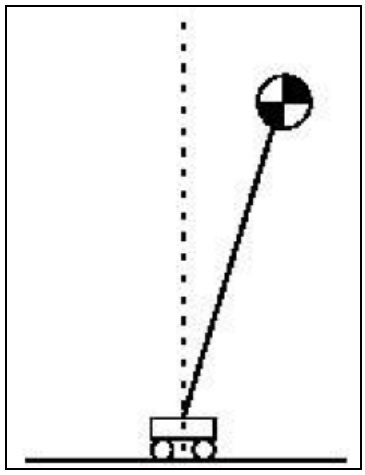


Figure 3: Inverted Pendulum Model [4]

2. Two Masses Inverted Pendulum Model – considers the robot to two point masses located at the C.O.G of the legs and the torso regions of the robot. One mass represents the motion of the legs and the other being the torso motion. This model describes the lower and upper body of the humanoid robot and provided that the two masses are not coincident, no undershoot limitation will occur. A diagram of this model is shown in a subsequent section in Figure 9.

3. Gravity Compensated Inverted Pendulum Model [11] – this model was developed by Park [11] due to his findings on the inaccuracy of the Inverted Pendulum Model, as the inertia of the legs is a main contributor to the location of the Zero Moment Point. Park developed the Gravity Compensated Inverted Pendulum Model (GCIPM) shown in Figure 5, which considers the robot being two (2) separate masses on two mass-less rods, one representing the swinging leg of the robot and the other representing the remaining masses of the robot. From this model, the effect of the swing leg can then be taken into account when calculating the Z.M.P and motions required to balance the robot.

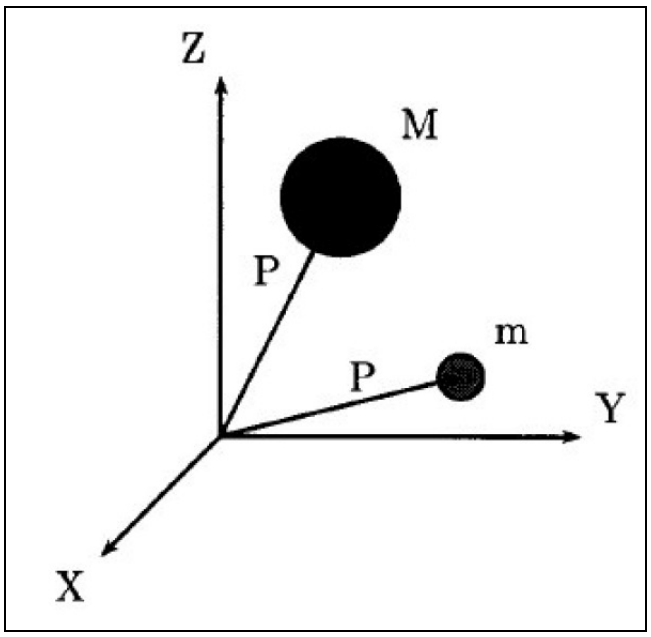


Figure 5: Gravity Compensated Inverted Pendulum Model [11]

2.1.4 CONTROL CONCEPTS AND TERMINLOGY

In defining control methods and systems, the major concepts and terminology used are given below:

- Feedback Control – this is a method of obtaining information from the actual output of the system and using it to modify the input of the system in order to generate the desired output.
- Feedforward Control [8] – is the process of measuring disturbances before it affects the output and compensating for those disturbances when they happen. This method requires the predicting of disturbances and if done correctly, it is possible to completely eliminate measured disturbances.

2.2 THE "GUROO" PROJECT

2.2.1 OVERVIEW

The University of Queensland is currently developing a humanoid soccer robot named "GuRoo". The robot consists of 23 DOF and weighs in at approximately 38kg [2]. It is currently in its 3rd year of development and can walk at a speed of 0.03m/s. The GuRoo robot currently does not have any active balance system implemented and has no awareness of its external state apart from motor encoder positions. It uses a predetermined gait factoring in Z.M.P analysis to keep the robot stable whilst walking.. Although the robot can walk, some support is required at times to keep the robot upright, as the robot cannot compensate for any external disturbances. The "GuRoo" project uses Dynamechs [12] to simulate any gait, balance and control methods before implementation on the robot.

The 2003 "GuRoo" team plan to improve the robot by addressing the following aspects:

- Integration of a 3-axis inertia and gyroscope sensor.
- Development of an active balance system to keep the robot upright.
- Integration of foot force sensors to provide support base and Z.M.P information
- Development of a flexible, parameterised and reliable gait.
- Re-design and implementation of robots motor control and processor boards.
- Integration of a camera for image recognition and tracking of a soccer ball.

The common goal of the team is to develop a humanoid robot that can track a soccer ball, walk up to it and kick the ball by the end of 2003. A diagram of the overall control system that the team is working on is shown in Figure 6.

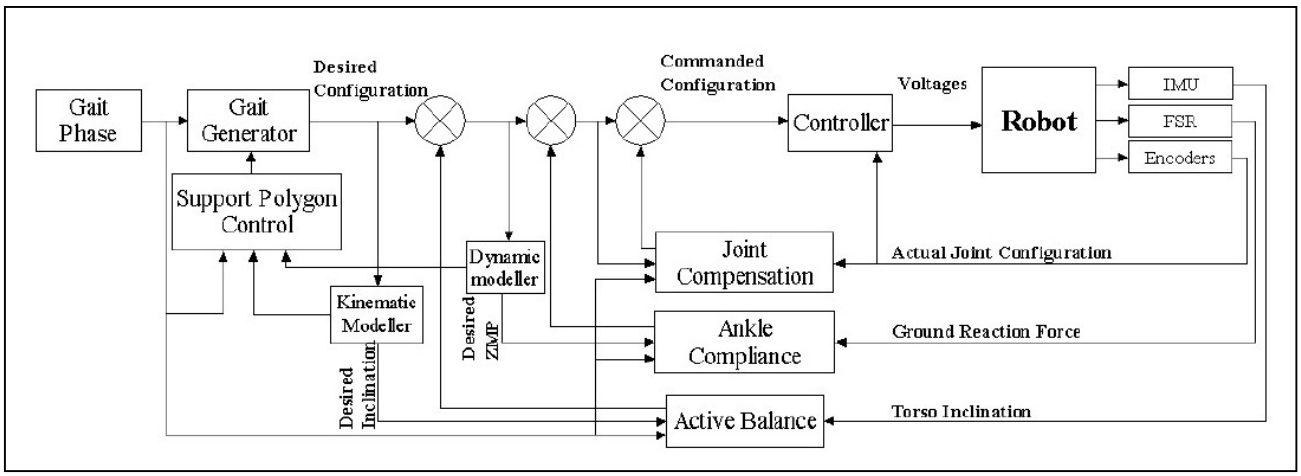


Figure 6: Block Diagram of Proposed "GuRoo" Control System [2]

Examining the diagram, the foot sensors will control the Ankle Compliance system whilst the Active Balance system uses the IMU sensor. This thesis will primarily focus on the Active Balance Control system using the IMU sensors, Ankle Compliance and Gait. A brief description of the various control loops defined by Kee [2] are given below:

- Joint Compensation – used as a predictive modulator to compensate for errors in joint trajectory for the gait phase and will modify the *Desired Configuration* appropriately.
- Ankle Compliance – the control of the robots ankle to comply with an inclined ground plane. This allows the desired Z.M.P and the ground reaction force to be coincident and thus no tipping moment is produce on the robot. *The Desired Z.M.P* is estimated by the *Dynamic Modeller*, which is based on the *Desired Configuration* of the robot. Ankle Compliance uses the input from the force foot sensors in order to control the ankle movements.
- Active Balance – designed to reject or compensate for external adverse disturbances. This control loop will be used to maintain stability and balance through the use of the *Kinematics Modeller*. The *Desired Configuration* and *Kinematics Modeller* define the desired torso position and motion. This is then compared to the motion sensed by the IMU sensor that will drive the control loop. This loop should counteract disturbances by attitude behaviours implemented on the torso.
- Support Polygon Control – produces a new support polygon modifying the *Gait Generator* if the *Active Balance* control loop cannot handle the disturbance. The realisation of the new support polygon requires a series of movements has to be incorporated into the Gait Generator in order stabilise the robot from such a large disturbance.

2.2.2 PREVIOUSLY DEVELOPED BALANCE SYSTEM

Recent active balance development by Ian Joseph Marshall [7] on the "GuRoo" robot modelled the robot by a set of point masses corresponding to each link. The accelerations of these links were then calculated through double integration of the link positions. From the location of these point masses and accelerations of the point masses, the zero moment point was then calculated. Although accelerations were calculated for each link, these accelerations were only due to joint motions. External forces were not taken into account in this balance system due to the lack of sensors implemented within the robot. The reaction forces from the ground were also not taken into account when the Z.M.P was calculated.

For the balance behaviour method to modify the Z.M.P position. Ian Joseph Marshall proposed a simple method of controlling the robots ankle movements. If the Z.M.P is exiting the support polygon, ankle movements would be initiated to move the Z.M.P within the desired threshold area. Although this model and method was proposed, only simulations were conducted. No real hardware implementation was conducted on the "GuRoo" robot due to development stage of the robot.

2.3 EXISTING HUMANOID ROBOTS

2.3.1 "WABIAN"

The Bipedal Humanoid Robot "Wabian" was developed at the Waseda University by [13] with studies on biped robots conducted since 1966. The total weight of the robot is 107kg, height of 1.66m and a total of 35 active DOFs. At present, "Wabian" can perform normal biped walking (forward and backward), dynamic dancing waving arms and hips as well as trunk-waist co-operative dynamic walking.

The balance system of "Wabian" involves the control of the Z.M.P of the robot. By keeping the actual Z.M.P within a certain error of the planned Z.M.P, the robot will remain stable. The robot was modelled as a system of particles consisting of the main masses. These were then simplified into eight (8) point masses to help calculate the required trunk motions to balance the robot. This is shown below in Figure 4. To balance the robot, the two Z.M.P's are compared and solutions for trunk/spine motions to counter external moments are formulated through a series of iterative algorithms. This behaviour was successfully implemented and the "Wabian" could dynamically walk at a speed of 0.7km/h.

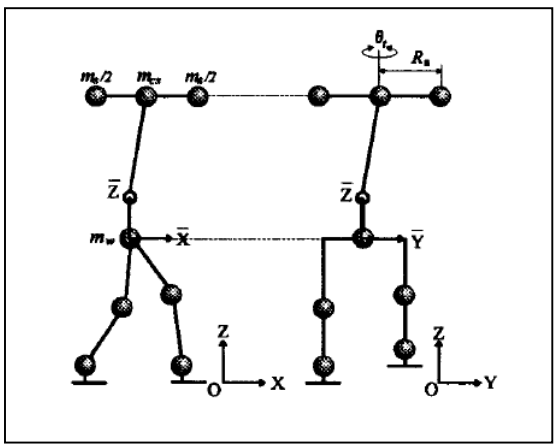


Figure 7: Linearised Model. [4]

2.3.2 THE HONDA HUMANOID ROBOT

The HONDA robot began in 1986 with the development of a humanoid robot with two arms and two legs by Hirai, Hirose, Haikawa and Takenaka [3]. This robot had a height of 1.82m, weight of 210kg and a total of 28 DOF's with two (2) DOF's used in the hand gripping mechanism. The HONDA robot has an impact absorption mechanism consisting of a rubber bush inserted into a guide. This helps minimise the landing impact force and vibrations of the leg.

The HONDA robot has three (3) control methods used to completely balance the robot, the Ground Reaction Force Control, Z.M.P Control and Foot Landing Position Control. In these control methods, the common element is to modify the Z.M.P and "Centre of Actual Total Ground Reaction Force" (C-ATGRF) to produce a tipping moment of zero. The C-ATGRF is the point on the ground in which the moment produced by the "Actual Total Ground Reaction Force" (ATGRF) becomes zero. The ATGRF is the combination of the ground reaction forces acting on both feet of the robot.

The Ground Reaction Force control method uses the C-ATGRF and shifts it to the appropriate position by adjusting each foot's desired position and posture. When the body tips forward, the robot lowers the front toe section of the supporting foot. When the robot tips backward, the robot lowers the heel of the supporting foot to shift the C-ATGRF rearward.

The Z.M.P control model is used to control the shifting of the desired Z.M.P in order to recover the robots posture. Changing the ideal body trajectory when the robot is about to tip over does this. The importance of posture is demonstrated with an example in which the upper body of the robot is inclined forward, even if the desired Z.M.P and the C-ATGRF are at the same point, the robot will still fall over unless its posture is corrected. Thus the Z.M.P control model increases the magnitude of the desired inertia force by accelerating the desired upper body position, according to the inclination of the robot. This changes the direction of the total inertia force and thus the position of the desired Z.M.P is shifted backward or forward from the original desired Z.M.P and consequently the posture of the robot will recover.

The Foot Landing Position control model corrects the relative position of the upper body and the feet in conjunction with the Z.M.P control. When the Z.M.P control recovers the robots upper body posture, the spatial configuration of the body and feet will differ from the ideal state. Thus the Foot Landing Control lengthens the stride and corrects the relative positions of the next supporting foot and the upper body to gradually recover to its original state.

To implement these three (3) control methods, a six (6) axis force foot sensor is used along with body inclination and inertia sensors. By having these three controls described working simultaneously, the HONDA robot has achieved a robot with posture balancing control similar to that of a human. A overall block diagram of the HONDA robots posture recovery system is shown in Figure 5 below.

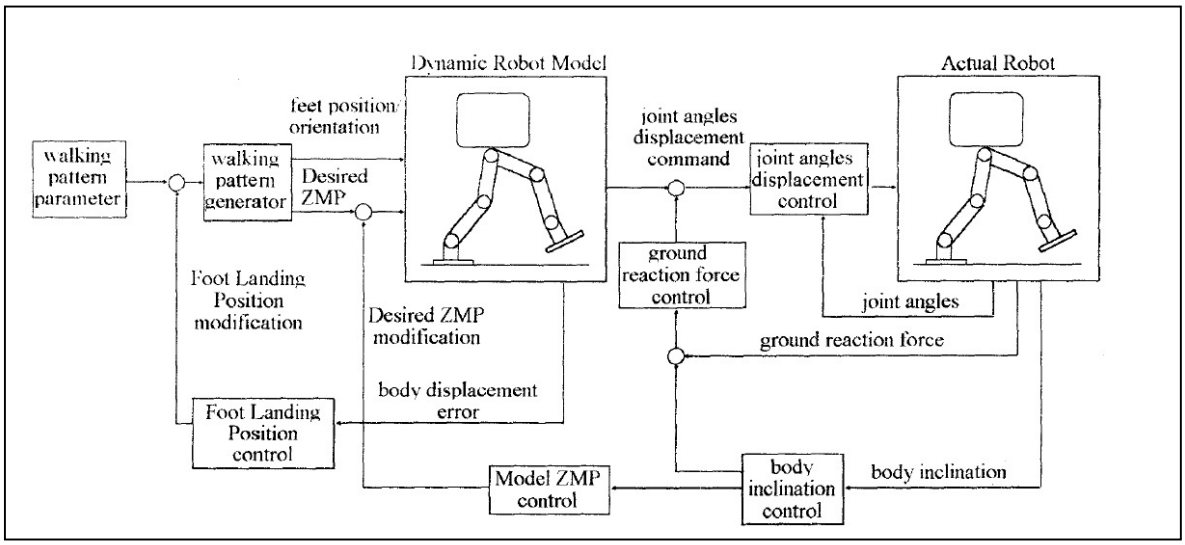


Figure 8: HONDA's Overall Block Diagram for Body Recovery Control [3].

2.4 OTHER CONTROL AND BALANCE TECHNIQUES

2.4.1 DIRECT ANGULAR MOMENTUM FEEDBACK

Shuuji Kajita, Kazuhito Yokoi, Muneharu Saigo and Kazuo Tanie [14] used the idea of balance control by direct feedback of total angular momentum and the position of the C.O.G. Contact torque sensors are used to directly manipulate the total angular momentum of the robot to keep it stable. In this control, only the ankle actuators of the support leg were used for the balancing. This method although never stated, still uses the concept of manipulating the Z.M.P.

2.4.2 FEEDBACK, FEEDFORWARD AND FUZZY CONTROL

A team at the University of Waterloo [8] have developed a Feedback, Feedforward and Fuzzy control system to balance and posture control during a gait. The team's objective is to mimic the response (both kinematics and dynamics) of the HAT (Head-Arms-Torso) to that of the Central Nervous System (CNS) during gait. It was found that the linear feedback control loops performed poorly in the sense that it could not cope adequate with the time delays and slow muscle responses present in the system. The led the team to develop the Feedback and Feedforward control loop that must be able to anticipate the onset of disturbances and correct them as the disturbances occur. It was found from the results of this control system combination that it reduced the oscillations of the HAT and the hip torque requirements significantly.

2.5 BALANCE CONTROL SYSTEM APPROACH

Obviously the Z.M.P is a key criterion in the balancing and defining stability for humanoid robots. All of the existing humanoid robots have used the concept of the Z.M.P to some extent. The Z.M.P, C.O.G and posture concepts will be the main criteria in the development of the active balance system for this thesis.

The balance behaviour solutions in this thesis will be focused on the torso, arms and head links of the robot, as the new gait being developed will control the legs and hips. This is to done to allow the team to develop their parts on separate parts individually keeping the system modularised and interchangeable.

To begin with, a balance control system that corrects the robots torso inclination by directly comparing the desired inclination to the actual inclination will be developed. This will then be tested and evaluated to help develop a more advanced active balance system. The integration of the ankle compliance control loop and the active balance control loop with complete overall balance system of the "GuRoo" robot. Unfortunately, due to the ankle compliance system not being completed, a very simplified substitute ankle control system was implemented to help complete the thesis topic outcomes.

CHAPTER 3

SPECIFICATIONS

This chapter will describe the purpose and goal of the thesis project as well as outlining the specifications and various constraints involved in the thesis project. The thesis project methodologies will also be defined.

3.1 PURPOSE

The primary purpose of this thesis is to allow the humanoid robot "GuRoo" to compete in the RoboCup competition by performing a series of tasks whilst remaining in a balanced and upright state. These tasks are given below:

- Walking – The robot must be able to walk 6 metres, walk around a marker and back
- Stand on One Leg – The robot must be able to stay stable for 1 minute whilst one foot is off the ground.
- Penalty Kick – The robot must be able to kick a soccer ball into an open goal.
- Free Style – The robot can perform a 5-minute demonstration of the team's choice.

The balance system is used in order to keep the robot within a state that can be easily controlled and manoeuvred with minimal risk of the robot ending up in an uncontrollable state. Unfortunately, the team did not go to this years RoboCup competition as it was decided that more focus could be put on improving and upgrading the robot rather then preparing for the competition.

3.2 GOAL

The overall goal of this thesis is to create and implement a system that will help allow the humanoid robot "GuRoo" to remain upright and balanced despite adverse disturbances. The primary goal for this thesis will be aimed at recovering and keeping the robot remaining upright against external disturbances such as a push, airflows and joint backlash. Disturbances are classified into three (3) categories, small, medium and large. Small disturbances are classified as those that move the overall angle less then 5 degrees whereas medium disturbances are those that move the overall angle by 5-10 degrees. Large forces are classified as those that move the torso by greater than 10 degrees and

generally requiring the robot to take a step to stay upright and balanced. The current humanoid robot requires outside assistant in order to stay upright and balanced whilst performing various tasks. Hence this thesis project aims to remove any outside assistant required.

In order to balance the robot, gyro and inertial sensors will be integrated into the robot. This in turn will give information about the robot accelerations, inclinations, angular rates and angular accelerations. The balance system will be comprised of four (4) main parts and will be developed in the following order listed below:

1. Sensors and sensor information - External forces and robot's state information is required in order to actively balance the "GuRoo" robot.
2. Balance Models and Criteria – Models of the robot must be developed and criteria for stability defined in order for the robot to know when it is stable or unstable.
3. Control techniques – Control methods to obtain sensor information and use that information to control the torso pitch and roll motors.
4. Behaviour techniques – Methods of torso trajectories and movements in order to keep the robot upright and counter any external forces.

3.3 METHODOLOGY

The methodologies used to achieve this goal are chosen on the basis on current active balance system designs, the overall GuRoo's control system and the current knowledge of control systems. Outlined in the table below are the methodologies used for the various systems that that completed the overall active balance system.

System	Methodology
Sensor System	The sensor system will be implemented using a 3-axis gyro-meter and 3-axis inertia sensor. This provides more that adequate information to design the active balance system. This Inertial Measurement Unit or IMU has been kindly donated by CSIRO for the humanoid to use.

Table 1: Methodologies of the balance components.

System	Methodology
Control System	The control system will be designed using classical SISO methods and will be implemented on the torso motors of the robot.
Behaviour System	The behaviour system will keep the robot at the desired overall angle or posture required by various gaits and robot functions. Although this system is not designed using inertia sensors to counter the forces acting on the robot, it does implicitly help counter forces by keeping the robots centre of gravity within the support polygon and reducing the movement of the zero moment point.

Table 1 - Continued: Methodologies of the balance components.

3.4 CONSTRAINTS

Taking into account the GuRoo's overall design, hardware and software limitations, certain constraints were set at the start of the thesis project. These constraints are described in detail below:

1. The active balance control system has to be designed specifically for the control of the torso to head regions of the "GuRoo" robot to allow modularity between the specific control systems organised on the humanoid robot.
2. Sensor data was limited to retrieving angle inclinations due to the sensor inertia/acceleration readings being very noisy making them difficult to use in the control system.
3. Balance behaviour techniques are limited to torso, ankle and upper body movements to help simplify the control of the overall robot system and are adequate in order to help balance the robot.
4. Due to a bad motor encoder on the torso twist hardware board, the balance system will only be designed and implemented on the roll and pitch motors of GuRoo's torso.
5. The active balance system will be implemented in software, as the current design of the robot is fully software controlled and in addition allows ease and flexibility of the torso control for other systems.
6. The IMU sensor is to be positioned in the head of the robot to duplicate the human biological inertia and gyroscopic sensors.
7. Large disturbances will not be examined and designed for in this thesis due to the complexity involved and the time constraints.

3.5 CONTRIBUTIONS

This thesis will contribute to the overall balance system proposed by Damien Kee [2]. The active balance system will be used to allow the “GuRoo” humanoid robot to remain upright from external disturbances such as pushes and errors in joint position. This will aid the “GuRoo” robot in doing a series of tasks specified by the RoboCup competition. From the research and development of the active balance system once complete, it is hoped that this thesis paper will contribute to creating a team of humanoid robots that can beat the world champion soccer team.

CHAPTER 4

MODELLING THE HUMANOID ROBOT

This chapter will give a comparison of the three (3) main balance models and then specify the chosen model that will be used to describe the humanoid robot. Additionally this chapter will define the model parameters, formulate the dynamic differential equations and determine the state-space equations of the balance model that will be used in subsequent chapters.

4.1 CHOOSING A MODEL

4.1.1 COMPARING THE MODELS

Several models were described in the literature review section of this thesis. These models advantages and disadvantages are briefly outlined below:

Model	**Advantages**	**Disadvantages**
Inverted Pendulum Model [2]	Simplifies the analysis and calculation of dynamic motion.	Performance limitation and undershoot.
Gravity Compensated Inverted Pendulum [1]	Inertia of the legs is taken into account.	No joint reflections and dynamics taken into account.
Two Masses Inverted Pendulum Model	Models the two (2) main masses and the interaction between the masses. No undershoot limitation.	Missing knee and thigh joint dynamics of the robot. Only models leg as one unit.

Table 2: Balance Model Comparisons

4.1.2 THE CHOSEN MODEL

The model chosen to represent the "GuRoo" robot was that of the Two Masses Inverted Pendulum Model. This model gives an adequate amount of information about the dynamics of the robot and provides a stable platform in which to extend upon. It provides simplified dynamics whilst still taking into account the two (2) main masses of the humanoid robot. The model will also allow for future developments whereby control systems can be designed for the robots ankle.

4.2 TWO-MASS INVERTED PENDULUM MODEL

4.2.1 HUMAN ROBOT TRANSLATION

The two (2) mass inverted pendulum model can easily describe a 3D system by separately analysing two (2) 2D models, one modelling the pitch angles and the other modelling the roll angles of the robot's ankles and torso. As the two (2) main masses of the robot are the torso and legs, the upper rod will describe the movement of the torso and the lower rod will describe the movement of the legs. The middle joint of the 2D model is equivalent to the torso pitch or the torso roll joints of the robot. The lower joint will model both ankles pitch and roll joints with the assumption that both are moving at the same time and speed. The pitch and roll 2D models are shown below in Figure 9 and 10 respectively.

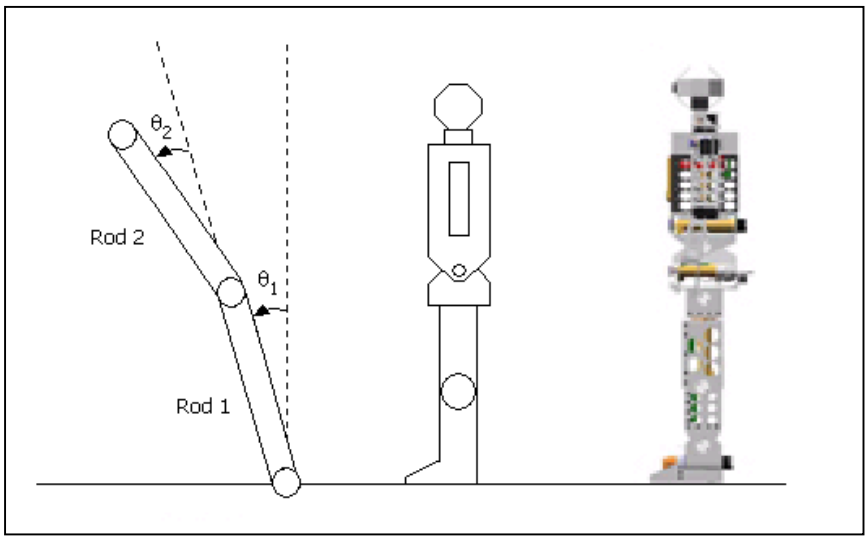


Figure 9: Pitch Model of Humanoid Robot.

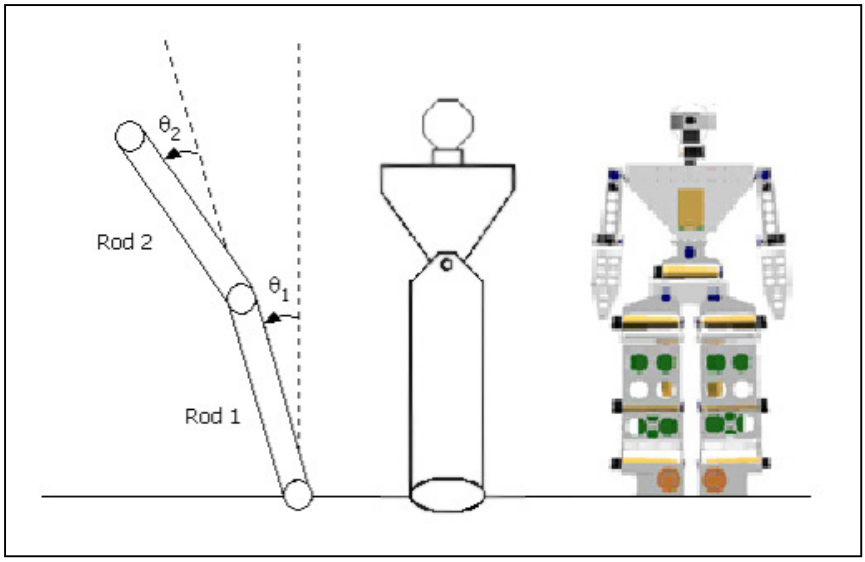


Figure 10: Roll Model of Humanoid Robot

The lower joint representing the ankles of the robot will be modelled a spring mass-damper system and should correctly model the robot if its feet do not leave the contact surface. The upper joint will be modelled as a simple mass-damper system to correctly represent the motor of the torso joint.

4.2.2 FORMULATING THE DYNAMIC DIFFERENTIAL EQUATIONS

In order to obtain the dynamic equations of the system, the models axes, angle directions and rod parameters need to be defined. Figure 11 shows the axes and positive direction for the angles torques, velocities and position. In addition to this, rod parameters for the model are given in Table 3 below.

Rod Number	Mass	Length	Spring Constant	Friction Coefficient
Rod 1	m_1	a_1	k_1	b_1
Rod 2	m_2	a_2	0	b_2

Table 3: Rod Parameters

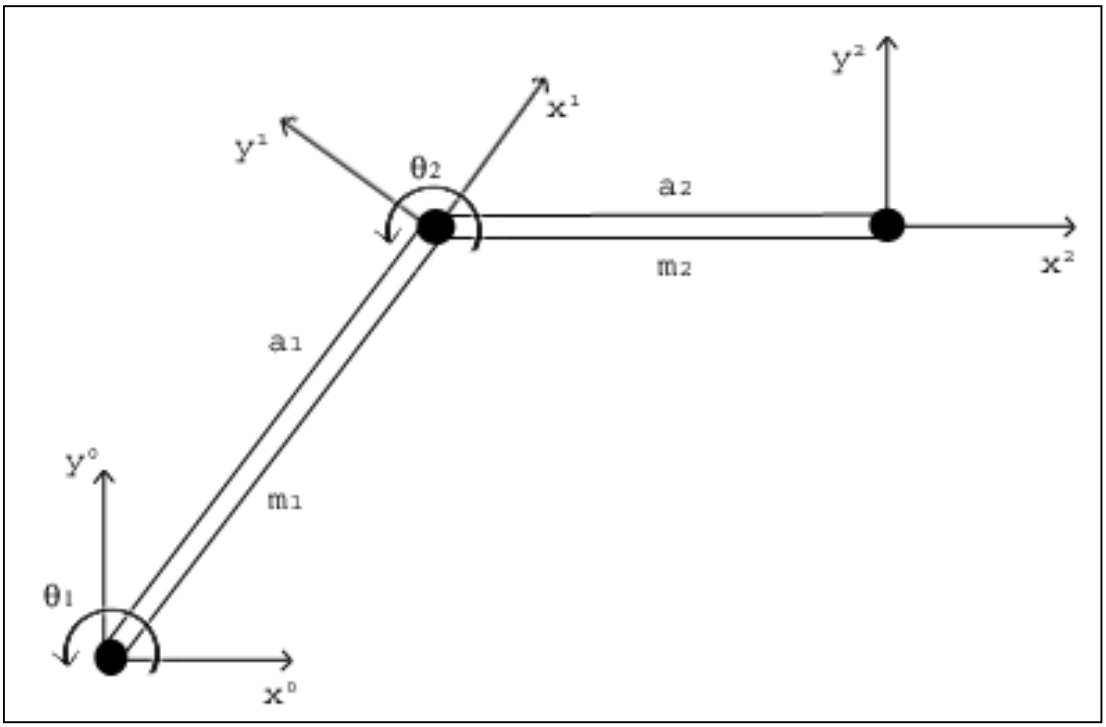


Figure 11: Model Parameters [1].

With the help of Schilling [1], the dynamic differential equations of motion for the two (2) mass inverted pendulum model was found. Thus in order to obtain these equations, the complete manipulator inertia tensor of the model must first be found. This is done by combining the Jacobian matrix for links and the inertia tensor of the individual links in base coordinates. The complete inertia tensor matrix for

the model is given in Equation 1 below. This tensor matrix combined with the joints velocity-coupling matrix, gravitational loading, spring force and damping is then used to formulate the dynamic model for the model. The total dynamic equations for joint 1 and 2 of the overall balance model are given below in Equation 2 and 3 respectively [1].

Inertia Tensor Matrix

$$D(q)=\begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix}$$

Matrix Values

$$D_{11}(q)=\left(\frac{m_1}{3}+m_2\right)a_1{}^2+m_2a_1a_2C_2+\frac{m_2a_2{}^2}{3}$$

$$D_{12}(q)=D_{21}(q)=\frac{m_2a_1a_2C_2}{2}+\frac{m_2a_2{}^2}{3}$$

$$D_{22}(q)=\frac{m_2a_2{}^2}{3}$$, where C_2 is Cosine (θ_2).

Equation 1: Complete Inertia Tensor Matrix Equations [1].

$$t_1=\left[\left(\frac{m_1}{3}+m_2\right)a_1{}^2+m_2a_1a_2C_2+\frac{m_2a_2{}^2}{3}\right]\ddot{q}_1+\left(\frac{m_2a_1a_2C_2}{2}+\frac{m_2a_2{}^2}{3}\right)\ddot{q}_2-m_2a_1a_2S_2\left(\dot{q}_1\dot{q}_2+\frac{\dot{q}_2{}^2}{2}\right)$$
$$+g_0\left[\left(\frac{m_1}{2}+m_2\right)a_1C_1+\frac{m_2a_2C_{12}}{2}\right]+b_1(\dot{q}_1)+k_1(q_1)$$

Equation 2: Balance Model Joint 1 - Dynamic Equation [1].

$$t_2=\left[\frac{m_2a_1a_2C_2}{2}+\frac{m_2a_2{}^2}{3}\right]\ddot{q}_1+\frac{m_2a_2{}^2\ddot{q}_2}{3}+\frac{m_2a_1a_2S_2\dot{q}_1{}^2}{2}$$
$$+\frac{g_0m_2a_2C_{12}}{2}+b_2(\dot{q}_2)$$

Equation 3: Balance Model Joint 2 - Dynamic Equation [1].

Where t_1 = joint 1 torque, t_2 = joint 2 torque, C_{12} = Cos(q_1 + q_2), S_2 = Sin(q_2), C_2 = Cos(q_2), g_0 = gravity, q_1 = angular position about joint 1 and q_2 = angular position about joint 2.

4.2.3 BALANCE MODEL STATE-SPACE EQUATIONS

For ease of software simulation and computing purposes, the state-space equations of the dynamic equations will be formulated. Using the previously defined equations 1,2 and 3, the state-space equations can then be found using the proposition shown in Equation 5 from Schilling [1] below. Thus by defining the states as shown in Equation 4, the dynamic equations can be represented by the state-space model shown below in Equation 5.

$$x^T = \left[q^T, v^T\right], \quad \text{where } v = \dot{q}$$

Equation 4: State Definitions.

$$\dot{q} = v$$

$$\dot{v} = D^{-1}(q)\left[\tau - h(q) - c(q,v) - b(v) - k(q)\right]$$

$$y = Cx$$

Equation 5: Proposition State Space Representation. [1]

Where C = $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$, τ = torque input, $h(q)$ = the gravitational loading, $c(q,v)$ = the coriolis and centrifugal forces and $b(v)$ = the friction or damping loading.

Therefore examining Equation 5, in order to evaluate the state-space equations, the inverse of the inertia tensor matrix is found using simple matrix inversion techniques and is defined below in Equation 6.

$$D^{-1}(q) = \frac{1}{\Delta(q)}\begin{bmatrix} D_{22} & -D_{12} \\ -D_{21} & D_{11} \end{bmatrix}$$

Equation 6: Inverse Inertia Tensor Matrix [1].

Where D_{11}, D_{12}, D_{21} and D_{22} are defined in Equation 1.

Furthermore the determinant $\Delta(q)$ of the inertia tensor matrix is given in Equation 7.

$$\Delta(q) = D_{11}D_{22} - D_{12}D_{21} = m_2(a_1a_2)^2\left[\frac{m_1}{9} + m_2\left(\frac{1}{3} - \frac{{C_2}^2}{4}\right)\right]$$

Equation 7: Determinant of Inertia Tensor Matrix [1].

By combining Equations 4, 5, 6 and 7, the completed state-space equations can be evaluated. The complete state-space equations of the model are shown in Equation 8 below.

$$\dot{q}_1 = v_1$$

$$\dot{q}_2 = v_2$$

$$\dot{v}_1 = \frac{1}{\Delta(q)}\left(\begin{array}{l} D_{22}(q)\left\{t_1 + m_2a_1a_2S_2\left(v_1v_2 + \frac{{v_2}^2}{2}\right) - g_0\left[\left(\frac{m_1}{2} + m_2\right)a_1C_1 + \frac{m_2a_2C_{12}}{2}\right] - b_1(v_1) - k_1(q_1)\right\} - \\ D_{12}(q)\left[t_2 - \frac{m_2a_2a_1S_2{v_1}^2}{2} - \frac{g_0m_2a_2C_{12}}{2} - b_2(v_2)\right] \end{array}\right)$$

$$\dot{v}_2 = \frac{1}{\Delta(q)}\left(\begin{array}{l} -D_{21}(q)\left\{t_1 + m_2a_1a_2S_2\left(v_1v_2 + \frac{{v_2}^2}{2}\right) - g_0\left[\left(\frac{m_1}{2} + m_2\right)a_1C_1 + \frac{m_2a_2C_{12}}{2}\right] - b_1(v_1) - k_1(q_1)\right\} + \\ D_{11}(q)\left[t_2 - \frac{m_2a_2a_1S_2{v_1}^2}{2} - \frac{g_0m_2a_2C_{12}}{2} - b_2(v_2)\right] \end{array}\right)$$

Equation 8: State Space Equations of Balance Model [1].

Where t_1 = joint 1 torque, t_2 = joint 2 torque, $C_{12} = \text{Cos}(q_1 + q_2)$, $S_2 = \text{Sin}(q_2)$, $C_2 = \text{Cos}(q_2)$, g_0 = gravity, q_1 = angular position about joint 1, q_2 = angular position about joint 2, v_1 = velocity about joint 1 and v_2 = velocity about joint 2.

CHAPTER 5

BALANCE CONTROL SYSTEM

This chapter will describe the method of designing the balance control system for the humanoid robot's torso joint. Firstly the SISO transfer function of the system is formulated by using MATLAB and linearising the system. A control system will then be designed using classical techniques and the design will be implemented along with motor characteristics and limits. Finally, by using MATLAB Simulink, the results of the linear vs. non-linear systems are compared.

5.1 OBTAINING THE TRANSFER FUNCTIONS

Examining the state-space equations given in Equation 8, the balance model is found to be a non-linear system. In order to obtain the transfer functions of the system, these state-space equations are linearised about the equilibrium point into four (4) transfer functions, torque 1 to angle 1, torque 2 to angle 2, torque 1 to angle 2 and torque 2 to angle 2. The system can then be converted from a Multiple Input Multiple Output (MIMO) system to a Single Input Single Output (SISO) system by considering inputs into the lower joint (torque 1) as disturbances, as the balance system will only control the torso joint (torque 2). These steps are detailed in the following sections.

5.1.1 LINEARISING THE SYSTEM

As the control system will be designed using SISO techniques, the balance model is required to be linearised in order to obtain the systems transfer functions. The equilibrium points of the balance model is defined as the upright position of the two-mass inverted pendulum model in accordance with Figure 11. Therefore the system will be linearised about the points $q_1 = \pi/2$ and $q_2 = 0$. Now by letting q_1 and q_2 be perturbed about the equilibrium points stated above, the system can be linearised using the terms shown in Equation 9.

$$q_1 = \pi/2 + \delta q_1$$

$$q_2 = 0 + \delta q_2$$

Equation 9: Equilibrium Points.

Therefore using common mathematical linearisation techniques, the non-linear cosine and sine terms can then be evaluated into linear terms. This is shown in Table 4.

Non-Linear	Linear
Cos ($\pi/2 + \delta q_1$)	$-\delta q_1$
Cos ($0 + \delta q_2$)	1
Cos ($\pi/2 + \delta q_1 + 0 + \delta q_2$)	$-(\delta q_1 + \delta q_2)$
Sin ($0 + \delta q_2$)	0

Table 4: Linearising Function Table.

Replacing the non-linear terms of the state-space equation with the linear terms derived in Table 4, the linearised state-space equations can then be formulated. These equations can then be programmed into MATLAB to produce the four transfer functions of the linearised system. The complete linearised state-space equations are shown in Equation 10, 11 and 12.

$$\delta\ddot{q}_1 = \delta v_1$$
$$\delta\ddot{q}_2 = \delta v_2$$

$$\delta\ddot{v}_1 = \frac{1}{\Delta(q)}\begin{pmatrix} D_{22}(q)\left\{t_1 + g_0\left(\frac{m_1}{2} + m_2\right)a_1\delta q_1 + \frac{g_0 m_2 a_2 \delta q_1}{2} + \frac{g_0 m_2 a_2 \delta q_2}{2} + b_1(\delta v_1) - k_1(\delta q_1)\right\} - \\ D_{12}(q)\left\{t_2 + \frac{g_0 m_2 a_2 \delta q_1}{2} + \frac{g_0 m_2 a_2 \delta q_2}{2} - b_2(\delta v_2)\right\} \end{pmatrix}$$

$$\delta\ddot{v}_2 = \frac{1}{\Delta(q)}\begin{pmatrix} -D_{21}(q)\left\{t_1 + g_0\left(\frac{m_1}{2} + m_2\right)a_1\delta q_1 + \frac{g_0 m_2 a_2 \delta q_1}{2} + \frac{g_0 m_2 a_2 \delta q_2}{2} + b_1(\delta v_1) - k_1(\delta q_1)\right\} + \\ D_{11}(q)\left\{t_2 + \frac{g_0 m_2 a_2 \delta q_1}{2} + \frac{g_0 m_2 a_2 \delta q_2}{2} - b_2(\delta v_2)\right\} \end{pmatrix}$$

Equation 10: Linearised State-Space Equations

$$\Delta(q) = D_{11}D_{22} - D_{12}D_{21} = m_2(a_1 a_2)^2\left[\frac{m_1}{9} + m_2\left(\frac{1}{12}\right)\right]$$

Equation 11: Linearised Determinant Equation.

$$D_{11}(q) = \left(\frac{m_1}{3} + m_2\right) a_1^{\,2} + m_2 a_1 a_2 + \frac{m_2 a_2^{\,2}}{3}$$

$$D_{12}(q) = D_{21}(q) = \frac{m_2 a_1 a_2}{2} + \frac{m_2 a_2^{\,2}}{3}$$

$$D_{22}(q) = \frac{m_2 a_2^{\,2}}{3}$$

Equation 12: Linearised Inertia Matrix

5.1.2 CONVERTING TO A SISO SYSTEM

To convert the balance model from a MIMO system to a SISO system, the four transfer functions and their relationship must first be defined. By using the MATLAB command and the linearised state-space equations developed in Section 5.1.1, the transfer functions can be found. Since the *ss(A,B,C,D)* command can only take in numeric values, the model constants must first be defined. These constants are shown in Table 5.

Description	Constant	Value
Gravity	g	9.81 m/s^2
Rod 1 Mass	m_1	22.967 Kg
Rod 2 Mass	m_2	13.653 Kg
Rod 1 Length	a_1	0.7 m
Rod 2 Length	a_2	0.5 m
Joint 1 Damping	b_1	10 N/rad/s
Joint 2 Damping	b_2	0.1 N/rad/s
Joint 1 Spring	k_1= (g*(m_1/2+m_2)*a1)	172.6123 N/m

Table 5: Table of Model Constants.

The balance system will only be designed to use the torso joint of the humanoid robot, as current joint control system on the robot will keep joint 1 at a constant position. A value of 10 N/rad/s for the Joint 1 (ankle) damping factor was estimated to be the factor produced by the current joint control system. Errors in joint positions were modelled using Joint 1 spring factor and was set to keep the equilibrium

point of the humanoid robot at 0.1 rads, thus a value of $k_1 = 172.6123$ was calculated to cancel gravity at this point.

By using these defined model constants and MATLAB, the four transfer functions that describe the linearised balance model are given in Equations 13 and 14. The m-file used to find the transfer functions can be found in Appendix A.

```
Transfer function from input "Torque 1" to output...
                    0.1844 s^2 + 0.01621 s - 5.426
 Angle 1:  ---------------------------------------------------     TF1
           s^4 + 2.109 s^3 - 56.49 s^2 - 54.8 s + 6.475e-014

                   -0.5716 s^2 + 5.076e-016 s + 5.426
 Angle 2:  ---------------------------------------------------     TF2
           s^4 + 2.109 s^3 - 56.49 s^2 - 54.8 s + 6.475e-014
```

Equation 13: TF from Torque 1 to Output.

```
Transfer function from input "Torque 2" to output...
                  -0.5716 s^2 + 1.777e-015 s + 5.426
 Angle 1:  ---------------------------------------------------     TF3
           s^4 + 2.109 s^3 - 56.49 s^2 - 54.8 s + 6.475e-014

                     2.651 s^2 + 1.621 s - 5.426
 Angle 2:  ---------------------------------------------------     TF4
           s^4 + 2.109 s^3 - 56.49 s^2 - 54.8 s + 6.475e-014
```

Equation 14: TF from Torque 2 to Output.

Using these transfer functions, the balance model dynamics can be described by adding the corresponding SISO transfer functions. The overall angle (q_3) of the model is the simply the sum of joint 1 and joint 2. This is shown in Figure 12.

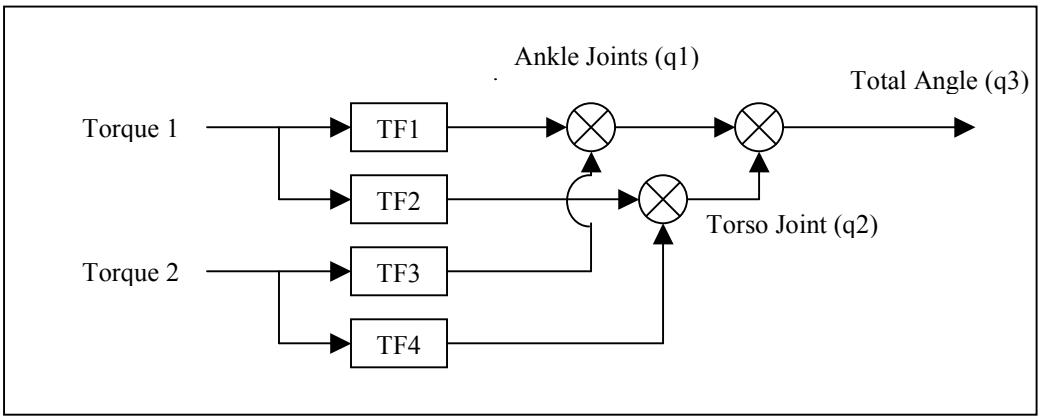


Figure 12: Linearised MISO System Diagram.

By examining Figure 12, it can be seen that Torque 1 through its transfer functions can be considered as a disturbance into the angles produced by Torque 2. Therefore to simplify the system from a MIMO system to a SISO system, Torque 1 effects on the joints will be consider as disturbances. Figure 13 shows the system in its simplified SISO form.

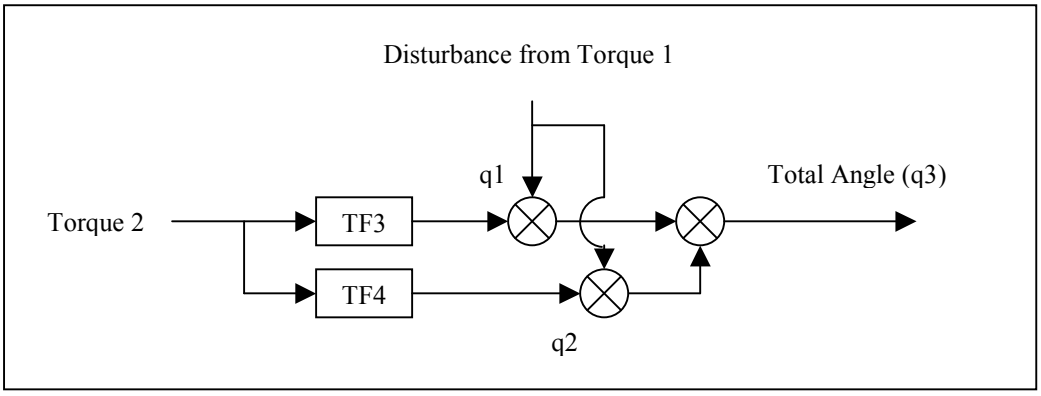


Figure 13: SISO System Diagram.

5.2 CONTROL SYSTEM DESIGN

The proposed control system for the balance model was that of a gravity and damping compensated PID controller that produces an error signal according to the overall desired angle versus the actual angle. In addition motor characteristics were modelled so that effects of motor limits could be examined. These steps are outlined in the sections below.

5.2.1 ROOT LOCUS DESIGN

Root locus techniques were mainly used to find zeros and poles that would stabilise the balance system. As the system was greater than second order, peak times, settling times and overshoot would not be as predicted by second order calculations. To design the PID compensator, MATLAB SISO tool was used and the root locus examined. MATLAB code used to find the root locus of the gravity and damping compensated system is attached in Appendix A.

Examining the original root locus diagram, a steady state error compensator of

$$\frac{s+0.5}{s}$$

was introduced to the root locus. This increased the system type and thus would eliminate any steady-

state error. Additionally a real zero was added at –10 to shift the root locus towards the left hand plane. This is shown in Figure 14 with the left root locus being the uncompensated system and the right root locus being the compensated system. The complete compensator is given in Figure 15.

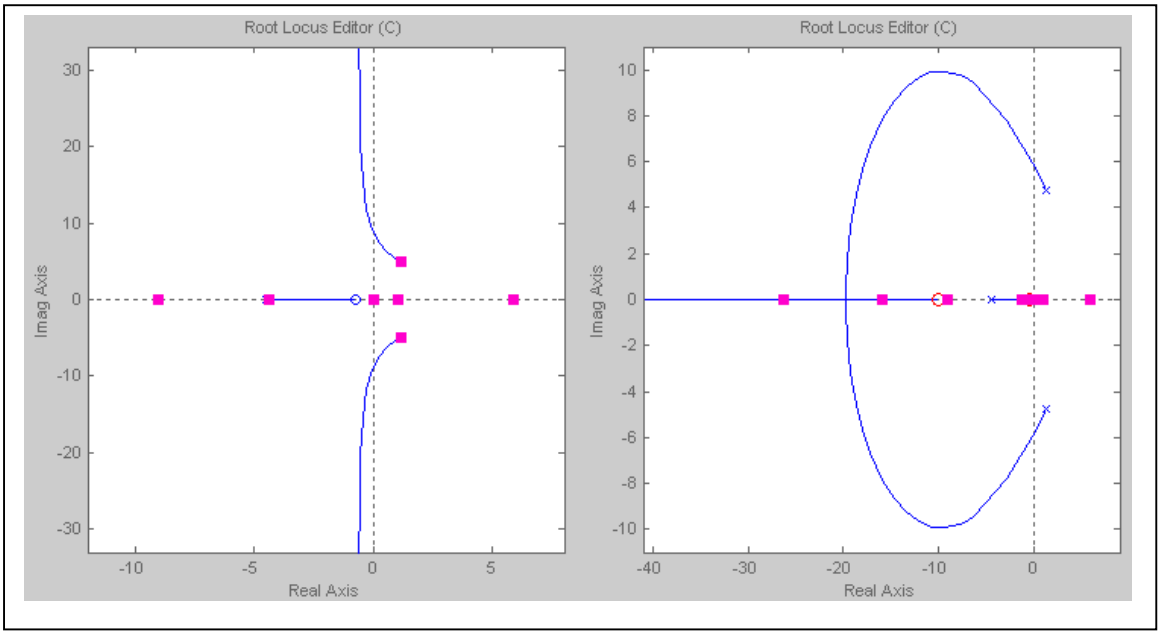


Figure 14: Root Locus Design. Left – Original System, Right – Compensated System

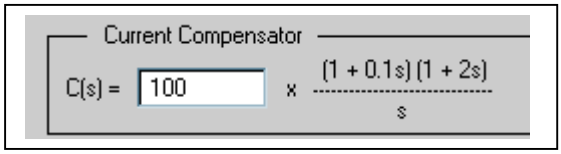


Figure 15: PID Compensator

After evaluating the compensator to produce the P, I and D gains shown in Table 6, these gains were then simulated on the linear and non-linear dynamic systems in MATLAB Simulink to examine suitability of gains and evaluate overshoot and time response of the system. This will be observed in section 5.3.

Gain Type	Value
Proportional (P)	210
Integral (I)	100
Derivative (D)	20

Table 6: Table of PID Controller Gains.

5.2.2 MODELLING MOTOR CHARACTERISTICS

In order to investigate the motor effects of the real robot system, the robots motor characteristics were obtained from the Maxon motor data sheets attached in Appendix C and were modelled in Simulink. The GuRoo joints are all controlled using the same specification motor, the RE 36 – Order Number 118799 and are all put through a gear ratio of 1:156 with a 0.7 efficiency rating. The standard DC motor equation is given below in Equation 16.

$$\frac{t_{out} r_a}{k_a} + k_b \omega = E_a$$

Equation 16: Motor Equation

Where t_{out} = torque output from motor, r_a = terminal resistance, k_a = torque constant, k_b= back EMF constant, ω = speed of motor and E_a = input voltage into motor.

Based on the motor driver limit of 4A, the maximum torque that could be outputted from the motors can be calculated as follows:

$$\begin{aligned} Maximum\ Torque &= k_m \times n_g \times \eta_g \times I \\ &= 0.0445 \times 156 \times 0.72 \times 4 \\ &= 19.99\,Nm \end{aligned}$$

Equation 17: Max Torque Calculation [2]

where k_m is the motors torque constant, n_g is the gear ratio, η_g is the gear efficiency and I is the maximum current outputted by the motor drivers.

Therefore to implement voltage and torque saturation limits, Equation 16 was used to find the desired voltage or pulse width modulation signal into be inputted to the system. This input was then limited to the nominal voltage of the motor, ±32V before being put into the motor. The equation was then used backwards to reformulate the torque outputted from the motor and this value limited to ±0.0178Nm before the gear ratio and thus effectively current limiting the motor to 4A or torque by the motor to 20Nm maximum.

Hence using this maximum torque limit calculated in Equation 17 and the nominal voltage specified by the Maxon datasheet, motor limits were then modeled into MATLAB shown in Figure 4. Furthermore key motor characteristic constants are stated in Table 7 below.

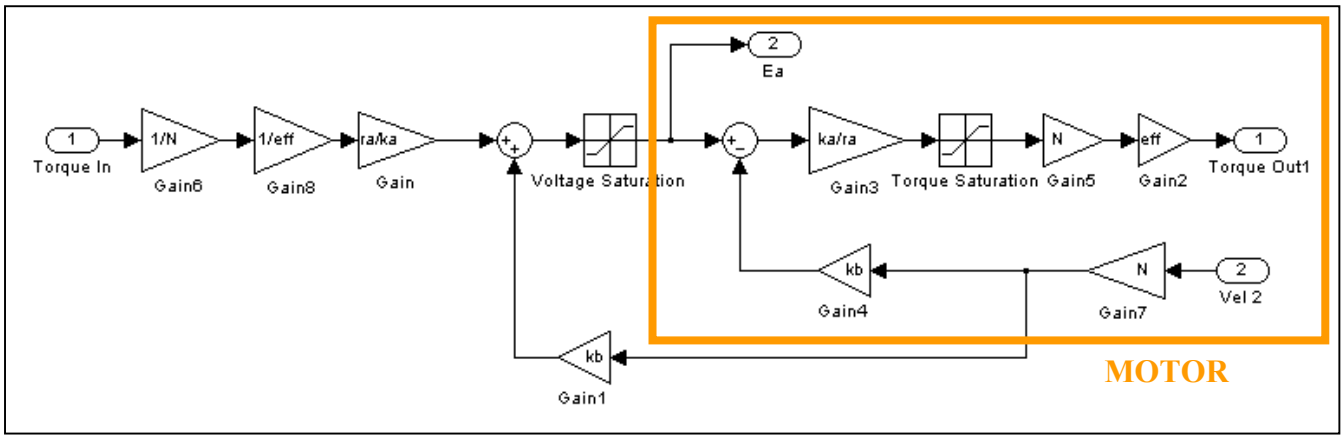


Figure 4: Motor Limits in Simulink

Name	Symbol	Value	Units
Gear Ratio	N	156	N/A
Gear Efficiency	eff	0.7	N/A
Torque Constant	k_a	0.0445	Nm/A
Back EMF Constant	k_b	0.0444153	V/rad/s
Torque Saturation	N/A	± 0.178	Nm
Voltage Saturation	N/A	± 32	Volts
Terminal Resistance	r_a	1.71	Ohms

Table 7: Motor Characteristic Values

With the complete motor limits, PID controller, gravity and damping compensation in place, the complete control system diagram in Simulink for the non-linear model is given in Figure 16. Please refer to Appendix B for the Simulink diagram of the linear model and m-files for both linear and non-linear dynamic simulation.

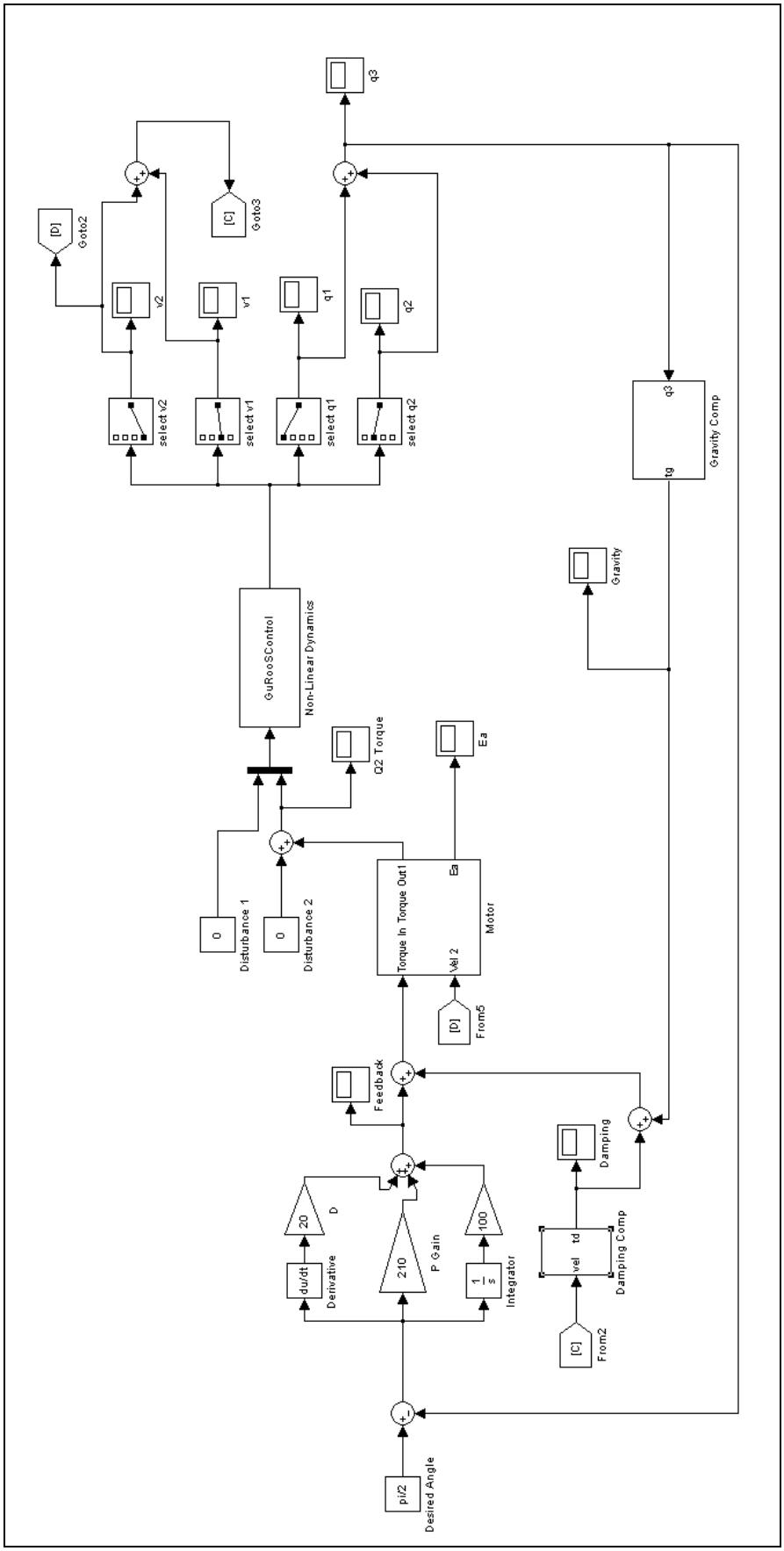


Figure 16: Simulink Diagram of PID, Gravity and Damping Controller

5.3 MATLAB SIMULATION RESULTS

The overall control system was implemented in MATLAB Simulink and simulated with graphs of the non-linear and linear time responses being examined. Some additional observations were also conduced and refinements were made to the control system to reduce complexity by removing components that produced negligible differences.

5.3.1 LINEAR AND NON-LINEAR TIME RESPONSES

The Simulink model shown in Figure 16 and the S-Function Block were used to simulate the dynamics of the balance model. The initial conditions of joint 1 being offset 0.1 rads and joint 2 being set at 0 rads was given to the system. Graphs of the linear versus non-linear balance model shown in Figure 17 were obtained and examined.

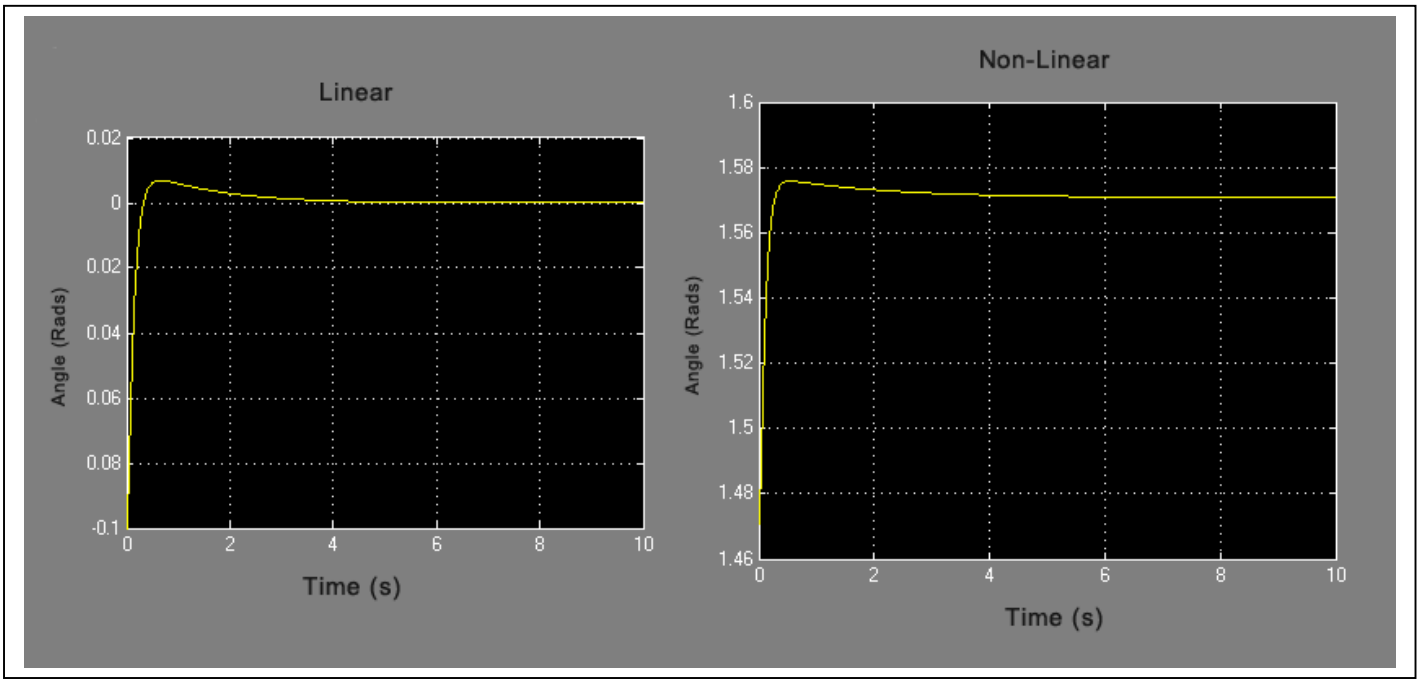


Figure 17: Control System Time Responses
Left – Linear Time Response, Right – Non Linear Time Response

From the graphs shown in Figure 17, it can be seen that the linear model is almost exactly the same as the non-linear model for relatively small movements around the equilibrium point. Differences in starting position stem from the linear model being linearised around an upright equilibrium point whereas the non-linear model follows the parameters set by Figure 11.

Investigating Figure 17 in much more detail, it can be seen that a response or peak time of the control system is around 0.5 seconds with an overshoot of 0.33%. This response was found to be stable and adequate for the desired system response.

5.3.2 REFINEMENTS TO THE DESIGN

Using the balance control system simulation results from MATLAB, the damping compensation was observed to provide very little compensation to the system. This was due to the very low damping coefficient in the model parameters and thus to avoid complexity in the control system, damping compensation was removed from the control system. Thus the new design for the control system consisted of a PID and gravity compensated control system.

CHAPTER 6

ATTITUDE BEHAVIOURS

In order for the robot to react to disturbances once they are detected, attitude behaviours must be examined and implemented. In this thesis, two main methods of behaviours were investigated. These methods being posture control and centre of gravity (COG) control. These attitudes are described in detail below.

6.1 POSTURE ATTITUDE CONTROL

Posture control of the humanoid robot is to simply keep the robot in a desired posture or position. In the case of the balance control system, the desired inclination of the torso is the desired posture of the robot. By keeping the posture of the robot upright, the robot is in a position that is much more stable and increases manoeuvrability to counter external disturbances. In addition to keeping the humanoid robot in a more stable position, the posture of the humanoid robot is also aesthetically pleasing and allows the robot to look more human-like when performing desired tasks.

6.2 COG ATTITUDE CONTROL

The centre of gravity attitude approach looks at keeping the robots centre of gravity at a desired position located within the support polygon. By keeping the COG within this support polygon, the robot is in a stable position. Using the two mass inverted pendulum balance model with point masses located at the centre of each rod, the overall inclination of the robot through the IMU sensor and the motor position of the torso using the motor encoder feedback, the lower joint of the robots model can be calculated. Using this value and the model parameters stated in Figure 11, the desired torso angle could then be calculated using trigonometry techniques in order to keep the COG of the robot in its desired location. Calculations for the desired inclination are shown in Equation 18 and 19.

$$\theta_1 = lowerangle = \theta_3 - \theta_2$$

Equation 18: COG Angle Calculations

where θ_3 = overall angle from IMU sensor (upright is 0 rad) and θ_2 = torso angle from motor encoders.

$$\theta_{2-Desired} = \frac{\pi}{2} - \arccos\left[\frac{2}{a_2}\left\{a_1 \cos\left(\frac{\pi}{2} - \theta_1\right) - DesCOG\right\} + \frac{m_1}{m_2}\left\{\frac{a_1}{2}\cos\left(\frac{\pi}{2} - \theta_1\right) - DesCOG\right\}\right.$$

Equation 19: Desired Inclination Using COG

where $\theta_{2\text{-Desired}}$ = desired inclination for robot, θ_1 = lower angle or joint of robot and DesCog = desired COG point relative to the axis shown in Figure 11. Additionally model parameters such as a_1, a_2, m_1 and m_2 are given in Table 5.

CHAPTER 7

SOFTWARE DESIGN AND IMPLEMENTATION

7.1 SIMULATION USING DYNAMECHS

The GuRoo project utilises the Dynamechs package developed by Scott McMillan [12] to simulate the robots dynamics and interactions between components and the environment. This package combined with the GuRoo code can be found electronically attached in Appendix E. In order to simulate the active balance system design, two (2) key components, the IMU sensor and simulated forces were required to be implemented in the simulation program.

7.1.1 CURRENT SIMULATION DESIGN

The current software simulation design is made up of two main files, *humanoid.cpp* and *lowlevel_sim.cpp*. The *lowlevel_sim.ccp* file contains all the interactions with the Dynamechs simulation files, which are implemented using object orientation techniques. This allows users the flexibility to construct any robot from aqua robots to spider robots. The objects provide a range of very useful functions and constants specific to each object such as masses, joint velocities, joint positions and link rotational matrices. The objects also provide functions to input various information into the object if applicable. For instance, the link *i* can have an input PWM to make it rotate around its joint.

The *humanoid.cpp* file contains all the implementation simulation such as delays in control loops and as well as simulating the can bus, board code and keeping various log files. *humanoid.cpp* calls various *lowlevel_sim.cpp* functions when required and also calls the Master-Slave update function to simulate the transfer of data from the balance system to the robot. A diagram showing this is given in Figure 18.

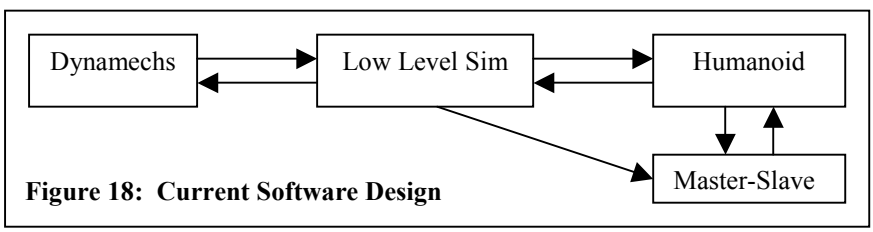


Figure 18: Current Software Design

7.1.2 IMU SENSOR DESIGN

For the IMU sensor to be implemented within the GuRoo simulation package, information from the Dynamechs package had to be extracted. To keep the simulation of the IMU as close as possible to the actual sensor, velocities from the rotation of the head were extracted from Dynamechs, these velocities were then filtered using a rolling average to smooth out the values. This function was run as fast as possible to give accurate velocities so that they can be integrated to provide correct inclinations. Whilst this function runs, another function was used to sample the velocities and integrate the velocities at a rate of the central speed (50Hz) as the sensor provides inclination information at this speed. This was implemented into the GuRoo simulator and tested to see if data was accurate and correct. The simulated IMU information was implemented to reflect the exact same information produced by the real IMU sensor and thus negative pitch and roll angles were produced when the robot leans forwards and leans left respectively. Graphs of the actual IMU sensor pitch angle versus the simulated IMU sensor pitch angle are shown in Figure 19 below.

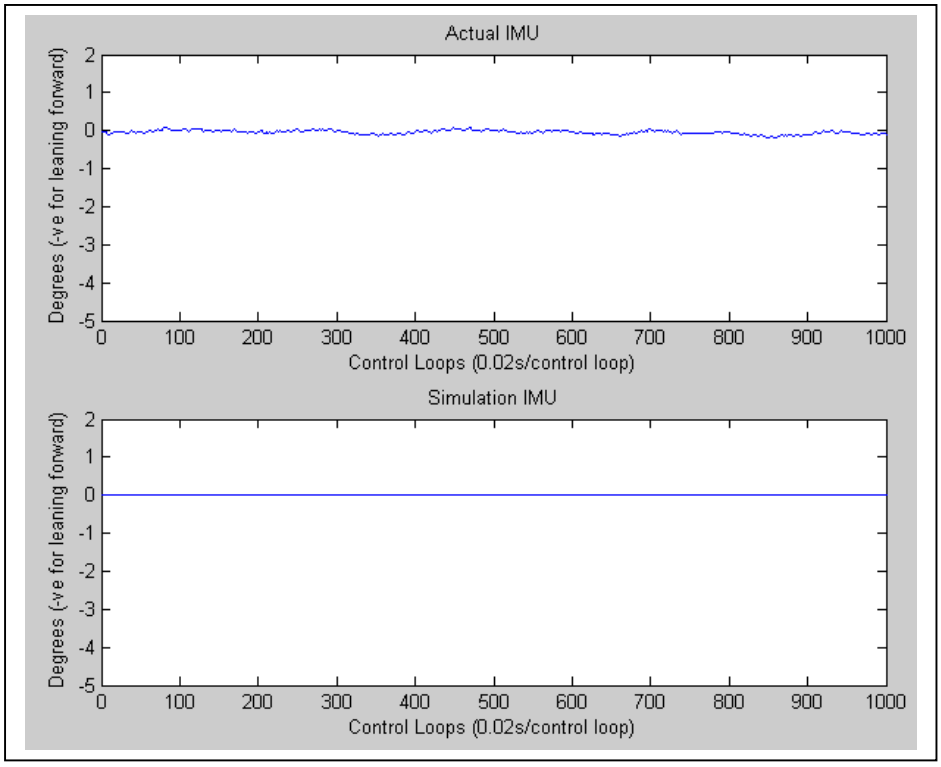


Figure 19: Actual vs. Simulation IMU

Examining Figure 19, the data from the simulation package Dynamechs is very accurate and thus the simulation IMU is held at zero (0) with only a very slight drift. Comparing the actual vs. simulated

IMU information, it can be seen that the actual IMU jitters around the zero (0) degree mark. For the purpose of simulation these slight differences were acceptable and the IMU simulation data found to be more than adequate. An overall diagram showing the interactions between the IMU sensor functions is given below in Figure 20. Software code for the IMU sensor implementation is given in Appendix D.

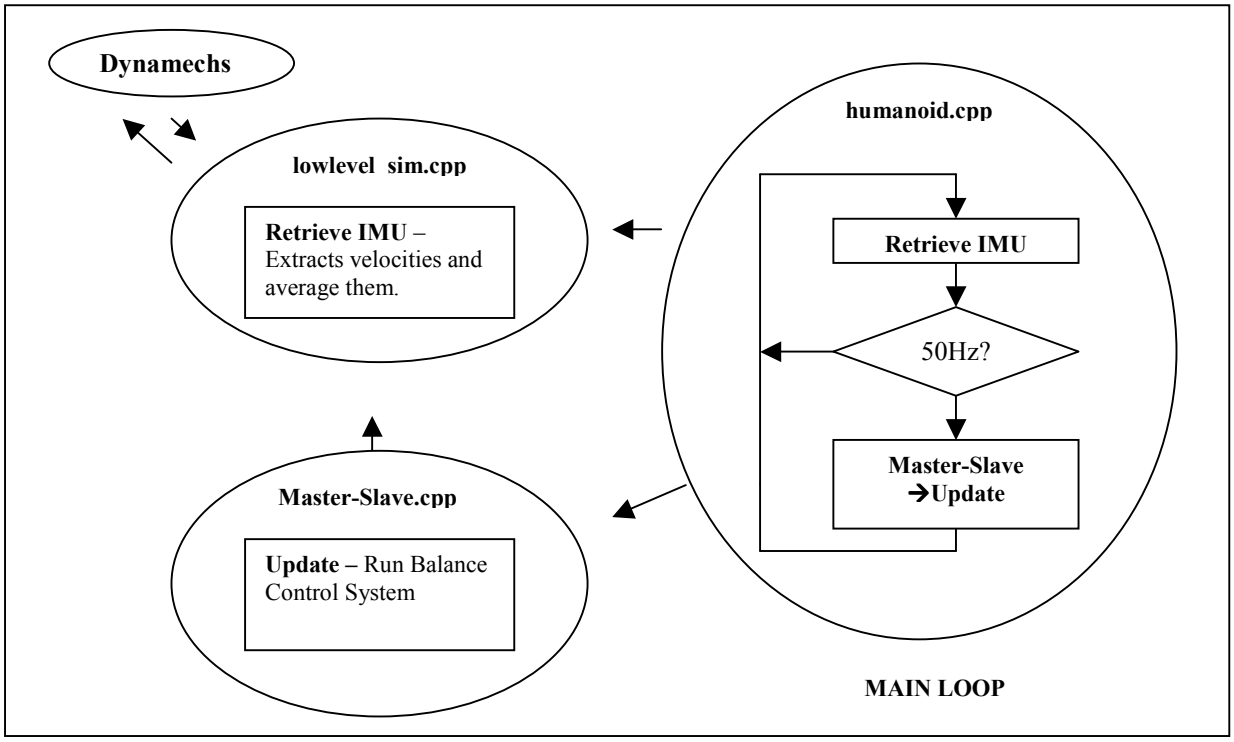


Figure 20: Flowchart Diagram for IMU simulation implementation.

7.1.3 SIMULATION INPUT FORCE DESIGN

For the simulation implementation of the force input to the robot, Dynamechs provides the function to set an input force vector to a link which consists of an moment about x, moment about y, moment about z, x, y and z with respect to the link coordinates. This function is located in the *dmRigidBody* code and is accessed through *rigidBody -> setExternalForce(your force vector)*. This force design was implemented in the *lowlevel_sim.cpp* part of the program, as access to Dynamechs was required. A slave or control class for the system was created to allow custom forces to be implemented in the simulator at various gait stages. A diagram showing the interaction between the force design and the current software system is shown below in Figure 21 with code implementation of the force design attached in Appendix D. For more information regarding the Master-Slave system software implementation and setup, please refer to the thesis paper "Gate Generation for a Humanoid" by Tim Pike [15].

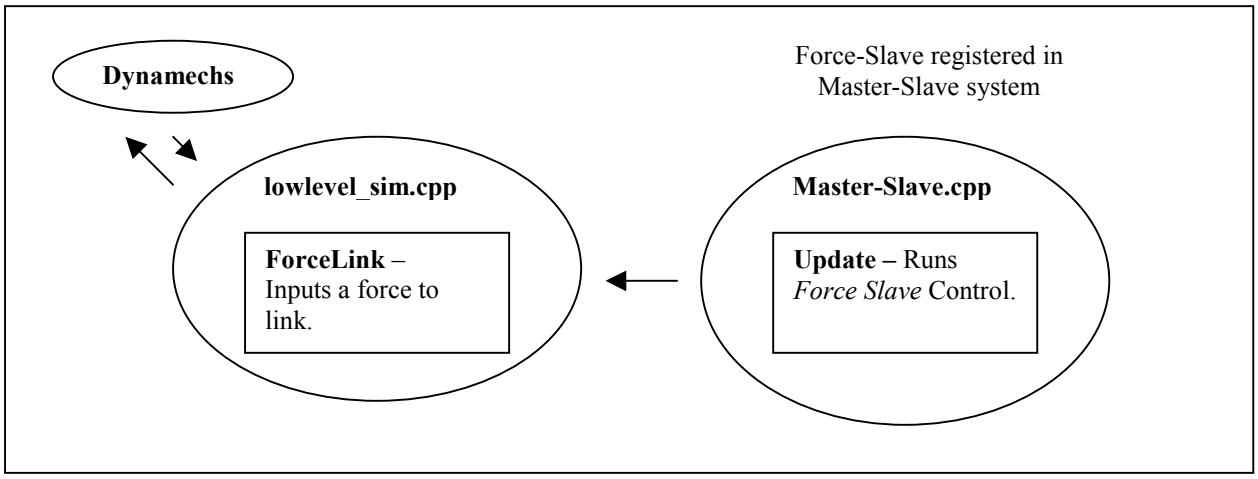


Figure 21: Flowchart Diagram for Force simulation implementation.

7.2 SOFTWARE DESIGN METHODS

The software for the balance control system was designed using two methods, the lower level design and higher-level design. The lower level plan was first designed, as the original implementation idea was to control the pulse width modulation (PWM) signal of the motor and thus implement the control system on the DSP boards. This allowed the system to be run at a much faster speed but it was found that conflicts would occur with the original motor control system already implemented on the DSP boards. This lower level design was then converted to a higher-level design using simple motor characteristics and whereby velocities were sent to the robots joint via a serial cable. Control of these joints through velocities is determined by the Master-Slave system implemented by Tim Pike [15]. The higher-level design is described below.

7.2.1 HIGHER LEVEL DESIGN

The active balance system has been designed and implemented at a slave of the Master-Slave system. This slave becomes registered on a process keyboard button 'b' and once registered, can control the torso joints of the robot through the function *master->modifyJointVel(TORSO_FWD, Velocity, JOINT_MODIFY_ADD)*. This function specifies a velocity to a particular joint and that velocity is updated at a rate of 50Hz. To extract the IMU data from simulation, the *IMUInfo* function is called that returns the pitch and roll information extracted from Dynamechs described in Section 7.1.2. On the other hand, if the real IMU information is required, a shared memory array is simply accessed and the values converted to rads.

This angle information either from the real or simulated IMU is then compared to the set desired overall angle and an error signal produced accordingly. Additionally a derivative, integral and compensation signal is produced using PID compensation in software techniques from Gordon's ELEC3500 lecture notes [16]. Since the compensation signal is in units of torque due to the gains being produced in MATLAB simulation, the compensation signal first has to be converted into velocities that can be sent to the GuRoo joints.

To do this, the torque compensation signal was converted into a voltage signal (E_a) to be inputted into the motor using the motor Equation 16. Examining Equation 16, the current velocity of the joint is also required in order to produce the desired voltage signal. These velocities are stored in shared memory on the computer running the software program and thus could be easily accessed. The last step was to convert the voltage signal into a velocity and therefore by using the speed constant of the motor, the desired velocity could be specified into the GuRoo joints via the Master-Slave system. A block diagram describing the active balance system software design and operation is given in Figure 22. Actual software code for the balance control slave is given in Appendix D.

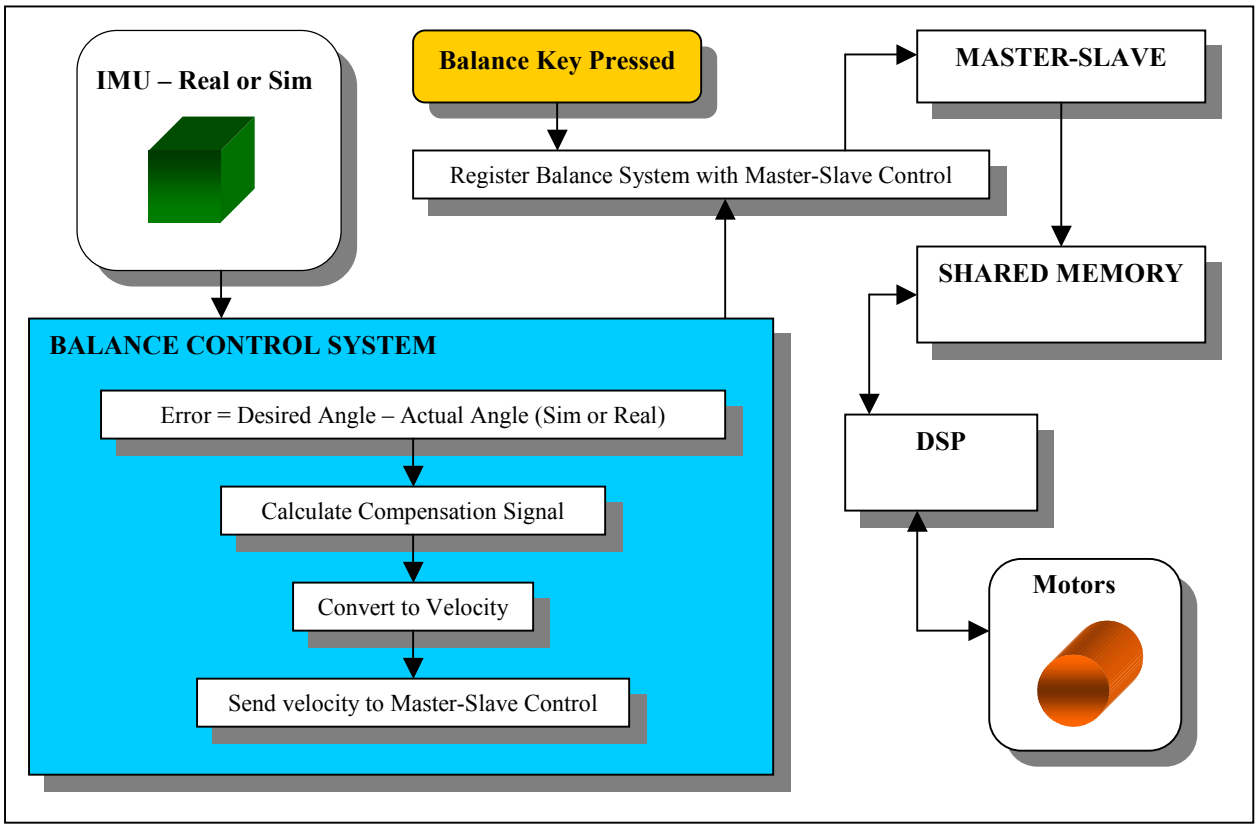


Figure 22: Block Diagram for Balance Control System Operation.

CHAPTER 8

SIMULATION RESULTS AND REFINEMENTS

8.1 PRELIMINARY SIMULATION RESULTS

Using the software implementation described in the previous chapters, the active balance system was simulated using the GuRoo simulator. The original gains of P = 210, D = 20 and I = 100 were inputted into the balance control system and log files generated. An analysis of the GuRoo simulation using these gains showed that the overshoot was too prominent and caused the humanoid robot to sway and take to long to stabilise. The log files were examined and the integral gain found to cause the overshoot. Thus an integral gain of 10 was chosen to reduce the overshoot of the system. This was simulated in MATLAB and the GuRoo simulation with a step. The simulation results are shown below in Figure 23.

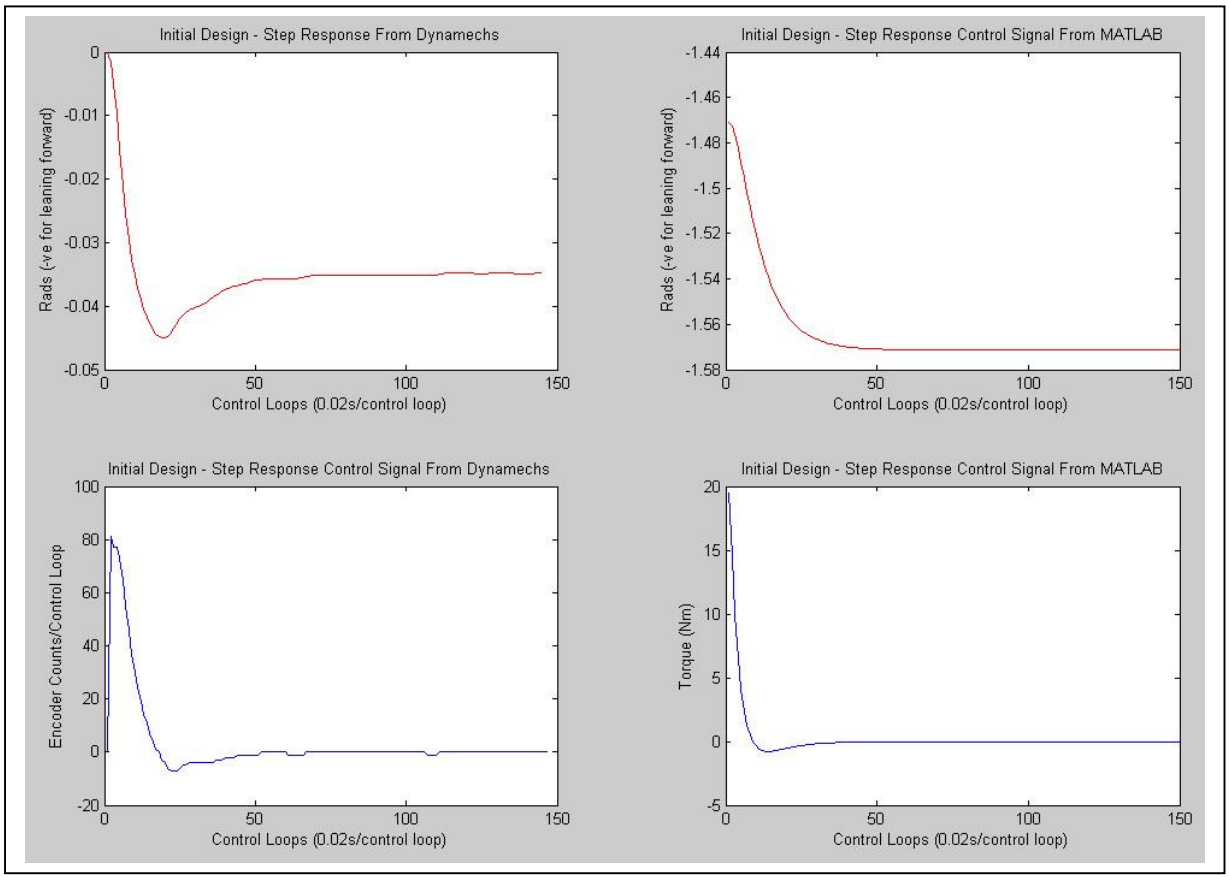


Figure 23: MATLAB Simulation vs. GuRoo (Dynamechs) Simulation. Left Side – Dynamechs Simulation, Right Side – Simulink (MATLAB) Simulation, Top Side – Step Responses, Bottom Side – Control Signals.

The top-left graph shows the step response in the GuRoo/Dynamechs simulator with the corresponding control signal in the bottom-left graph. This was compared to MATLAB simulation step response and control signal in the top-right and bottom-right graphs respectively. Although different step sizes were inputted into the system, the response should be exactly the same with the only difference being that in amplitude.

Comparing the Dynamechs simulation to MATLAB Simulink simulation, it can be seen that more overshoot occurs in the Dynamechs simulation. This is mainly due to the additional knee plus hip joints and foot contact forces of the humanoid robot whereas the MATLAB simulation only models the torso joint and ankles of the robot. Apart from the overshoot of 28%, it can be seen that the response times are quite close with only a 0.5 second difference in time to peak between the two simulations, In comparing time to settle, it can be seen that hardly any difference can be examined from the graph. Overall, the designed control system time response was within specifications but the overshoot on the high side. This high overshoot is dealt with in the next section.

8.1.2 MORE REFINEMENTS

Using these gains of P = 210, D = 20 and I = 10, although the system had a large overshoot, the Dynamechs simulation was smooth and able to stabilise for a step response. The large overshoot and fast time response effects came into play when tasks or forces were implemented on the humanoid robot in simulation, the torque produced by the torso to control the desired overall angle made the robot feet loose contact with the ground surface and thus completely nullifying the balance model. Thus it was determined by a series of trials that a new response time of 1.5 seconds and overshoot of less than 20% was required and would keep the torque produced on the ankles by the torso movement to a minimum. These final gains were found to be that shown in Table 7 and are simulated in subsequent section 8.2.

Gain Type	Value
Proportional (P)	42
Integral (I)	0.7
Derivative (D)	7

Table 8: Table of Final PID Controller Gains.

8.2 FINAL SIMULATION RESULTS

8.2.1 STEP RESPONSE RESULTS

Using the newly and final chosen gains of the balance control system, the system was implemented and the step response obtained to verify the desired response and overshoot. Figure 24 shows the Dynamechs simulator step response for the overall desired angle using control of the torso pitch joint.

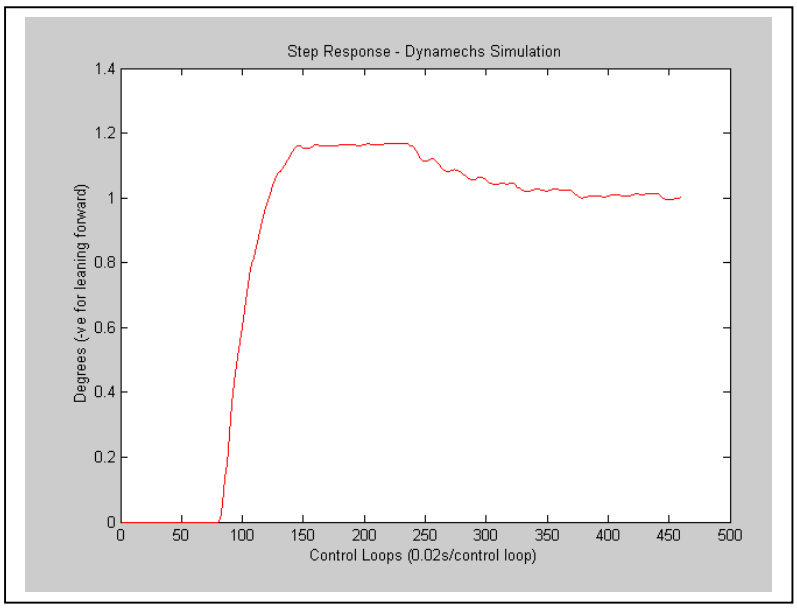


Figure 24: Dynamechs Step Response

Examining Figure 24, the time to peak is 70 control loops = 1.5 seconds and the percentage overshoot is 15%. Comparing these to the original gains, the system has increased the time to peak as desired and almost halved the percentage overshoot meeting the new specifications. Using this response, attitude behaviours were then simulated on top of two main movements, crouch and GA-Walk. The behaviours were then simulated with input forces and results obtained.

8.2.2 COG CONTROL RESULTS

The COG control was implemented in Dynamechs simulation and examined to be working as designed keeping the COG of the system to a fixed pitch position specified in the balance control system. This was set at 0.008m from the base of the ankles in the direction of the toes of the feet and a roll position of 0.0m in between of the robots feet. Although this was found to be working correctly, once any movement, forces or tasks were implemented on the robot, due to the changing desired overall angle to keep the COG at a fixed position, the upper torso required very large movements. This caused the

robots torso to reflect too much torque on the ankles and thus causing them to buckle and fall over. In addition to the robot falling over due to the large torques produced, when the robot was upright and stable for slight movements such as crouch, the robot ends up in positions that is not possible or undesirable by humans. Thus the simple COG implementation of keeping the COG fixed was found to be unacceptable.

8.2.3 POSTURE CONTROL RESULTS

The simple posture control attitude behaviour was found to produce some good results. Again, using the finalised gains, the posture control behaviour was implemented in the Dynamechs simulator with a desired posture or overall angle of the humanoid robot set to –0.035 rads to help stabilise the robot whilst performing tasks or inputting forces. The posture control was first tested on top of the movement crouch then GA Walk with the results of these tests shown in Figure 25 below.

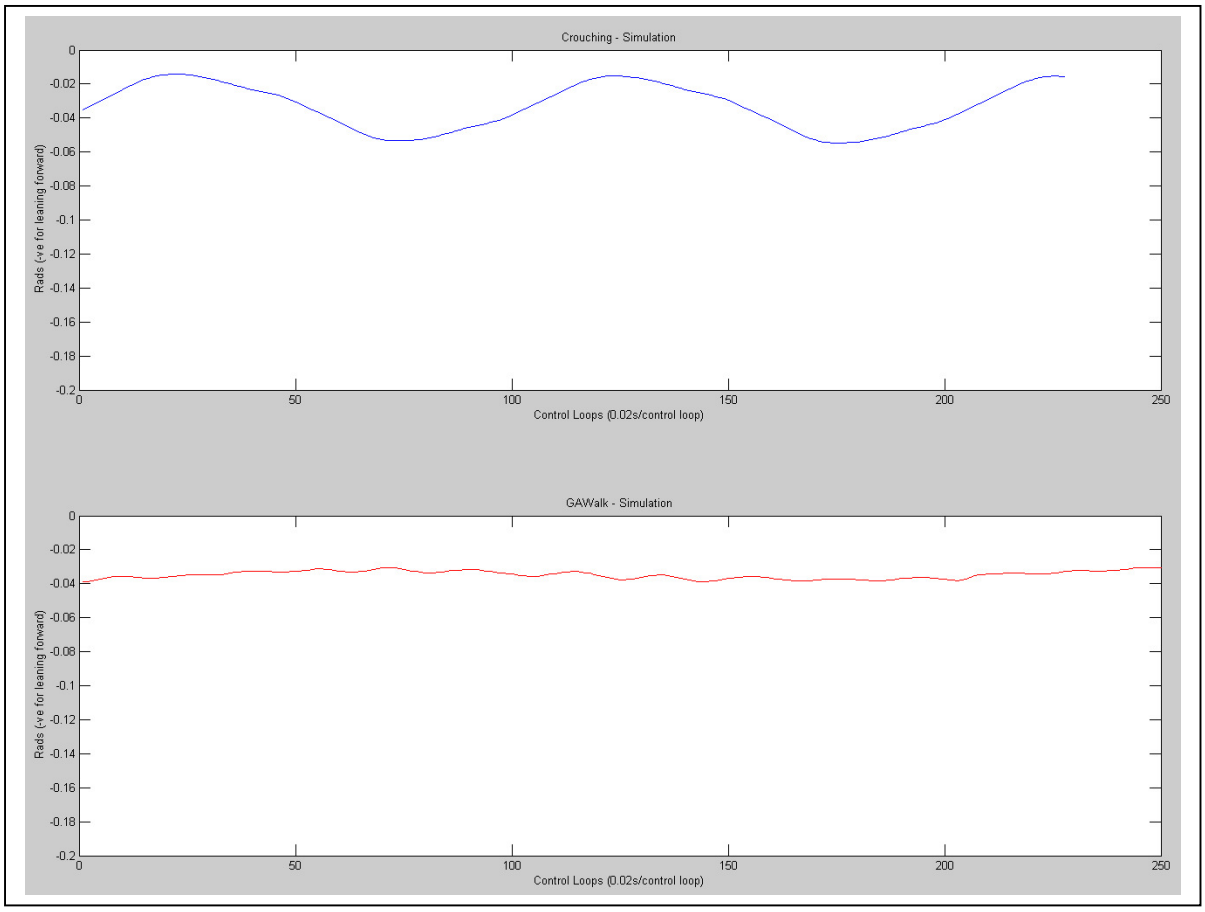


Figure 25: Dynamechs Simulation, Top – Crouch, Bottom – GA Walk.

From Figure 25, it can be seen that the desired overall angle tries to correct itself as the robot crouches. The graph above shows the robot crouching two (2) times with the robot angle leaning forward at the bottom cycle of each crouch. As the robot crouches quite fast, the response time of the balance control system cannot cope and as a consequence the desired position could not be held exact. Although the system still helps the robot balance by keeping the robots posture correct whilst crouching.

Better results were obtained on the GAWalk in the lower half of Figure 25. Here, it can be seen that the balance system works well with the smooth GAWalk simulation keeping the desired angle close to -0.035 rads. This kept the posture of the robot upright and thus puts the robot in a better position to counter external forces.

The next test for the posture control behaviour was that of input forces onto the robot. These forces were generated on the torso of the robot to simulate a push in the chest. Two forces were inputted into the robot, a small force of 60N applied for 20ms and a medium force of 90N applied for 20ms as well. Graphs of the force input test with and without the balance system is shown in Figure 26 below.

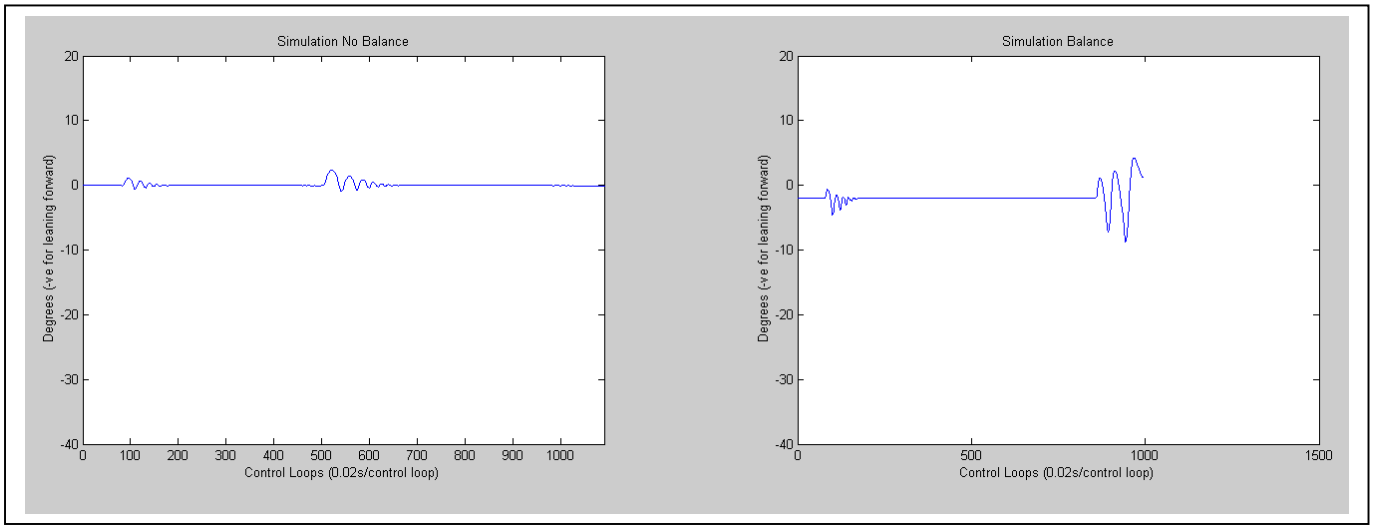


Figure 26: Dynamechs Simulation With Input Forces.

Some unexpected results were found with the posture control system and input forces as seen in the graphs shown in Figure 26. According to the graphs, the robot was much better off without the posture control balance system. Examining the two graphs, the robot with no balance was able to stabilise for both the small and medium forces whereas with the balance system, for the small push, more oscillations occurred and with a medium push, the robot became unstable. This was certainly not

desirable and was due to the robot loosing footing from the ground because of the input forces. This would cause the robot to be pushed back with the balance system moving the torso forward to try and correct the error. The robot will then tip forward and due to the control system correction, the robot's torso will need to be tipped back. This tipping back moment added with the ground reaction forces on the toes of the robot produces feedback forces within the system and thus cause the system to oscillate more than usual. For the medium force, this feedback is big enough that unstable oscillations occurred. Thus it was examined that to remove this ground reaction force feedback loop in the system and to make posture control stable, some form of ankle control was required.

8.3 POSTURE WITH ANKLE CONTROL

It was determined in the previous section that some form of ankle control was required in order to realise the posture balance behaviour. The original goal of the thesis project was to only control the torso joints of the robot to form one half of the balance system with the other half being the ankle compliance system. This ankle compliance system shown in Figure 6 would control the ankle behaviours and the desired Z.M.P of the robot, but as the ankle compliance system had not been completed to a stage to be tested with the active balance system, a simple ankle behaviour that will substitute for the ankle compliance system for this thesis was created.

8.3.1 SIMPLE ANKLES BEHAVIOURS

To implement the substitute ankle compliance system, the same control system designed for the torso joint with gains changed by a multiple of 0.3 was used to control the robots ankles. This produces an ankle behaviour that simply pushes the robot's toes on the ground surface when the robots torso is being moved forward and vice versa. This allows the robot to counter the force of the torso moving forward with the ground reaction force on the toes of the robot created by moving the ankles in an opposite motion to that of the torso. This is illustrated in Figure 27 below.

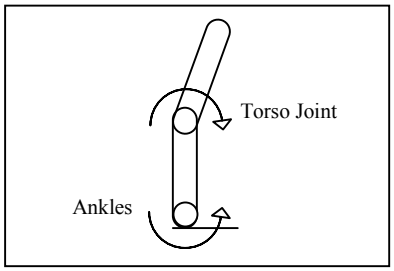


Figure 27: Simple Ankle Behaviour

8.3.2 POSTURE WITH ANKLE CONTROL RESULTS

Posture control of the torso combined with the simple ankle behaviour produced some very good results. All tests were conducted again using the newly developed balance system with ankle control. This time the robot was able to stabilise for both small and medium forces as well as recovering its original foot positions flat on the ground much faster than without the balance system. This allows the robot to recover its footing and thus is able to stabilise the torso posture control. A graph showing a simulation of the exact same medium force applied previously is shown below in Figure 28.

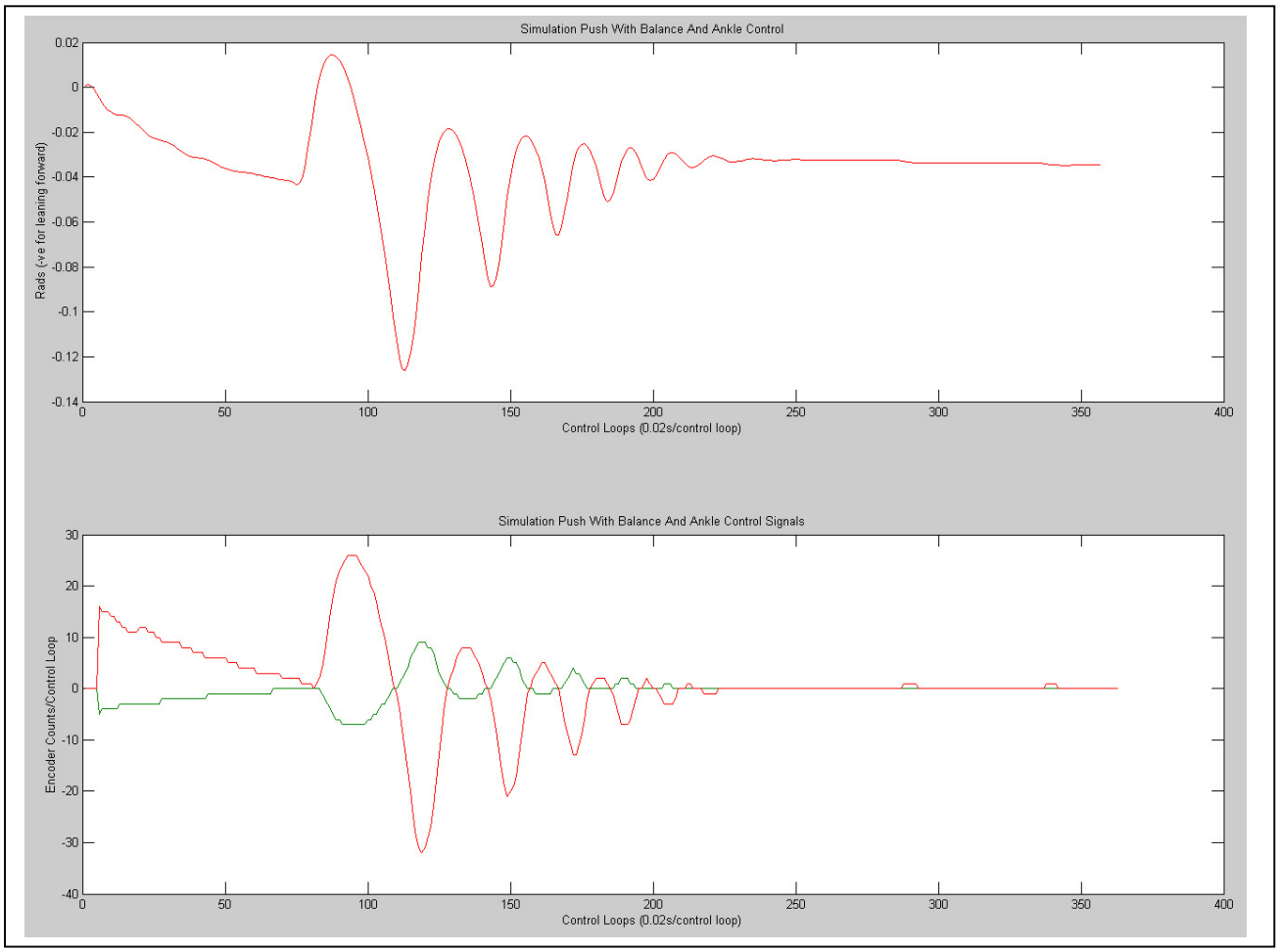


Figure 28: Posture and Ankle Control Simulation with Medium Force Applied.
Top – Overall Robot Angle, Bottom – Control Singles, Red = Torso, Green = Ankles

As seen in the top of Figure 28, the robot is able to stay upright and balance with the same medium force applied that caused instability in the posture without ankle control system. The initial movement of the graph from 0 rads to –0.04 rads is the step to the initial desired position set to -0.035 rads. A medium force can be seen being applied at control loop 70 (1.4 seconds). The control signals are shown in the bottom of Figure 28 with the ankles shown in green and torso velocity control signal shown in

red. As seen from the graph, the ankle control signals contribute quite a bit to the system from –5 to +10 encoder counts per control loop.

The ankle control system was also implemented on the crouch and GA Walk movements and found successful with very little change from the system without ankle control. Thus by introducing the ankle control behaviour system, the robot was now able to stabilise for small and medium forces plus keep the posture of the robot close to the desired posture whilst performing tasks in simulation. After the numerous changes applied to achieve the desired attitude and control system, the active balance system was then implemented on hardware and results obtained from the real hardware. This is observed in the succeeding chapter.

CHAPTER 9

HARDWARE IMPLEMENTATION

To implement the active balance system on the real robot, the IMU sensor was the only hardware required to be setup and implemented. As the GuRoo robot project has been running for three (3) years, all software and hardware was already implemented for the control of the torso and ankle joints. Thus in this chapter will describe the IMU implementation and examine/compare the hardware results.

9.1 IMU SPECIFICATIONS

The inertial measurement unit (IMU) consist of three main sensor components:

1. One (1) 3-Axis Gyroscope.
2. Three (3) Magnometers.
3. Four (4) Accelerometers.

These sensors can be used to produce inclination, rotational and translational velocities plus rotational and translational accelerations. The sensor is programmed via a serial RS232 connection and can be set up to deliver information in numerous ways. Since only inclinations are required by the active balance control system, only pitch, roll and heading information were extracted. The information is given in a 16-bit format that requires scaling to produce the correct inclination. Details of the IMU sensor, the initialisation procedure and the format of extracted information can be found in the electronically attached IMU manual [17] found in Appendix E.

9.2 IMU SOFTWARE AND HARDWARE DESIGN

The software design and implementation was conducted by Jonathan Roberts [17] and Damien Kee [2] and is included here for completeness. The IMU sensor was implemented on the GuRoo's CAN bus with the IMU initialised to output pitch, roll and heading data in calibrated format. This CAN message is picked up by the GuRoo CAN bus, decoded, scaled accordingly and put input shared memory that can be accessed by any function. A block diagram showing the IMU design implementation is shown below in Figure 29.

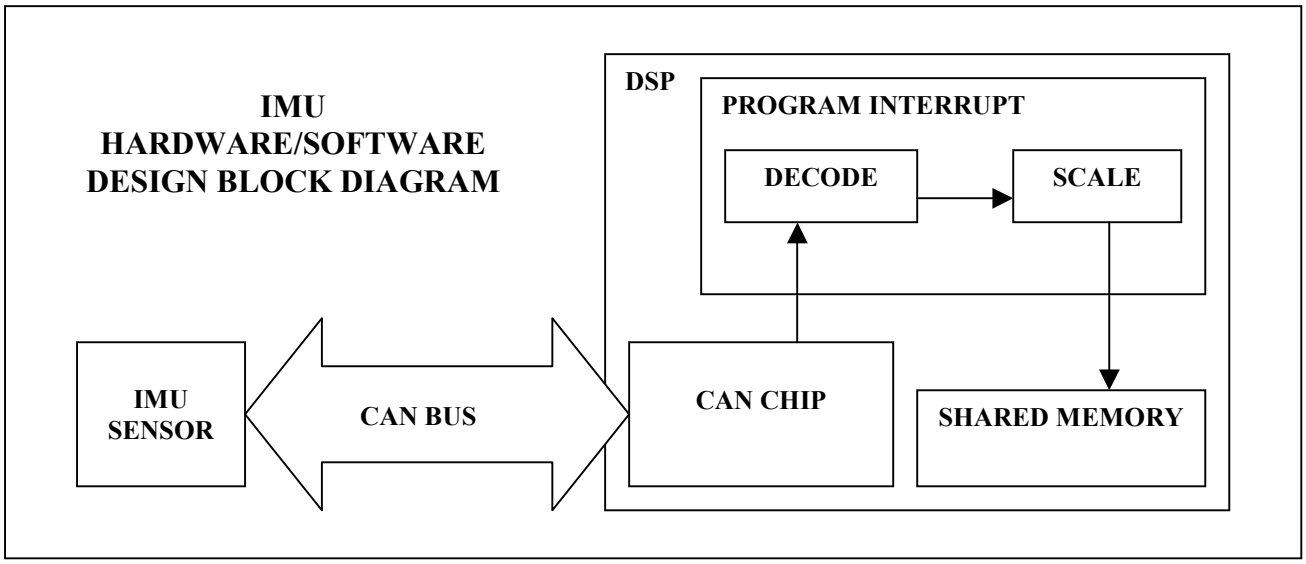


Figure 29: IMU Design Block Diagram

9.3 REAL HARDWARE RESULTS

As the GuRoo simulator is specially setup to easily swap between simulation and the real robot, all of the previous software designed and conducted in the GuRoo simulator could be easily run on the real robot hardware with results being obtained immediately. An *#define SIMULATION* line was the only line required to swap between simulation and the real robot software system. Tests on the real robot were conducted in the same order as that conducted in simulation with another test of sustained pushing added to cover another disturbance area. The complete hardware tests are categorised below:

1. Step response without the use of the robots ankles was obtained and verified with simulation.
2. Two (2) test tasks, crouch and GA Walk were implemented without the new ankle control system
3. Small and medium forces applied on the real robot and the responses of the balance system obtained without the use of the simple ankle control.
4. Step response with the complete ankle and posture balance system.
5. Two (2) tests task, crouch and GA Walk with the complete ankle and posture balance system
6. Small and medium forces applied on the real robot with the complete ankle and posture balance system.
7. Small/Medium sustained pushing was applied on the real robot with the complete ankle and posture control system.

Unfortunately, due to the roll motors in the torso of the GuRoo robot failing and not being able to be fixed due to obsolete motor encoders, only balance effects in the pitch motion of the robot were investigated and implemented. Although the roll of the torso does play an important part in balancing the roll motion of the robot, it is not as critical as the pitch balancing because of the wider support polygon whilst standing and because of the less number of roll joints in the humanoid robot.

9.3.1 INITIAL RESULTS AND REFINEMENTS

The step response of the hardware system was the first test to be conducted but was found to performing incorrectly and jittered around its desired position. The motor was unable to stabilise properly to the desired point. Log files of the balance system were examined and it was found that the jitter was due to the noise produced by the IMU sensor. This was magnified by the gravity compensation signal and thus caused the robot to jitter accordingly. Thus as the robot operates around the upright equilibrium point, this gravity compensation signal provides minimal compensation and more harm than good, thus it was removed. This in turn removed the jittering of the motor and therefore the control system was reduced to only a PID compensated controller.

9.3.1 REAL STEP RESPONSE RESULTS

Graphs showing the real vs. simulation step response are shown at the top of Figure 30. Of several runs conducted on the robot, two of the runs were picked and shown in Figure 30; left graphs are run 1 whereas right graphs are run 2. The simulation time response signal is shown in red where as the real hardware implementation is shown in blue. The control signals for the real hardware implementation runs are shown in the lower graphs of Figure 30 in magenta (purple/pink).

Examining the simulation vs. real hardware step response graphs, it is easily seen that the robot behaves as expected from simulation with same time response and overshoot as designed and produced in simulation. Again from the control signals shown in the lower graphs, it can be seen that the control system is contributing to the correction of the overall desired posture. Differences is the starting positions of the simulation vs. real responses graph is due to an offset added to the response for ease of comparison reasons only.

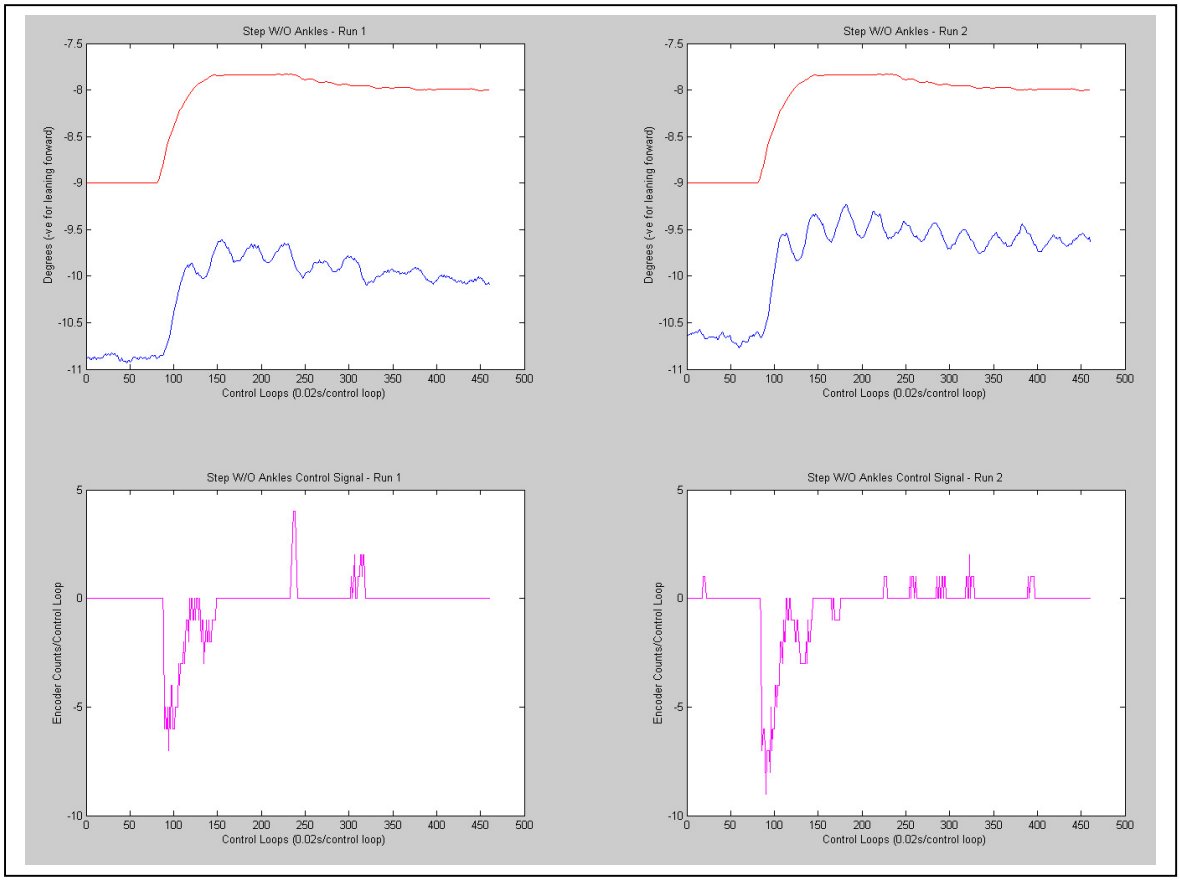


Figure 30: Posture Step Responses Without Ankle Control – Real and Simulation.

Top Left – Run 1 Step Response, Top Right – Run 2 Step Response,
Bottom Left – Run 1 Torso Control Signal, Bottom Right – Run 2 Torso Control Signal

9.3.2 REAL POSTURE CONTROL RESULTS

The posture control system with no ankle control was implemented on top of two (2) tasks; crouch and GA walk as conducted in simulation. Although the simulation of the GAWalk and balance system produced some promising results, the opposite was found once implemented on the real robot GAWalk. This was due to the real robot GAWalk being very different from the simulation GAWalk. The GAWalk on the real robot changes its torso inclination more rapidly with very large angle movements and thus the designed balance system is unable to response fast enough to these changes. In turn the balance system causes the robot to sway and cause much larger angle movements than without the balance system. This is shown in the graph in Figure 31.

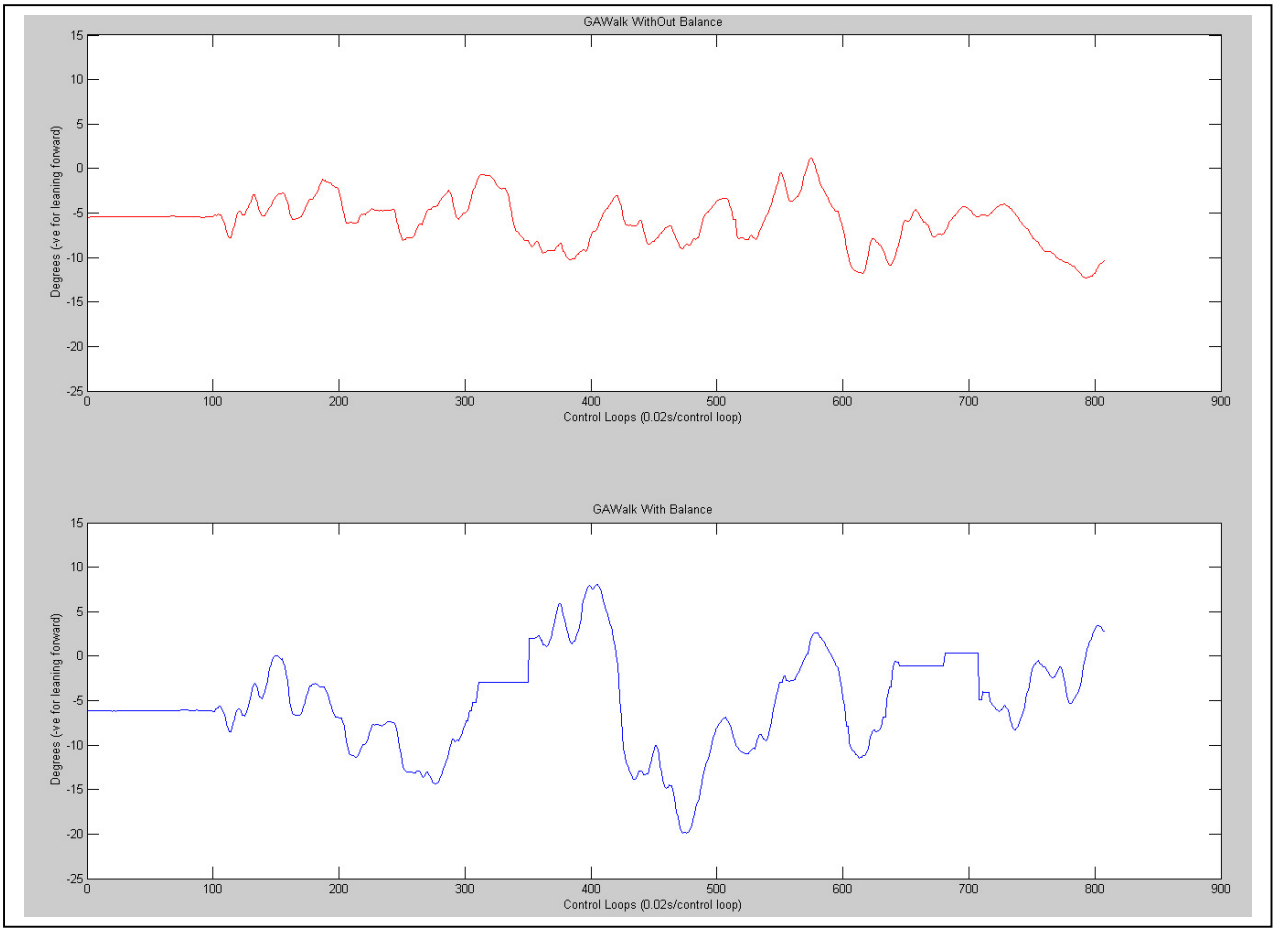


Figure 31: Real Hardware GAWalk Responses Without Ankle Control.
Top – GAWalk without Balance, Bottom– GAWalk with Balance

Due to the very large torso angle changes in the real GAWalk, these changes were much too large and jerky to counter. Responses could be improve by increasing control gain but it was examined that by simply increasing time response the robots torso will again generate too much torque and cause the robot feet to loose contact with the ground or cause the ankles to buckle.

Although GAWalk failed to operate under the posture balance system, the other task crouch benefited from the balance system. Usually the robot is unable to crouch properly without human help unless it was set up and tuned perfectly but by using the active balance system to dynamically alter the posture of the robot whilst crouching, the robot was able to crouch unattended without the need to be perfectly setup. Graphs of the robots overall inclination whist crouching are shown in Figure 32 below.

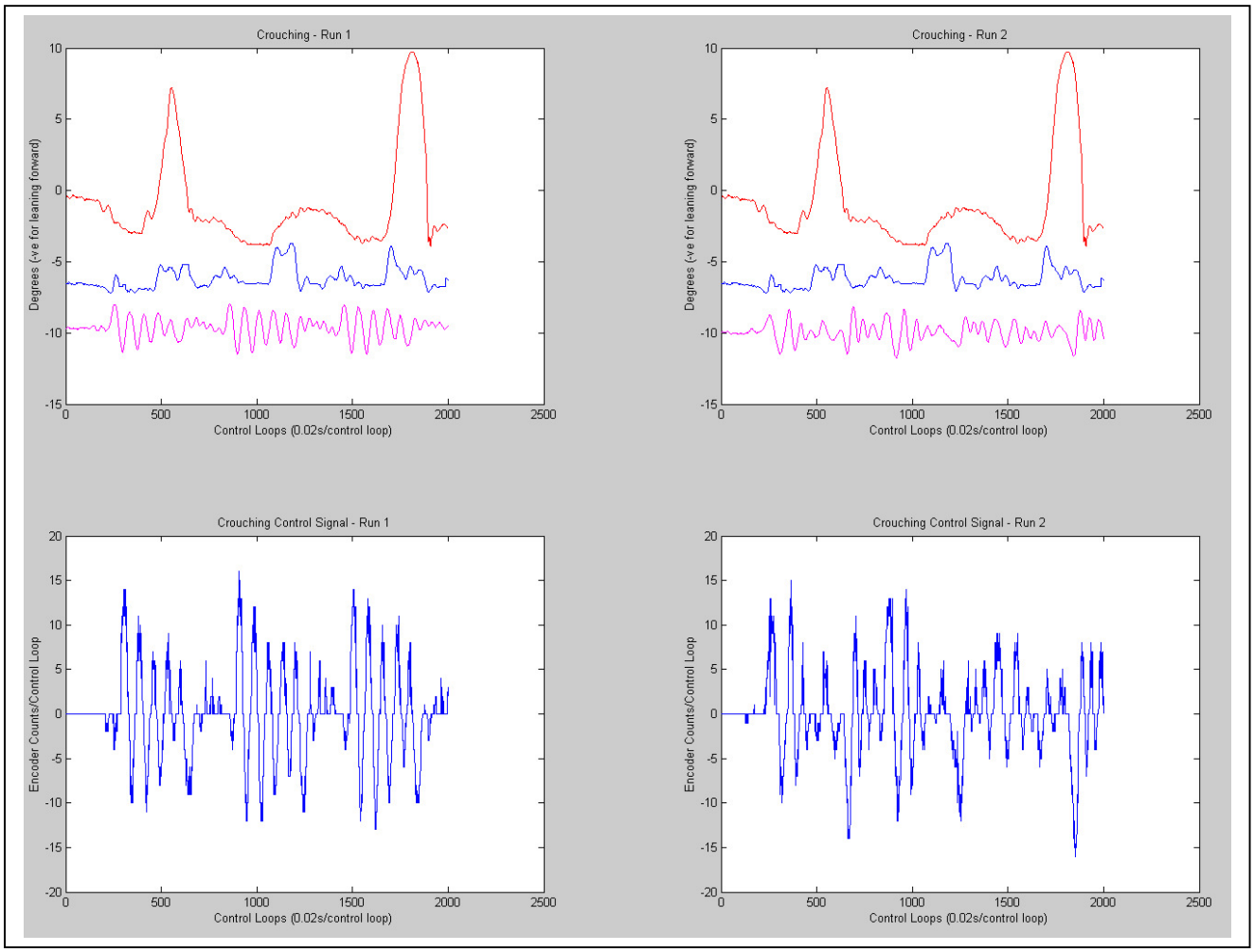


Figure 32: Real Hardware Crouching Responses Without Ankle Control.

Top: Run 1 and Run 2 - Red Line, Blue Line = Crouch without Balance, Magenta Line = Crouch with Balance
Bottom: Run 1 and Run 2 - Blue Line = Control Signal For Balance System

The upper graphs in Figure 32 shows the crouching inclination responses of two (2) runs conducted on the real robot. The red graph is the robot crouching without the balance system with the large spikes showing the robot falling over and being caught by outside assistant. This graph is used to emphasise the regions of the crouching motion that the robot is most unstable. The blue line in the top graphs of Figure 32 shows the robot crouching once more without the balance system but will less help from outside assistant. Again this graph is used to emphasise the regions of the crouching motion that the robot is most unstable. Finally the magenta line in the top graphs and blue line of the lower graphs of Figure 32 show the robot crouching with the balance system and the control signals produced by the balance system respectively. It can be seen from these graphs that the balance system is able to reduce the deviation from the desired posture and able to balance the crouching motion when it is most needed

during the up cycle of the crouching motion. Therefore the balance system was found successful for the crouching task keeping the robot upright and balanced with absolutely no human help required.

The final test conducted with the posture balance system with control of the ankles was that of the input forces applied to the robot. The graphs of these tests are shown below in Figure 33.

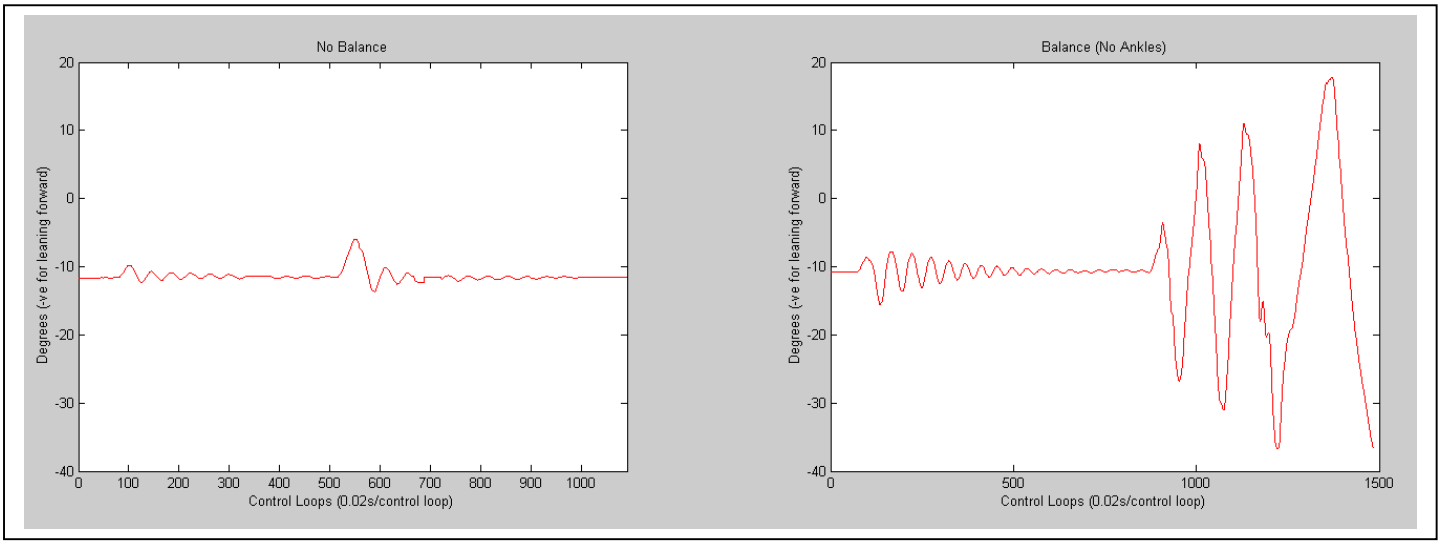


Figure 33: Real Robot Results With Input Forces.
Left – No Balance System, Right – With Balance System

From Figure 33, it can be seen that the behaviour of the real robot is exactly the same to that of the simulated robot pushes shown in Figure 26. The left graph shows the robot without the balance system and a small and medium push applied to the robot successively. The right graph shows the robot with the active balance system with the same small and medium pushes applied. Thus this verified the need for some form of ankle control system to allow the posture control system to stabilise and counter disturbances such as pushes.

9.3.3 REAL POSTURE WITH ANKLE CONTROL RESULTS

As the first hardware tests were performing reasonably well, the posture control with the simple ankle control system was implemented on the real robot. Only impact and sustained tests were conducted using the ankle control system due to reasons of safety that may be violated by mixing various control signals. Results of the impact tests are shown in Figure 34 below.

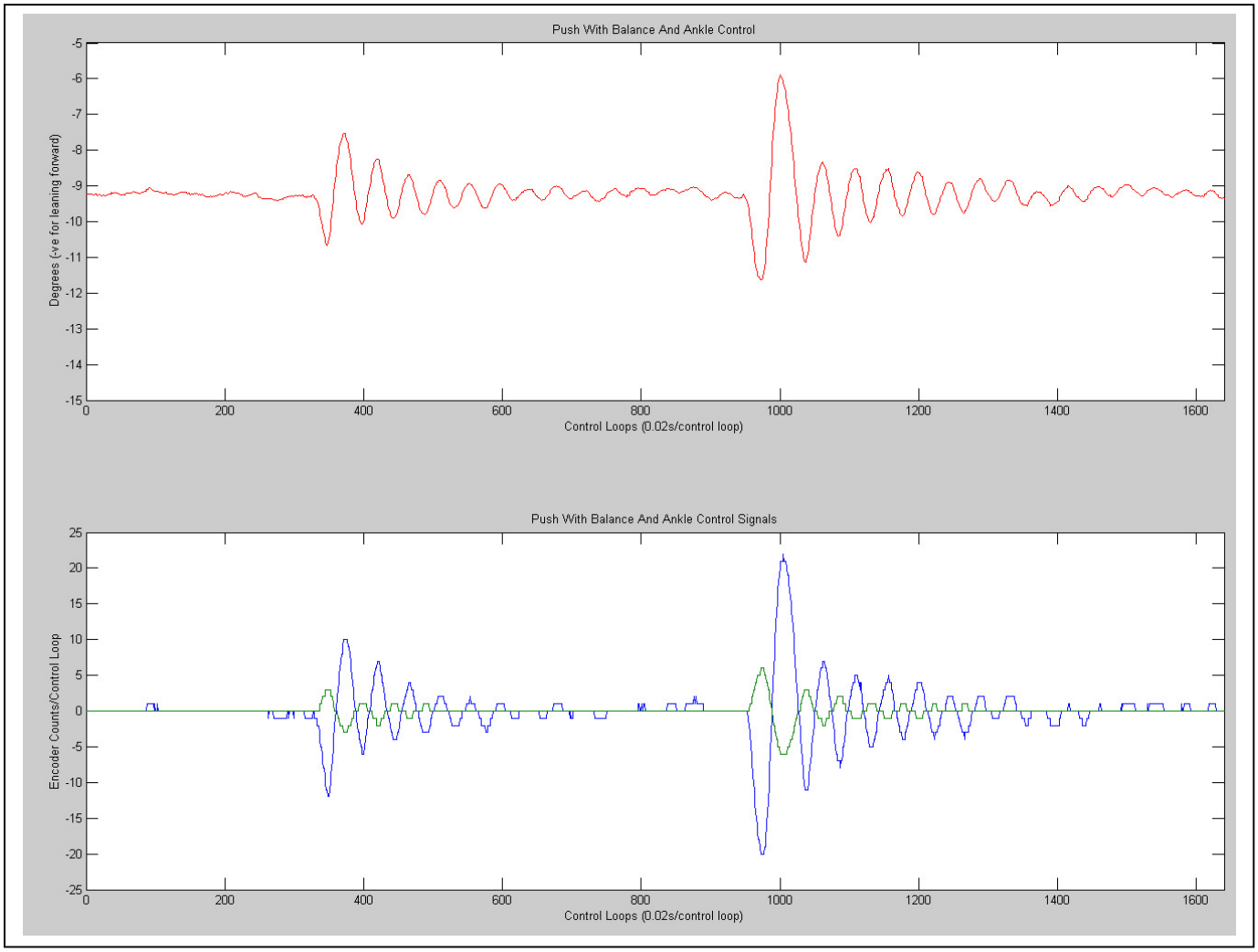


Figure 34: Real Robot Results with Ankle Control and Input Forces.

Top – Posture and Ankle Balance System with Two (2) Applied Forces.
Bottom – Posture and Ankle Control System Signals, Green = Ankles, Blue = Torso.

Examining the graphs shown in Figure 34, the top graph shows the response of the posture and ankle control system with two forces applied on the robots torso. The lower graph shows the corresponding control signals from the balance system. Thus it can be seen the ankle and posture balance system behaves as expected from the simulation in Figure 28 and is able to stabilise the robot for small and medium forces with some minor oscillations occurring due to the numerous pitch joints located throughout the legs of the robot.

One final test was conducted on the robot to cover a range of the types of forces applied to the robot. Originally, impact pushes were applied to the robot and the robot responded as expected. A final test of sustained pushing on the robot was developed and tested on the real robot. The results of the sustained pushing test are shown in Figure 35 below.

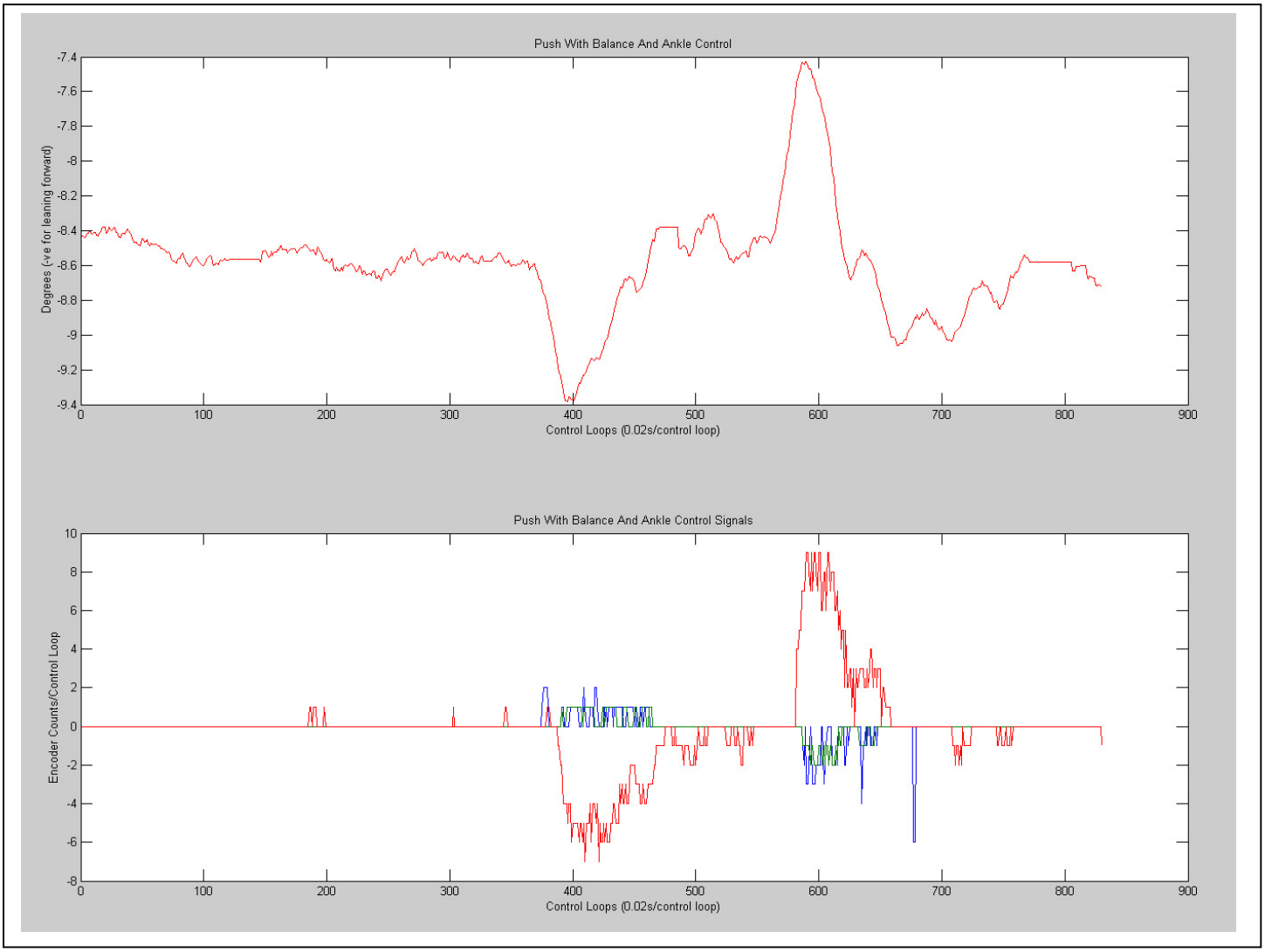


Figure 35: Real Robot Results with Ankle Control and Input Forces.

Top – Posture and Ankle Balance System with Two (2) Applied Forces.
Bottom – Posture and Ankle Control System Signals, Green = Ankles, Blue = Torso.

The upper graph of Figure 35 shows the robots overall inclination with a sustained push being initiated at control loop number of 380. This push is held until control loop 560 where the force is taken away and no longer applied. The lower graph of Figure 35 shows the corresponding control signals movements with the red line showing the torso movement with the green and blue lines showing the ankle control movements

From the upper graph of Figure 35, it can be examined that the robot is able to correct its posture to the desired position despite the sustained push. This in turn pushes back on the force applied to the robot and thus the robot is able to stay upright. Once the sustained push is taken away from the robot, the balance system is once again able to correct its posture and remain in an upright position. This would

not occur without the balance system, as the robot would not resist the sustained force being applied and thus eventually fall over. This test showed some very good results showing the that active balance system is not only able to help in impact forces but with other disturbances as well such as sustained pushing. When pushing the robot, the robot is able to react to the force applied and counter or push back on the force applied. The makes the robot feel very realistic and "alive" as it is able to react to outside disturbances.

9.4 OVERALL COMPARISON WITH SIMULATIONS

Comparing the overall results obtained from real hardware versus the results from the GuRoo (Dynamechs) simulator. Many of the results were very consistent and as expected. Before each of the tests was carried out, it was examined whether or not the balance system would be able to cope with the tasks or forces. Thus because of the large torso movements generated by the GAWalk on the real robot, it was no surprise that the balance system did not help the task. Apart from the GAWalk being incorrect, all other simulations and real results matched up correctly. This verified that the implementation of the IMU sensor and forces in simulator closely matched that of the real world and the balance system was working correctly in both the simulated and real world.

CHAPTER 10

DISCUSSION

10.1 ACTIVE BALANCE SYSTEM EVALUATION

To evaluate the active balance system, the overall scope and goal of this thesis topic will be redefined here for convenience. The overall goal of this thesis is to create and implement a system that will help allow the humanoid robot "GuRoo" to remain upright and balanced despite adverse disturbances. These adverse disturbances were broken up into three (3) categories, small forces, medium forces and large forces. Large forces were not to be handled in this active balance system because of the complexity and time involved.

The initial goal was limited to the control of the torso joints only but due to the ankle compliance system not being complete, a very simple ankle control system was developed so that outcomes of this thesis project could be produced.

At the beginning of this thesis project, the robot did not have any information regarding the outside world other than that of the motor encoders. This meant that the robot does not know whether it is standing up or flat down on the floor. The robot was unable to stay upright and required assistance in crouching and walking tasks. The robot was able to stay stable for small and medium impact pushes but would oscillate and not counter the force applied. For sustained pushes the robot would fall over.

With the balance system implemented at the end of the thesis topic, the robot is now able to crouch unattended, resist and counter small and medium impact forces as well as stay upright and balanced for sustained pushes shown a the snapshot capture in Figure 36. Although the robot is now able to balance for these forces and tasks, the balance system for the main task of the robot walking was not successful. This was due to the torso moving much to far and much to fast for the balance system to help. An examination of why the real GAWalk vs. the simulation GAWalk was conducted and reasons of difference believe to be the contact foot models in the simulation being incorrect. This causes the robot

to land smoothly and not caused the torso to jerk around whilst walking. Although the balance system cannot operate on top of the GAWalk, the newly developed gait by Tim Pike shows a much smoother walk in which the balance system is able to respond to.

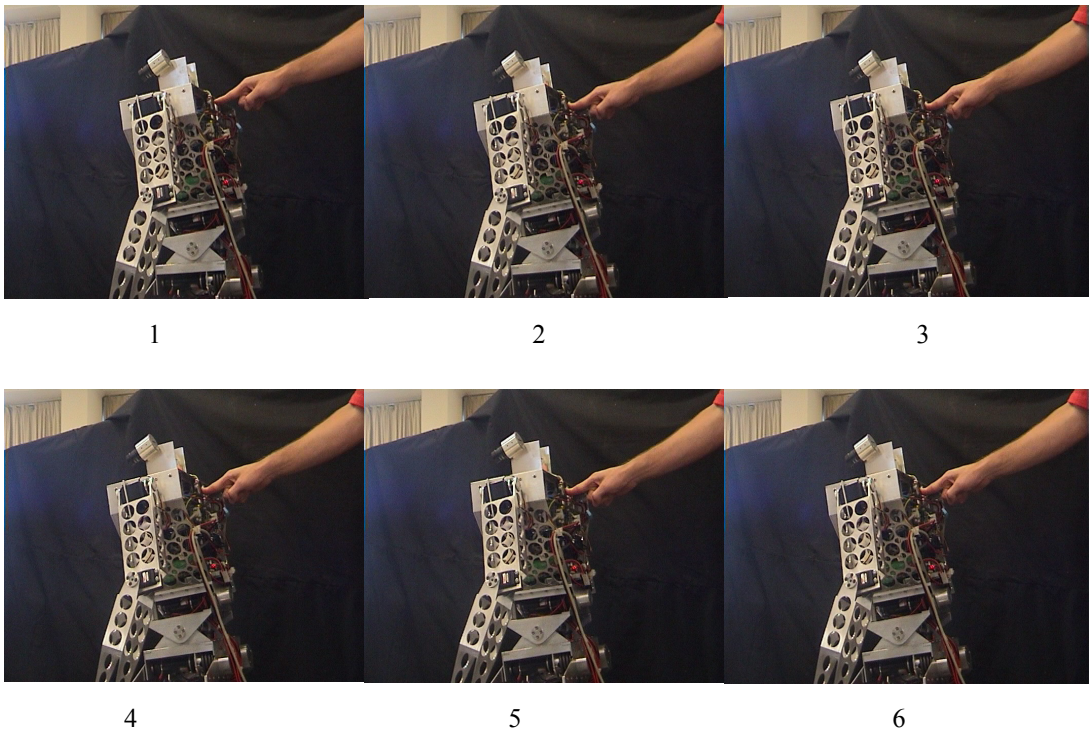


Figure 36: Snapshot Capture of Robot Resisting or Countering Sustained Push From Picture 1 to 6.

To sum up the evaluation, the active balance system designed and implemented in this thesis was able to counter small to medium disturbances seen in tasks, joint errors and/or forces applied. Due to the response of the designed control system, large and fast disturbances, which are not encompassed in the scope of this thesis, caused problems and the balance system was not able to perform the corrections to the robot. As simple ankle control was substituted for the ankle compliance control, the active balance system shows great hope in countering larger forces when the much better ankle compliance system is implemented completing the overall balance system for the robot. A six-frame snapshot of the sustained pushing is given in Figure 36. Paying close attention to the torso of the robot and the human hand, it can be seen that the posture of the robot remains the same and force applied on the robot is being countered. Complete videos of the robot performing the crouching motion with the balance system and the robot resisting impact and sustained pushes are attached electronically in Appendix E.

10.2 FUTURE DEVELOPMENTS

Many future developments can be made to the active balance system and the GuRoo project to help balance the robot. An immediate development that would show instant results would be to interface the tasks or gait phases to the active balance system. This would allow the walk to generate desired postures for the various phases encompassed in their tasks. As the balance system was only tested using a fixed posture, this is not at all correct for many tasks being performing by humans such as walking and kicking. By providing desired posture positions according to the tasks phases, this would help stabilise the robot tremendously.

Foot sensors for the robot would be the next development to complete and in turn being able to complete the ankle compliance system to control the Z.M.P of the robot. The active balance control system already shows very good results with a simple ankle control system and thus shows great promise for when the ankle compliance system is integrated into the robot.

Another future development to the active balance system produced in this thesis would be to develop a much more advanced balance model that models the robots feet, knees and hip joints. An advanced MISO control system design can then be developed using additional IMU sensor information such as velocities and accelerations to provide much better and more customised responses. The control system would then be able to reduce or increase the torques generated by the torso when required and in turn be able to control the Z.M.P of the robot to a certain extent.

These are just three of the many numerous improvements and developments that can be made in the future. This thesis merely provides the basis or platform in which many more generations of balance system can build upon.

10.3 CONCLUSION

In conclusion, the active balance system created for the thesis topic “Active Balance System of a Humanoid Robot” was successful in keeping the humanoid robot upright and stable despite small and medium adverse disturbances. The balance system stabilised the crouching motion task and was able to stay upright against applied impact forces and sustained forces.

The course of the thesis project began with the development of a balance model and formulation of the transfer function. Next the design of a control system was conducted and attitude behaviours created. Finally the software and hardware implementation of the active balance system was done. Throughout the design process of the active balance system, fine-tuning of the various system were made from examining preliminary results when the system was simulated or implemented. In the end, a complete balance system was created that had been tuned to counter small to medium disturbances and perform tasks unaided.

This active balance system was first implemented using simple posture control for the balancing of the robot and was found to be successful to an extent with only the use of the torso joints. To counter much greater adverse disturbances, an ankle control system was introduced later to the balance system as a crude substitute for the ankle compliance system. With this additional system, the robot was able to successfully keep its posture and balance for small and medium disturbances encountered from tasks and forces applied.

The thesis was completed successfully, met specifications, achieved its the goal and scope set out at the start of the project. This thesis will form a solid base and platform for more advanced balance systems to build on and will hopefully contribute to the overall RoboCup goal of winning against the human worlds top soccer team.

BIBLIOGRAPHY

1. Schilling, R.J., *Fundamentals of Robotics - Analysis and Control*, in *Fundamentals of Robotics - Analysis and Control*. 1990, Prentice Hall.
2. Kee, D., *Gait Generation and Implementation for Humanoid Robotics*, in *School of IT and Engineering*. 2003, University of Queensland: Brisbane. p. 43.
3. Hirai, K.H., M.; Haikawa, Y.; Takenaka, T. *The development of Honda humanoid robot*. in *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*. 1998. Leuven , Belgium.
4. Y.N. Tomomichi Sugihara, H.I. *Realtime Humanoid Motion Generation through ZMP Manipulation based on the Inverted Pendulum Control*. in *IEEE Internation Conference on Robotics & Automation*. 2002. Washington DC.
5. Niku, S.B., *Introduction to Robotics - Analysis, Systems, Applications*. 1 ed, ed. D.A. George. 2001: Prentice Hall. 349.
6. Pagello, E., *RoboCup 2003 - General Information*. 2003.
7. Marshall, I.J., *Active Balance Control for a Humanoid Robot*, in *School of ITEE*. 2002, University of Queensland: Brisbance. p. 89.
8. Kubica, E.W., D.; Winter, D. *Feedforward and deterministic fuzzy control of balance and posture during human gait*. in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*. 2001. Seoul , South Korea.
9. Hing, C.S., *Development of a Dynamic Gait for a Simulated Humanoid Robot*, in *School of Electrical Engineering and Telecommunications*. 2002, The University of New South Wales. p. 81.
10. Vukobratovic, M.a.D.J. *Contribution to the Synthesis of Biped Gait.* in *IEEE Trans. on Bio-Medical Engineering*. 1969.
11. Park, K.D.K.J.H. *Biped Robot Walking Using Gravity-Compensated Inverted Pendulum Mode and Computed Torque Control*. in *International Conference on Robotics and Automation*. 1998. Leuven, Belgium.
12. McMillan, S., *DynaMechs (Dynamics of Mechanisms): A Multibody Dynamic Simulation Library*. 2001.

13. Yamaguchi, J.S., E.; Inoue, S.; Takanishi, A. *Development of a bipedal humanoid robot-control method of whole body cooperative dynamic biped walking*. in *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*. 1999. Detroit, MI , USA.

14. Kajita, S.Y., K.; Saigo, M.; Tanie, K. *Balancing a humanoid robot using backdrive concerned torque control and direct angular momentum feedback*. in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*. 2001. Seoul , South Korea.

15. Pike, T., *Gait Generation For a Humanoid Robot*, in *ITEE*. 2003, University of Queensland: Brisbnace.

16. Wyeth, G., *Module 3, Multiple Joint Design*. 2002, University of Queensland: Brisbane.

17. Roberts, J., *EiMU - User Manual Version 0.47, CMIT Technical Report*. 0.47 ed. 2003: CSIRO.

APPENDIX A

Linearising.m

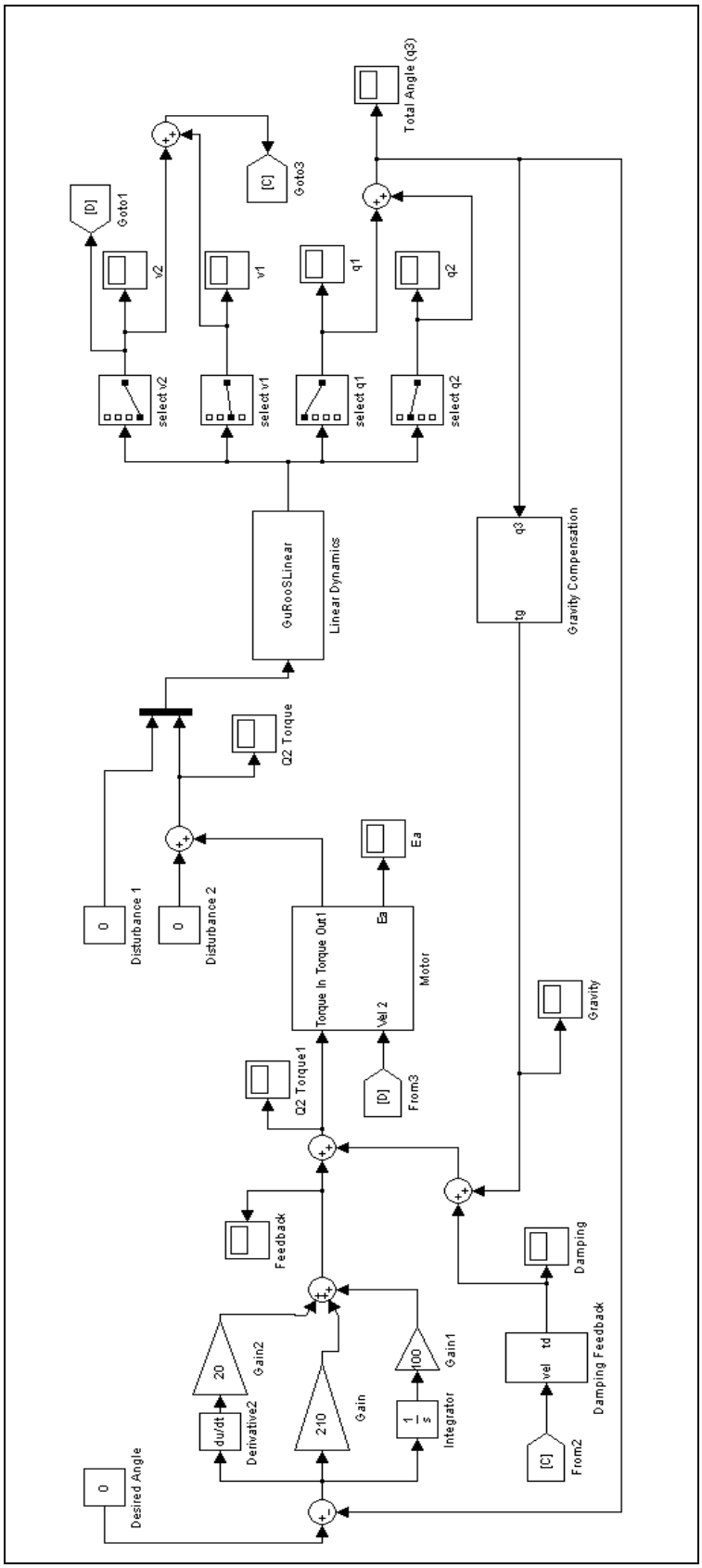
```
g = 9.81;
m1 = 22.967;
m2 = 13.653;
a1 = 0.7;
a2 = 0.5;
b1 = 10;
b2 = 0.1;
k=(g*(m1/2+m2)*a1);

q = ((m2*(a1*a2)^2)*(m1/9 + m2/12));
d11 = ((m1/3 + m2)*a1^2 + m2*a1*a2 + (m2*a2^2)/3);
d12 = ((m2*a1*a2)/2 + (m2*a2^2)/3);
d21 = d12;
d22 = (m2*a2^2)/3;

% x = [q1 q2 v1 v2]T
% xdot = [v1 v2 a1 a2]T

a11 = ((1/q)*(d22*(g*(m1/2+m2)*a1 + g*m2*a2/2-k)-(d12*g*m2*a2)/2));
a22 = ((1/q)*(d22*g*m2*a2/2 - d12*g*m2*a2/2));
a3 = -(d22/q)*b1;
a4 = (d12/q)*b2;

a5 = ((1/q)*(-d21*(g*(m1/2+m2)*a1 + g*m2*a2/2-k)+(d11*g*m2*a2)/2));
a6 = ((1/q)*(-d21*g*m2*a2/2 + d11*g*m2*a2/2));
a7 = (d21/q)*b1;
a8 = -(d11/q)*b2;

A = [0 0 1 0
   0 0 0 1
   a11 a22 a3 a4
   a5 a6 a7 a8];

% u = [t1 t2]T

B = [0 0
   0 0
   d22/q -d12/q
   -d21/q d11/q];

C = [1 0 0 0
   0 1 0 0
   0 0 0 0
   0 0 0 0];

D = [0 0
   0 0
   0 0
   0 0];

states = {'Angle 1' 'Angle 2' 'Vel 1' 'Vel 2'};
inputs = {'Torque 1' 'Torque 2'};
outputs = {'Angle 1' 'Angle 2' 'Vel 1' 'Vel 2'};
```

```
sys_mimo = ss(A,B,C,D,'statename',states,...
                'inputname',inputs,...
                'outputname',outputs);

g = tf(sys_mimo)
```

FindRLocus.m

```
%Constant Values
g = 9.81;
m1 = 22.967;
m2 = 13.653;
a1 = 0.7;
a2 = 0.5;
b1 = 10;
b2 = 0.1;
k=(g*(m1/2+m2)*a1);

%Transfer Function of GuRoo Linearised
numg=[-0.5716 1.777e-15 5.426]
numh=[2.651 1.621 -5.426]
den = [1 2.109 -56.49 54.8 6.475e-014]

f=tf(numg,den)
h=tf(numh,den)

%Total TF from both contributions
z = f+h

%Damping Feedback
numdamp = [b2 0]
dendamp = [1]
damp = tf(numdamp,dendamp)

%Gravity Feedback
numgrav = [g*m2*a2/2]
dengrav= [1]
grav = tf(numgrav,dengrav)

%Both Gravity and Damping Feedback
back = damp+grav

%Using only Gravity Feedback to plot Root Locus
t = feedback(z,grav)

%Single Input and Output Tool of transfer function T with unity Feedback of Angle.
sisotool(t)
```

APPENDIX B



GuRoo Linear Simulink Diagram:

Uses S-function block in Simulink to simulate the dynamics of the system.

The s-function files are found attached below.

GuRooSControl.m

```
function [sys,x0,str,ts] = GuRooS(t,x,u,flag)
% S-function sf_aerodyn.M
% This S-function represents the nonlinear aircraft dynamics

% Copyright 1986-2002 The MathWorks, Inc.
% $Revision: 1.5 $  $Date: 2002/04/04 15:22:49 $

switch flag,

  %%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes;

  %%%%%%%%%%%%%%%%%
  % Derivatives %
  %%%%%%%%%%%%%%%%%
  case 1,
    sys=mdlDerivatives(t,x,u);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys=mdlUpdate(t,x,u);

  %%%%%%%%%%
  % Outputs %
  %%%%%%%%%%
  case 3,
    sys=mdlOutputs(t,x,u);

  %%%%%%%%%%%%%%%%%%%%%%%%%
  % GetTimeOfNextVarHit %
  %%%%%%%%%%%%%%%%%%%%%%%%%
  case 4,
    sys=mdlGetTimeOfNextVarHit(t,x,u);

  %%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%
  case 9,
    sys=mdlTerminate(t,x,u);

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl


%=============================================================================
% mdlInitializeSizes
```

```
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded.  This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%
sizes = simsizes;

sizes.NumContStates  = 4;
sizes.NumDiscStates  = 0;
sizes.NumOutputs     = 4;
sizes.NumInputs      = 2;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% Initial conditions
% x = [q1,q2,v1,v2]';
% To linearized the model assume  phi = 0 and theta = 0;

x0  = [pi/2-0.1 0 0 0];

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
ts  = [0 0];

% end mdlInitializeSizes

%
%=============================================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=============================================================================
%
function sys=mdlDerivatives(t,x,u)

%% The state vector is x = [q1,q2,v1,v2]'%%
%% The input vector is u = [t1,t2]'%%

%% DYNAMIC CONSTANTS

g = 9.81;
m1 = 22.967;
m2 = 13.653;
a1 = 0.7;
```

```
a2 = 0.5;
b1 = 10;
b2 = 0.1;

k=abs((- g*( ((m1/2 + m2)*a1*cos(pi/2-0.1)) + ((m2*a2*cos(pi/2))/2)))/(0.1));

%% DYNAMIC CALCULATIONS USING STATE SPACE METHODS

d11 = ( ((m1)/3+m2)*(a1^2) + m2*a1*a2*cos(x(2)) + (m2*(a2^2))/3 );
d12 = ((m2*a1*a2*cos(x(2)))/2 + (m2*(a2^2))/3);
d22 = ((m2*(a2^2))/3);
d21 = ((m2*a1*a2*cos(x(2)))/2 + (m2*(a2^2))/3);

q = (  ((m1*m2*((a1*a2)^2))/9) + (((m2*a1*a2)^2)/3) - ( ((m2*a1*a2*cos(x(2)) )^2 )/4 )   );

bracket1 = ( u(1) + ( m2*a1*a2*sin(x(2)) )*( x(3)*x(4) + (x(4)^2)/2 ) - g*( ((m1/2 +
m2)*a1*cos(x(1))) + ((m2*a2*cos(x(1)+x(2)))/2) ) - b1*x(3) - k*(x(1)-pi/2) );
bracket2 = ( u(2) - (( m2*a1*a2*sin(x(2))*(x(3)^2) )/2) - ((g*m2*a2*cos(x(1)+x(2)))/2) - (b2*x(4)) );

v1dot = (1/q)*(d22*bracket1 - d12*bracket2);
v2dot = (1/q)*(d11*bracket2 - d21*bracket1);

sys = [x(3);x(4);v1dot;v2dot];

% end mdlDerivatives

%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function sys=mdlUpdate(t,x,u)

sys = [];

% end mdlUpdate

%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function sys=mdlOutputs(t,x,u)

sys = x;

% end mdlOutputs

%
%=============================================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.  Note that the result is
% absolute time.  Note that this function is only used when you specify a
% variable discrete-time sample time [-2 0] in the sample time array in
% mdlInitializeSizes.
%=============================================================================
%
```

```
function sys=mdlGetTimeOfNextVarHit(t,x,u)

sampleTime = 1;    %  Example, set the next hit to be one second later.
sys = t + sampleTime;

% end mdlGetTimeOfNextVarHit

%
%=============================================================================
% mdlTerminate
% Perform any end of simulation tasks.
%=============================================================================
%
function sys=mdlTerminate(t,x,u)

sys = [];

% end mdlTerminate
```

GuRooSLinear.m

```
function [sys,x0,str,ts] = GuRooS(t,x,u,flag)
% S-function sf_aerodyn.M
% This S-function represents the nonlinear aircraft dynamics

% Copyright 1986-2002 The MathWorks, Inc.
% $Revision: 1.5 $  $Date: 2002/04/04 15:22:49 $

switch flag,

  %%%%%%%%%%%%%%%%%%
  % Initialization %
  %%%%%%%%%%%%%%%%%%
  case 0,
    [sys,x0,str,ts]=mdlInitializeSizes;

  %%%%%%%%%%%%%%%%%%
  % Derivatives %
  %%%%%%%%%%%%%%%%%%
  case 1,
    sys=mdlDerivatives(t,x,u);

  %%%%%%%%%%
  % Update %
  %%%%%%%%%%
  case 2,
    sys=mdlUpdate(t,x,u);

  %%%%%%%%%%
  % Outputs %
  %%%%%%%%%%
  case 3,
    sys=mdlOutputs(t,x,u);

  %%%%%%%%%%%%%%%%%%%%%%%%%%
  % GetTimeOfNextVarHit %
  %%%%%%%%%%%%%%%%%%%%%%%%%%
  case 4,
```

```
    sys=mdlGetTimeOfNextVarHit(t,x,u);

  %%%%%%%%%%%%
  % Terminate %
  %%%%%%%%%%%%
  case 9,
    sys=mdlTerminate(t,x,u);

  %%%%%%%%%%%%%%%%%%%%
  % Unexpected flags %
  %%%%%%%%%%%%%%%%%%%%
  otherwise
    error(['Unhandled flag = ',num2str(flag)]);

end

% end sfuntmpl


%=============================================================================
% mdlInitializeSizes
% Return the sizes, initial conditions, and sample times for the S-function.
%=============================================================================
%
function [sys,x0,str,ts]=mdlInitializeSizes

%
% call simsizes for a sizes structure, fill it in and convert it to a
% sizes array.
%
% Note that in this example, the values are hard coded.  This is not a
% recommended practice as the characteristics of the block are typically
% defined by the S-function parameters.
%
sizes = simsizes;

sizes.NumContStates  = 4;
sizes.NumDiscStates  = 0;
sizes.NumOutputs     = 4;
sizes.NumInputs      = 2;
sizes.DirFeedthrough = 1;
sizes.NumSampleTimes = 1;   % at least one sample time is needed

sys = simsizes(sizes);

%
% Initial conditions
% x = [q1,q2,v1,v2]';
% To linearized the model assume  phi = 0 and theta = 0;

x0  = [-0.1 0 0 0];

%
% str is always an empty matrix
%
str = [];

%
% initialize the array of sample times
%
```

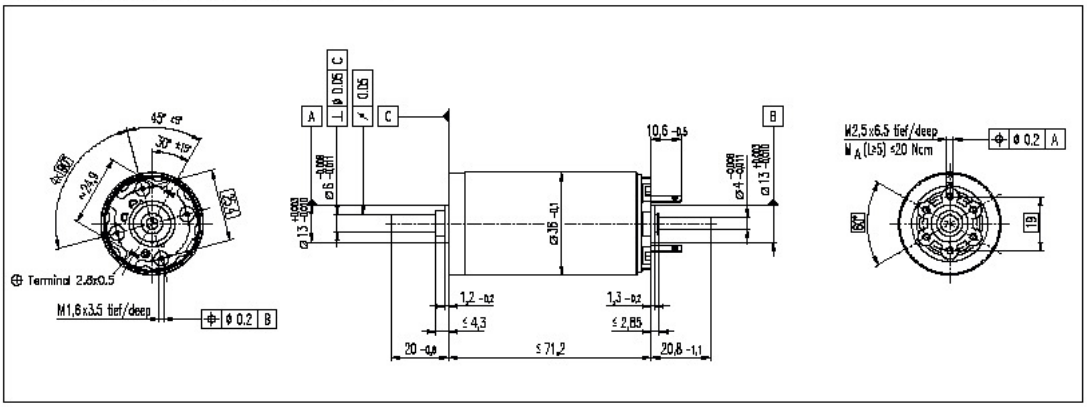
```
ts  = [0 0];

% end mdlInitializeSizes

%
%=============================================================================
% mdlDerivatives
% Return the derivatives for the continuous states.
%=============================================================================
%
function sys=mdlDerivatives(t,x,u)

%% The state vector is x = [q1,q2,v1,v2]'%%
%% The input vector is u = [t1,t2]'%%

%% LINEAR CONSTANTS

g = 9.81;
m1 = 22.967;
m2 = 13.653;
a1 = 0.7;
a2 = 0.5;
b1 = 10;
b2 = 0.1;
k=(g*(m1/2+m2)*a1);

%% LINEARISED STATE SPACE DYNAMIC EQUATIONS

q = ((m2*(a1*a2)^2)*(m1/9 + m2/12));
d11 = ((m1/3 + m2)*a1^2 + m2*a1*a2 + (m2*a2^2)/3);
d12 = ((m2*a1*a2)/2 + (m2*a2^2)/3);
d21 = d12;
d22 = (m2*a2^2)/3;

% x = [q1 q2 v1 v2]T
% xdot = [v1 v2 a1 a2]T

a11 = ((1/q)*(d22*(g*(m1/2+m2)*a1 + g*m2*a2/2-k)-(d12*g*m2*a2)/2));
a22 = ((1/q)*(d22*g*m2*a2/2 - d12*g*m2*a2/2));
a3 = -(d22/q)*b1;
a4 = (d12/q)*b2;

a5 = ((1/q)*(-d21*(g*(m1/2+m2)*a1 + g*m2*a2/2-k)+(d11*g*m2*a2)/2));
a6 = ((1/q)*(-d21*g*m2*a2/2 + d11*g*m2*a2/2));
a7 = (d21/q)*b1;
a8 = -(d11/q)*b2;

A = [0 0 1 0
   0 0 0 1
   a11 a22 a3 a4
   a5 a6 a7 a8];

% u = [t1 t2]T

B = [0 0
   0 0
   d22/q -d12/q
   -d21/q d11/q];

sys = A*x + B*u;
```

```
% end mdlDerivatives

%
%=============================================================================
% mdlUpdate
% Handle discrete state updates, sample time hits, and major time step
% requirements.
%=============================================================================
%
function sys=mdlUpdate(t,x,u)

sys = [];

% end mdlUpdate

%
%=============================================================================
% mdlOutputs
% Return the block outputs.
%=============================================================================
%
function sys=mdlOutputs(t,x,u)

sys = x;

% end mdlOutputs

%
%=============================================================================
% mdlGetTimeOfNextVarHit
% Return the time of the next hit for this block.  Note that the result is
% absolute time.  Note that this function is only used when you specify a
% variable discrete-time sample time [-2 0] in the sample time array in
% mdlInitializeSizes.
%=============================================================================
%
function sys=mdlGetTimeOfNextVarHit(t,x,u)

sampleTime = 1;    %  Example, set the next hit to be one second later.
sys = t + sampleTime;

% end mdlGetTimeOfNextVarHit

%
%=============================================================================
% mdlTerminate
% Perform any end of simulation tasks.
%=============================================================================
%
function sys=mdlTerminate(t,x,u)

sys = [];

% end mdlTerminate
```

APPENDIX C

maxon DC motor

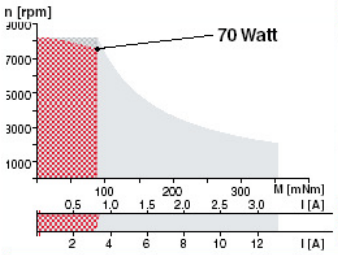
RE 36
Ø36 mm, Graphite Brushes, 70 Watt



Motor Data: **Order Number**

			118797	118798	118799	118800	118801	118802	118803	118804	118805	118806	118807	118808	118809	118810
1	Assigned power rating	W	70	70	70	70	70	70	70	70	70	70	70	70	70	70
2	Nominal voltage	Volt	18.0	24.0	32.0	42.0	42.0	48.0	48.0	48.0	48.0	48.0	48.0	48.0	48.0	48.0
3	No load speed	rpm	6410	6210	6790	7020	6340	6420	5220	4320	3450	2830	2280	1780	1420	1180
4	Stall torque	mNm	730	783	832	865	786	785	627	504	403	326	258	198	158	127
5	Speed/torque gradient	rpm/mNm	8.96	8.05	8.27	8.19	8.14	8.25	8.41	8.65	8.67	8.80	8.96	9.17	9.21	9.51
6	No load current	mA	147	105	89	70	61	55	42	33	25	20	15	12	9	7
7	Starting current	A	27.8	21.5	18.7	15.3	12.6	11.1	7.22	4.80	3.06	2.04	1.30	0.784	0.501	0.334
8	Terminal resistance	Ohm	0.647	1.11	1.71	2.75	3.35	4.32	6.65	10.00	15.7	23.5	36.8	61.3	95.8	144
9	Max. permissible speed	rpm	8200	8200	8200	8200	8200	8200	8200	8200	8200	8200	8200	8200	8200	8200
10	Max. continuous current	A	3.14	2.44	1.99	1.59	1.44	1.27	1.03	0.847	0.679	0.556	0.445	0.346	0.277	0.226
11	Max. continuous torque	mNm	82.4	88.8	88.5	89.8	90.4	90.1	89.8	89.0	89.2	88.8	88.1	87.3	87.2	85.8
12	Max. power output at nominal voltage	W	119	125	146	157	129	131	84.9	56.4	36.0	23.9	15.2	9.09	5.78	3.82
13	Max. efficiency	%	84	85	86	86	86	86	85	84	82	81	79	77	75	72
14	Torque constant	mNm/A	26.3	36.4	44.5	56.6	62.6	70.7	86.9	105	131	160	198	253	315	380
15	Speed constant	rpm/V	364	263	215	169	152	135	110	90.9	72.7	59.8	48.2	37.8	30.3	25.1
16	Mechanical time constant	ms	6	6	6	6	6	6	6	6	6	6	6	6	6	6
17	Rotor inertia	gcm²	62.0	67.7	65.2	65.4	65.6	64.6	63.3	61.5	61.3	60.3	59.2	57.8	57.5	55.7
18	Terminal inductance	mH	0.10	0.20	0.30	0.49	0.60	0.76	1.15	1.68	2.62	3.87	5.96	9.70	15.10	21.90
19	Thermal resistance housing-ambient	K/W	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4
20	Thermal resistance rotor-housing	K/W	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4
21	Thermal time constant winding	s	39	43	41	41	41	41	40	39	39	38	37	36	36	35

Operating Range



Comments

Details on page 36

Recommended operating range

Continuous operation
In observation of above listed thermal resistances (lines 19 and 20) the maximum permissible rotor temperature will be reached during continuous operation at 25°C ambient.
= Thermal limit.

Short term operation
The motor may be briefly overloaded (recurring).

118804 Motor with high resistance winding
118797 Motor with low resistance winding

Stock program
Standard program
Special program (on request!)

- Axial play 0.05 - 0.15 mm
- Max. **ball bearing** loads
  axial (dynamic)
  not preloaded 5.6 N
  preloaded 2.4 N
  radial (5 mm from flange) 28 N
  Press-fit force (static) 110 N
  same as above, shaft supported 1200 N
- Radial play **ball bearings** 0.025 mm
- Ambient temperature range -20/+100°C
- Max. rotor temperature +125°C
- Number of commutator segments 13
- Weight of motor 350 g
- Values listed in the table are nominal.
  For applicable tolerances (see page 33) and additional details please request our computer printout.

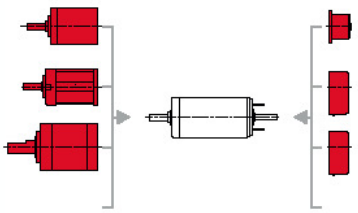
⚠ Tolerances may vary from the standard specification.

maxon Modular System

Planetary Gearhead
Ø32 mm
0.75-4.5 Nm
Details page 161

Planetary Gearhead
Ø32 mm
0.4-2 Nm
Details page 163

Planetary Gearhead
Ø42 mm
3-15 Nm
Details page 165



DC Tacho
Ø22 mm
0.52 V
Details page 172

Digital Encoder HP HEDS 5540
500 CTP, 3 channels
Details page 176

Digital Encoder HP HEDL 5540
500 CTP, 3 channels
Details page 178

APPENDIX D

ForceSlave.h

```
/*****************************************************************************
 * Copyright 2003, Gordon Wyeth
 *****************************************************************************
 *     File: ForceSlave.h
 *   Author: Toby Low
 *  Project: Humanoid 2.0 Dynamechs 4.0
 *  Created: 22nd Augst 2003
 *  Summary: Input forces into simulator
  *****************************************************************************/

#ifndef _ForceSlave_H_
#define _ForceSlave_H_

#include <stdio.h>
#include "../MasterSlave.h"
#include "dm.h"

class ForceSlave : public SlaveControl
{
protected:

                    SpatialVector inputvector;

                    float stageStart;
                    bool  stageAct;
                    float stageTime;

                    float PitchAngle;
                    float RollAngle;

public:
        ////////////////////////
        /// Constructor (default, do not use)
        ////////////////////////
        ForceSlave ();

        ////////////////////////
        /// Constructor (with pointer to Master Control)
        /// Use this constructor to instance a class since
        /// you need a pointer to the master control
        ////////////////////////
        ForceSlave (char* name, MasterControl *master);

        ////////////////////////
        /// Deconstructor
        ////////////////////////
        virtual ~ForceSlave ();

        ////////////////////////
        /// This function is called when the slave is registered
        /// with a master controller and finished is false.
        /// Called by update function of the master controller
        ////////////////////////
        virtual void controlFunction ();
```

```
};

#endif // ForceSlave
```

ForceSlave.cpp

```
/*****************************************************************************
 * Copyright 2003, Gordon Wyeth
 *****************************************************************************
 *    File: ForceSlave.cpp
 *  Author: Toby Low
 *  Project: Humanoid 2.0 Dynamechs 4.0
 *  Created: 16th Sept 2003
 *  Summary: Force Input Controller
 *          ToUse: Set external force using SpatialVector(mx,my,mz,x,y,z) WRT link coordinates.
 *                          I.e. SpatialVector VECTOR_YOU_SET = {set,set,set,set,set,set};
 *                          ForceLink(0 for force on or 1 for force off, SIM_NO._JOINT,
VECTOR_YOU_SET);
 *                          NB: Only has one stage. Can implement more later.
 *****************************************************************************/

#include <math.h>
#include <stdio.h>
#include "../Common/Slaves/ForceSlave.h"
#include "../Common/jointnum.h"
#include "../Common/lowlevel.h"
#include "../Common/guroodef.h"
#include "../Common/logdata.h"

#include <dm.h>

#ifdef WIN32
#include <memory.h>
#include <windows.h>
#endif

#include <stdlib.h>
#include <string.h>

////////////////////////////////////////
// Default constructor (do not use)
////////////////////////////////////////
ForceSlave::ForceSlave ()
{
 name = "unnamed";
 printf ("WARNING: ForceSlave::ForceSlave (), slave should be created with a master.\n");
}

////////////////////////////////////////
// Constructor, creates a instance with a name and pointer to a MasterController
////////////////////////////////////////
ForceSlave::ForceSlave (char* name, MasterControl *master) : SlaveControl (name, master)
{
                float baseTime = master->getTime ();

                stageAct = false;
                stageStart = baseTime + 0.0f ;
                stageTime =  0.09f;
```

```
}

//////////////////////////////////////
// Deconstructor
//////////////////////////////////////
ForceSlave::~ForceSlave ()
{

}

//////////////////////////////////////
// Control function, called by MasterControl
// All the gait generation work is done here
//////////////////////////////////////
void ForceSlave::controlFunction ()
{

        PitchAngle = IMUinfo(0);
        RollAngle = IMUinfo(1);
        //logPitchAngle(PitchAngle,RollAngle);

        if(stageAct){
                SpatialVector inputvector = {0,0,0,0,100,0};
                ForceLink(0, SIM_TORSO_FWD, inputvector);
        }else{
                SpatialVector inputvector = {0,0,0,0,0,0};
                ForceLink(1, SIM_TORSO_FWD, inputvector);

        }

   // Stage is finished
   if (stageAct == true && (master->getTime () >= (stageStart+stageTime)))
   {
     // Stop the stage
     stageAct = false;
   }
   else
   // Stage is ready to start
   if (stageAct == false && (master->getTime () >= stageStart) &&
       stageStart >= 0 && (master->getTime () <= (stageStart + stageTime)))
   {
     // Start the stage
     stageAct = true;
   }

}
```

Balance.h

```
#ifndef _BALANCE_H_
#define _BALANCE_H_

//Balance and Motor Constants
#define PITCH 0
#define ROLL 1
```

```
//*************************************************************************************

//Comment this code to swap between gait running balance and simulation running code.
//Defined means running simulation mode.

#define SIMULATION

//*************************************************************************************

//Old motor constants
//#define KA 0.0445
//#define KB 0.0444

#define KA 5                    //Overall KA, KB and RA
#define KB 7.02
#define RA 1.71


#define DGAIN 7
#define PGAIN 42
#define IGAIN 0.7
#define DGAINROLL 7
#define PGAINROLL 42
#define IGAINROLL 0.7

#define NRATIO 156
#define Eff 0.72
#define G -9.81
#define M2 22
#define A2 0.5
#define M1 13
#define A1 0.7
#define INERTIA 1.53

#define EATORADFACTOR 0.14245


#define OLDBALANCE
//#define COGBalance
//#define ROLLANKLES
//#define ANKLES

#include "../MasterSlave.h"

class Balance : public SlaveControl
{
protected:

                        float DesiredPitchAngle;

public:
        ////////////////////////
        /// Holds the desired factor settings
        /// This is accessible by anyone
        ////////////////////////
        float SetDesiredPitchAngle;
```

```
public:
        //////////////////////
        /// Constructor (default, do not use)
        //////////////////////
        Balance ();

        //////////////////////
        /// Constructor (with pointer to Master Control)
        /// Use this constructor to instance a class since
        /// you need a pointer to the master control
        //////////////////////
        Balance (char* name, MasterControl *master);

        //////////////////////
        /// Deconstructor
        //////////////////////
        virtual ~Balance ();

        //////////////////////
        /// This function is called when the slave is registered
        /// with a master controller and finished is false.
        /// Called by update function of the master controller
        //////////////////////
        virtual void controlFunction ();

};

#endif;
```

Balance.cpp

```
//-------------------------------------------------------------------------------Balance Control System-----------
-----------
//
//          By:       Toby Low        Date: 8/9/03        Version: 1.1
//
//          Function Name:          Balance Slave
//          No: of Inputs:          0
//          Input:                          None
//          Returns:                Nothing
//          Description:            Controlled by master - Balance Control System
//

//-------------------------------------------------------------------------------------------------------------------
----



#include <math.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "balance.h"

#include "../jointnum.h"
#include "../guroodef.h"
#include "../logdata.h"
```

```
#ifdef SIMULATION

#include <dm.h>
#include "../lowlevel.h"

#else

#include <iostream.h>
#include <conio.h>
#include <iostream>

#endif

#ifdef WIN32
        #include <windows.h>
#endif

extern int actual_joint_vel[NUMBER_MOTORS];
extern int actual_joint_position[NUMBER_MOTORS];
extern float eIMU_data[eIMU_DATA_SIZE];
float Integral, Derivative;
float Integral2, Derivative2;
float LastVal=0, LastVal2=0;
float OldErr2=0, OldErr=0;


//------------------------------------------------------------------------------------------------------------------------
----
// Default constructor (do not use)
//------------------------------------------------------------------------------------------------------------------------
----
Balance::Balance ()
{
 name = "unnamed";
 printf ("WARNING: Balance::Balance (), slave should be created with a master.\n");
}


//------------------------------------------------------------------------------------------------------------------------
----
// Constructor, creates a instance with a name and pointer to a MasterController
//------------------------------------------------------------------------------------------------------------------------
----
Balance::Balance (char* name, MasterControl *master) : SlaveControl (name, master)
{

SetDesiredPitchAngle = -0.035;

}

//------------------------------------------------------------------------------------------------------------------------
----
// Deconstructor
//------------------------------------------------------------------------------------------------------------------------
----
Balance::~Balance ()
{
 name = "";
 master = NULL;
```

```
}


//-------------------------------------------------------------------------------------------------------------------
----
// Control function, called by MasterControl
// All the gait generation work is done here
//-------------------------------------------------------------------------------------------------------------------
----
void Balance::controlFunction ()
{

#ifndef OLDBALANCE

        float PitchAngle = 0, RollAngle = 0; //Pitch value in degrees.
        float DesiredPitchAngle, DesiredRollAngle;
        float DesiredTorsoVel;
        int   DesiredEncs;

#else

        float PitchErr, RollErr;
        float PitchAngle = 0; //Pitch value in degrees.
        float RollAngle = 0;
        float DesiredPitchAngle, DesiredRollAngle;
        float TorqueReq, TorqueMotor, TorqueReq2, TorqueMotor2;

        float RadVelRoll =0 ,RadVelPitch = 0;
        float NewSpeedPitch,NewSpeedRoll;
        float gravcomppitch, gravcomproll;

        float lowerangle;
        float tempdes;
        float DesCOG;

#endif


// If Simulation defined, take IMU data from Dynamechs else get data from IMU Server
#ifdef SIMULATION

        PitchAngle = IMUinfo(PITCH);
        RollAngle = IMUinfo(ROLL);
        logPitchAngle(PitchAngle,RollAngle);
        cerr << "PitchAngleSim = " << PitchAngle << endl ;
        cerr << "RollAngleSim = " << RollAngle << endl ;


#else

        RollAngle = (eIMU_data[0])*2*PI/360;
        PitchAngle = (eIMU_data[1])*2*PI/360;
        cout << "PitchAngle = " << PitchAngle << endl ;
        cout << "RollAngle = " << RollAngle << endl ;

#endif


//PITCH CONTROL LOOP
```

```
	//COG Calculation

#ifdef COGBalance
	lowerangle = -PitchAngle - ENC2RAD(actual_joint_position[TORSO_FWD]);
	cerr << "lowerangle: " << lowerangle  << endl;
	cerr << "PitchAngle: " << PitchAngle  << endl;

	tempdes = acos(   (1+(M1/(2*M2)))*(A1/A2)*cos(PI/2 - lowerangle));

	DesCOG = 0.008;

	tempdes = acos((2/A2)*( A1*cos(PI/2 - lowerangle) - DesCOG + (M1/M2)*( (A1/2)*cos(PI/2
- lowerangle)-DesCOG )));

	DesiredPitchAngle = (PI/2 - tempdes);
	cerr << "DesiredPitchAngle: " << DesiredPitchAngle  << endl;

#else

	DesiredPitchAngle = SetDesiredPitchAngle;
	cout << "DesiredPitchAngle = " << DesiredPitchAngle  << endl;

#endif

	PitchErr = DesiredPitchAngle - PitchAngle;
	Integral = Integral + PitchErr;
	Derivative = PitchErr - OldErr;
	OldErr = PitchErr;

	//cerr << "PitchAngle: " << PitchAngle  << endl;
	//cerr << "Integral: " << Integral  << endl;
	//cerr << "Derivative: " << Derivative  << endl;
	//cerr << "PitchErr: " << PitchErr  << endl;

	gravcomppitch = 0;//(G*M2*A2*sin(PitchAngle))/2;
	//cerr << "gravcomp: " << gravcomp  << endl;

	TorqueReq = PitchErr*PGAIN + Integral*IGAIN + Derivative*DGAIN + gravcomppitch;
	TorqueMotor = TorqueReq;///(NRATIO*Eff);
	//cerr << "TorqueMotor: " << TorqueMotor  << endl;

	RadVelPitch = ENC2RAD(actual_joint_vel[TORSO_FWD]);


#ifdef SIMULATION
	cerr << "RadVelPitch: " << RadVelPitch  << endl;
#else
	cout << "RadVelPitch: " << RadVelPitch  << endl;
#endif

	//cerr << "TorqueReq: " << TorqueReq  << endl;
	//cerr << "NewSpeedPitch: " << NewSpeedPitch  << endl;
	//cerr << "actual_joint_position: " << actual_joint_position[TORSO_FWD] << endl;

	NewSpeedPitch = BOARD_DELAY*EATORADFACTOR*(TorqueMotor*RA/KA +
KB*RadVelPitch); // vel of motor in rad/s need rad/control loop?

	//cerr << "NewSpeedPitch: " << NewSpeedPitch << endl;
	//cerr << "KB*NRATIO*RadVelPitch: " << KB*RadVelPitch << endl;
```

```
        //cerr << "TorqueMotor*RA/KA : " << TorqueMotor*RA/KA  << endl;
        //cerr << "NewSpeedPitch: " << NewSpeedPitch << endl;

#ifndef SIMULATION
        cout << "NewSpeedPitch = " << -NewSpeedPitch << endl;
#endif

        master->modifyJointVel(TORSO_FWD, -NewSpeedPitch, JOINT_MODIFY_ADD);

#ifdef ANKLES

        master->modifyJointVel(LEFT_ANKLE_FWD, 0.3*NewSpeedPitch,
JOINT_MODIFY_ADD);
        master->modifyJointVel(RIGHT_ANKLE_FWD, 0.3*NewSpeedPitch,
JOINT_MODIFY_ADD);


#endif


        //ROLL CONTROL LOOP

        DesiredRollAngle = 0.0;

        RollErr = DesiredRollAngle - RollAngle;
        Integral2 = Integral2 + RollErr;
        Derivative2 = RollErr - OldErr2;
        OldErr2 = RollErr;

        //cerr << "RollAngle: " << RollAngle  << endl;
        //cerr << "Intergral2: " << Integral2  << endl;
        //cerr << "Derivative2: " << Derivative2  << endl;
        //cerr << "RollErr: " << RollErr  << endl;

        gravcomproll = 0;

        TorqueReq2 = RollErr*PGAINROLL + Integral2*IGAINROLL +
Derivative2*DGAINROLL + gravcomproll;
        TorqueMotor2 = TorqueReq2;

        RadVelRoll = ENC2RAD(actual_joint_vel[TORSO_SIDE]);

#ifdef SIMULATION
        cerr << "RadVelRoll: " << RadVelRoll  << endl;
#else
        cout << "RadVelRoll: " << RadVelRoll  << endl;
#endif

        //cerr << "RadVelRoll: " << RadVelRoll  << endl;

        NewSpeedRoll = BOARD_DELAY*EATORADFACTOR*(TorqueMotor2*RA/KA +
KB*RadVelRoll);
        //cerr << "KB*NRATIO*RadVelRoll: " << KB*RadVelRoll << endl;
        //cerr << "TorqueMotor2*RA/KA: " << TorqueMotor2*RA/KA << endl;
        //cerr << "NewSpeedRoll: " << NewSpeedRoll << endl;

#ifndef SIMULATION
        cout << "NewSpeedRoll = " << NewSpeedRoll << endl ;
#endif
```

```
//logs balance files
        log_Balance(NewSpeedPitch,NewSpeedRoll,RadVelRoll,RadVelPitch,RollErr,Integral2,Deri
vative2,gravcomproll,PitchErr,Integral,Derivative,gravcomppitch,PitchAngle,RollAngle);



#ifdef ROLLANKLES



        master->modifyJointVel(LEFT_ANKLE_SIDE, 0.3*NewSpeedRoll,
JOINT_MODIFY_ADD);
        master->modifyJointVel(RIGHT_ANKLE_SIDE, 0.3*NewSpeedRoll,
JOINT_MODIFY_ADD);

#endif



}
```

lowlevel_sim.cpp

```
//-------------------------------------------------------------------------------Active Balance Functions-----------
-----------
//
//      By: Toby Low
//      Includes:
//              FindVel Function:
        Finds the Velocity of a Joint in Rad/s given a Link number.
//              IMU Retrieve Sensor Information Function: Returns the IMU data, takes in a
constant value. 0 - Pitch, 1 - Roll.
//              IMU Filtering Function:                                         Gets called
as often as possible to update the IMU Angular Velocity used to
//
                derive the angular positions of the robots head.
//              Force Input Function:                                           Gives the
link a jolt of force.
//
//-----------------------------------------------------------------------------------------------------------------------
----

float HeadVelVector[6];
float HeadRot[2][2];
int count=0;

float headpitch=0;
float headroll=0;
//float oldvel[5];
int count2=0;
int window=0;
float oldtime=0, newtime=0, timer=0, avgvel=0;
vector <double> VelFilter;
vector <double> VelRollFilter;
float AngVelPitch=0;
float AngVelRoll=0;
float OldAngVel=0;
float RollOldAngVel=0;
```

```
float Sum=0;
float RollSum=0;


//-------------------------------------------------------------------------------IMU Sensor Information Retrieval Function--------
//
//		By: Toby Low	Date: 6/9/03		Version: 1.0
//
//		Function Name:		IMUinfo(int value)
//		No: of Inputs:		1
//		Input:				Int - 0 for Pitch, 1 for Roll
//		Returns:			Float - Pitch and Roll Positions Respectively
//		Description:		Uses angular vel from IMU Filtering Function to find and return the pitch and roll angular positions
//							of the GuRoo head simulating the IMU device that will be implemented in the GuRoo.
//

//-------------------------------------------------------------------------------------------------------------------------------

float IMUinfo(int value){

		float temppitch = 0;		//Temporary Pitch Variable

switch(value)
		{

		//Retrieves Pitch Infomation
		case 0: {


				headpitch = headpitch + OldAngVel*CENTRAL_SPEED;	//Integrate ang vel to find position, CENTRAL SPEED is time step
				OldAngVel = AngVelPitch;
				//cerr << "AngVelPitch: " << AngVelPitch << endl;
				//cerr << "headpitch: " << headpitch << endl;

				return headpitch;

				}

		//Retrieves Roll Information
		case 1: {
				headroll = headroll + RollOldAngVel*CENTRAL_SPEED;
		//Integrate ang vel to find position, CENTRAL SPEED is time step
				RollOldAngVel = AngVelRoll;
				//cerr << "AngVelRoll: " << AngVelRoll << endl;
				//cerr << "headroll: " << headroll << endl;

				return headroll;

				}

		//Default Case Arguement
		default:  {
				cerr << "Error in IMU retrival" << endl;	//If input value incorrect or out of range, output error msg.
```

```
                return 0;
            //Return the value of 0 if incorrect value.
                }

        }

}



//-------------------------------------------------------------------------Joint Angular Velocity Retrival Function-------
//
//      By: Toby Low    Date: 6/9/03    Version: 1.0
//
//      Function Name:          FindVel(int j)
//      No: of Inputs:          1
//      Input:                      Int - Simulation Joint Number
//      Returns:                Float - Joint Velocity in Rad/s Respectively
//      Description:            Used for testing purposes - finds the velocity of any joint in the robot.
//

//-------------------------------------------------------------------------------------------------------------------

float FindVel(int j){

        float joint_pos[1];
            //Variable to take in joint positions, array of 2 elements

                    //this is a 2-element array due to the number of degrees of

                    //freedom for each link

        float joint_vel[1];
        //variable to take in joint velocities, again an array of 2 elements

            robot_link[j]->getState(joint_pos, joint_vel);      //gets the velocity and position of a joint from the simulator

        return(joint_vel[0]);
        //Return joint velocity in rad/s

}
```

```
//-------------------------------------------------------------------------IMU Filtering and Dynamechs
Simulation---------
//
//          By:       Toby Low          Date: 6/9/03       Version: 1.0
//
//          Function Name:              RetrieveIMU()
//          No: of Inputs:              0
//          Input:
//          Returns:
//          Description:                Averages IMU data, Filter Size is defined below.
//

//-------------------------------------------------------------------------------------------------------------
----

#define FILTER_SIZE 20

void RetrieveIMU(){

SpatialVector HeadVel;
//RotationMatrix HeadRot;                       //used to extract the head angles

//dmglGetSysTime(&INUcurrTime);
          int i = SIM_HEAD;
          dmArticulation    *guroo;
          guroo = dynamic_cast<dmArticulation*>(G_robot);
          dmLink* link = guroo->getLink(i);
          dmRigidBody *rigidBody = dynamic_cast<dmRigidBody*>(link);

          if (rigidBody != NULL) { // check to see if link is out of range
                    const dmABForKinStruct* HeadDynamics = guroo->getForKinStruct(i);
          //count++;
          //cerr << "    Count: " << count << endl;
          //cerr<<"First Log"<<endl;
          //Rates and Velocities of Head from Simulator

          SpatialVector HeadVel =   {         (Float)HeadDynamics->v[0],
                                                        (Float)HeadDynamics->v[1],
                                                        (Float)HeadDynamics->v[2],
                                                        (Float)HeadDynamics->v[3],
                                                        (Float)HeadDynamics->v[4],
                                                        (Float)HeadDynamics->v[5]
                                                  };


                    HeadVelVector[0] = HeadVel[0];
                    HeadVelVector[1] = HeadVel[1];
                    HeadVelVector[2] = HeadVel[2];
                    HeadVelVector[3] = HeadVel[3];
                    HeadVelVector[4] = HeadVel[4];
                    HeadVelVector[5] = HeadVel[5];


          //Filtering of the Velocities obtained from the simulator.
                    VelRollFilter.push_back(HeadVel[1]);
                    VelFilter.push_back(HeadVel[2]);
                    //cerr << "HeadVel[2] " << HeadVel[2] << endl;

                    if(VelRollFilter.size() > FILTER_SIZE){
```

```
                        VelRollFilter.erase(VelRollFilter.begin());
                        RollSum = 0;

                        for(int y=0;y < VelRollFilter.size();y++){
                                RollSum = RollSum + VelRollFilter.at(y);
                                //cerr << "y " << y << endl;
                        }

                        AngVelRoll = RollSum / VelRollFilter.size();
                        //cerr << "AngVelRoll " << AngVelRoll << endl;
                }

                if(VelFilter.size() > FILTER_SIZE){

                        VelFilter.erase(VelFilter.begin());
                        Sum = 0;

                        for(int y=0;y < VelFilter.size();y++){
                                Sum = Sum + VelFilter.at(y);
                                //cerr << "y " << y << endl;
                        }

                        AngVelPitch = Sum / VelFilter.size();
                        //cerr << "AngVelPitch " << AngVelPitch << endl;
                }
}
}


//-------------------------------------------------------------------------------Dynamech Force Simulation-----------
//
//      By:     Toby Low        Date: 7/9/03    Version: 1.0
//
//      Function Name:          ForceLink(int end, int link, SpatialVector forcevector)
//      No: of Inputs:          3
//      Input:                          Int end - Set to 1 if want to remove forces
//                                      Int link - Set to link that force will be applied to.
//                                      SpatialVector forcevector - Set to the force
vector that will be applied to link - (mx,my,mz,x,y,z)
//      Returns:                Nothing
//      Description:            Applies force to link to simulate pushes and any external forces.
//

//-------------------------------------------------------------------------------------------------------------------
----

void ForceLink(int end, int link, SpatialVector forcevector){

        //Define ZeroForceVector
        SpatialVector zerovector = {0,0,0,0,0,0};

        //Dynamechs Setup
        dmArticulation    *guroo;
        guroo = dynamic_cast<dmArticulation*>(G_robot);
        dmLink* ForceLink = guroo->getLink(link);
        dmRigidBody *rigidBody = dynamic_cast<dmRigidBody*>(ForceLink);

        if (rigidBody != NULL) { // check to see if link is out of range
```

```
                    if(end==0){
                            rigidBody->setExternalForce(forcevector);           //Set external force
using SpatialVector(mx,my,mz,x,y,z) WRT link coordinates.
                    }else if(end==1){
                            rigidBody->setExternalForce(zerovector);            //Set external force on
link to zero.
                    }
            }
}
```

APPENDIX E

Electronically attached items are located within the CDROM below. This CD includes:

- MATLAB code for linearising and generating the system transfer function
- MATLAB Simulink models of the linear and non-linear system with the design control system.
- MATLAB dynamic S-Functions files used in the Simulink models.
- C++ Source Code for the GuRoo Simulator and Gait Generation Code. This code includes the complete balance implementation, IMU and force input simulation code as well.
- PDF Copy of this report.
- MATLAB graphs and workspace of results from simulation and the real robot.
- IMU Manual Version 0.47 Public.
- Videos of the GuRoo crouching and being pushed.