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

Active Balance Control for a Humanoid Robot

Ian Joseph Marshall

Bachelor of Engineering
Electrical (Honours)

October 16, 2002

13 Judith St
Dorrington, QLD 4060

16th October 2002

Head of School
School of Information Technology and Electrical Engineering
University of Queensland
St Lucia, QLD 4072

Dear Professor Kaplan,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Electrical Engineering, I present the following thesis entitled *"Active Balance Control for a Humanoid Robot"*. The project was completed under the supervision of Dr Gordon Wyeth.

I declare that the work submitted in this thesis has not previously been submitted for a degree at the University of Queensland or any other institution. To the best of my knowledge, all material in this document written or published by any other person has been appropriately referenced.

Yours sincerely,

Ian Marshall
33582399

Abstract

The purpose of the GuRoo Humanoid Project is to build an anthropomorphic robot that will be competitive in the humanoid league division of the annual RoboCup Competition. The scope of this thesis is to design and implement an Active Balance Control System that will enable the robot to remain upright despite adverse disturbances such as external forces and uneven terrain.

The course of the project saw the development of the robot from a simulated on-paper design to a functional hardware prototype capable of performing simple movements. Constraints due to a limited amount of feedback information significantly influenced the complexity of the Balance Control System.

The proposed structure of the balance system is broken into three modules:

- Supporting base Model
- Centre of Mass (C.o.M.) Model
- Attitude Control System

Actual position data is passed to the central controller via a serial link from the distributed controller boards and CAN network. The attitude control module uses the state information developed from the centre of mass model to define new desired joint angles for maintaining robot stability. All calculations are executed with respect to the centre of the support polygon, which is determined purely through geometrical analysis.

The robot is modelled as a 3D Linear Inverted Pendulum (3D-LIPM) which consists of a point mass (C.o.M.) and a massless telescopic leg extending from the centre of the supporting base to the centre of mass of the robot. Zero Moment Point criteria is considered whereby new desired joint positions for the ankles are defined by considering the effects of inertial, gravitational and reaction forces on the inverted pendulum.

The design was successfully developed, simulated, and evaluated with the robot capable of remaining stable for a series of simple movements. Due to errors in the serial feedback of actual joint position data, the design could not be implemented and tested on real hardware.

The proposed design forms a solid platform that will form the basis of further testing, development and implementation.

Acknowledgements

I would like to thank the following people for their contribution towards the completion of this thesis:

- My supervisor, Dr Gordon Wyeth, for his guidance and patience throughout the course of the project.

- Damien Kee for his inspiration and assistance.

- The other GuRoo team members, Adam Drury and Andrew Hood for their help, friendship and dedication to the project.

- The other late-night lab-inhabitants, Chris and Rob, for providing plenty of laughter at the strangest of times.

- My Family, Beck, and friends for their valuable support throughout the year.

Table of Contents

CHAPTER 1 – INTRODUCTION 1

1.1 Humanoid Robots 1

1.2 RoboCup Competition 2

1.3 Walking Algorithms and the Difficulty of Dynamic Balance Control 3

1.4 Motivation 4

1.5 Project Outcomes and Achievements 5

1.6 Chapter outline 6

CHAPTER 2 – LITERATURE REVIEW 7

2.1 Previous Work (GuRoo) 7

2.2 Other Humanoids in General 9

2.3 Description of Existing Humanoid Balance Systems 10
  2.3.1 Defining Key Concepts 10
  2.3.2 Defining Stability Conditions 12
  2.3.3 Review of Methods of Balance Control 14
  2.3.4 Inverted Pendulum Modes 14

CHAPTER 3 – SPECIFICATIONS 18

3.1 Introduction 18

3.2 Preparation for RoboCup 18

3.3 Development of an Active Balance Control System 19
  3.3.1 Research of Balance Control Methodology 20
  3.3.2 Supporting Base Model 20
  3.3.3 Centre of Mass Model 21
  3.3.4 Attitude Control 21

CHAPTER 4 – ANALYSING THE HARDWARE 23

4.1 Mechanical Design 23

4.2 Distributed Control Network 27

CHAPTER 5 – MATHEMATICAL MODELS 29

5.1 Linear Inverted Pendulum..29

5.2 Zero Moment Point..31

CHAPTER 6 - SOFTWARE DESIGN AND IMPLEMENTATION **36**

6.1 Introduction..36

6.2 Initialisation and Matrix Transformations...37

6.3 Determining the Supporting Base...38

6.4 Centre of Mass..40

6.4.1 Robot Centre of Mass Model..40

6.4.2 Simulator Centre of Mass Model..43

6.5 Attitude Control..43

CHAPTER 7 - SYSTEM EVALUATION..**46**

7.1 Determining the Supporting Base...46

7.2 Centre of Mass Calculations...48

7.3 Attitude Control..53

CHAPTER 8 - CONCLUSIONS AND FUTURE DEVELOPMENTS **58**

8.1 Conclusions...58

8.2 Future Developments..59

APPENDIX A : CENTRE OF MASS LOCATIONS......................................**63**

APPENDIX B : C/C++ CODE..**64**

B.1 - Balance.h...64

B.2 - Balance.c...65

B.3 – Central.c (extract)...87

APPENDIX C : SIMULATOR CENTRE OF MASS CODE **88**

APPENDIX D : CD...**89**

D.1 – MATLAB Code..89

List of Figures

FIGURE 1: GUROO SIMULATOR 8
FIGURE 2: GUROO HARDWARE 8
FIGURE 3: SONY'S ASIMO & HONDA'S SDR-4X 9
FIGURE 4: SUPPORT POLYGON FOR DOUBLE SUPPORT PHASE 11
FIGURE 5: INTERACTION OF THE SUPPORTING BASE WITH THE GROUND SURFACE 13
FIGURE 6: STABILITY MARGIN 14
FIGURE 7: INVERTED PENDULUM OF A LEGGED SYSTEM [7] 15
FIGURE 8: GRAVITY COMPENSATION INVERTED PENDULUM [14] 16
FIGURE 9: ONE INVERTED PENDULUM AND TWO QUASI-STATIC COUPLED PENDULUMS 16
FIGURE 10: GUROO DEGREES OF FREEDOM AND JOINT COORDINATE AXES 24
FIGURE 11: MODEL OF A TYPICAL LINK 25
FIGURE 12: POSITION OF REFERENCE COORDINATE FRAME 26
FIGURE 13: GUROO CONTROL ARCHITECTURE 27
FIGURE 14: 3D LINEAR INVERTED PENDULUM MODEL 29
FIGURE 15: FREE BODY DIAGRAM OF AN ARBITRARY LINK 32
FIGURE 16: SOFTWARE SEGREGATION 36
FIGURE 17: BALANCE SYSTEM BLOCK DIAGRAM 37
FIGURE 18: METHOD FOR DETERMINING THE SUPPORTING BASE 39
FIGURE 19: SIMPLIFIED MODEL FOR CALCULATING C.O.M. 41
FIGURE 20: CALC_COM() BLOCK DIAGRAM 42
FIGURE 21: SIMULATOR C.O.M MODEL BLOCK DIAGRAM 43
FIGURE 22: SIGN-BASED CONTROL LAW FOR BALANCE ATTITUDE 45
FIGURE 23: SEQUENCES OF IDEAL AND PREDICTED SUPPORT PHASES FOR THE DYNAMIC WALKING GAIT 47
FIGURE 24: 3D C.O.M. PLOT IN MATLAB 49
FIGURE 25: EVALUATION OF C.O.M. MODEL 50
FIGURE 26: C.O.M. LOCATIONS FOR WALKING GAIT 52
FIGURE 27: LOCATION OF THE ZMP FOR THE DYNAMIC WALKING GAIT 54
FIGURE 28: DESIRED JOINT VELOCITIES SPECIFIED BY THE ATTITUDE CONTROL SYSTEM 55
FIGURE 29: STABLE POSITION OF THE GUROO 56

List of Tables

TABLE 1: BALANCE CONTROL CRITERIA..11
TABLE 2: MDH PARAMETERS 26
TABLE 3: BALANCE SYSTEM SOFTWARE FUNCTIONS 37
TABLE 4: STANDARD DEVIATION OF C.O.M. COMPARISON 51

Chapter 1

Introduction

The purpose of this chapter is to give a brief introduction to the concept of Humanoid Robots and Active Balance Control. Motivation for the use of such robots is discussed and the concept of the RoboCup competition is introduced. Finally, descriptions of the outcomes and achievements for this thesis are given.

1.1 Humanoid Robots

Potential for the use of humanoid robots stems from the belief that machines of human-like dimensions are capable of easily adapting to human living environments. With the publishing of Rodney Brooks' behavioural-based approach to robot intelligence in 1985, the realisation of such useful mobile robots has significantly increased.

Bipedal walkers have significant advantages over conventional wheel-based robots because their mechanical design allows for better mobility. Dynamic control, and the ability to lift a support point off the ground gives a robot the ability to move over rough terrain and negotiate obstacles more easily than traditional statically stable walking machines. However, with the advantage of increased mobility comes the significant problem of stability problems - an issue that needs addressing through high levels of analysis and computation.

Currently, 49 major humanoid projects exist around the world [1], and many different approaches have been adopted in regards to robot design. The approaches vary from predominantly behavioural and interactive designs, to sophisticated platforms used for the development of joint control, trajectory analysis, vision systems and dynamic balance control.

The purpose of the University of Queensland "GuRoo Humanoid Project" is to build an anthropomorphic robot that will be competitive in the humanoid league division of the annual RoboCup Competition.

1.2 RoboCup Competition

The concept of an annual RoboCup Competition originated from an idea proposed in Japan at the "Workshop on Grand Challenges in Artificial Intelligence" in 1992. The first official RoboCup games and conference was held in 1997, and is now recognised as the world's largest mobile robot event [2].

The competition, based around the sport of soccer, aims to "promote robotics and AI research" by offering a "publicly appealing, but formidable challenge." The vision proposed by the competition organisers is:

> *"By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champions."* [2]

To realise this vision, a humanoid must first be developed to demonstrate reliable speed, strength, intelligence and balance control. The aim of the University of Queensland's GuRoo team is to develop a robot capable of competing in the solo division of RoboCup's humanoid competition.

In 2002, the competition involves three challenges:

1. Standing on One Leg
2. Humanoid Walk
3. Penalty Shot

1.3 Walking Algorithms and the Difficulty of Dynamic Balance Control

Biped walking algorithms can be distinguished as being either static or dynamic. The distinction is made depending on the location of the centre of mass during motion. For static walking, the centre of mass is always located above a polygon created by the external boundaries of the leg base (see section 2.3.1). The biped will remain statically stable if it is paused at any time during its motion.

Dynamic walking is generally much faster than static walking. In dynamic walking, inertial effects are considered, and it is possible for the centre of gravity to be outside the supporting base area. Human walking patterns are considered to be dynamic and exist as a series of “calculated falls” from one supporting foot to the next.

Traditionally, robots have maintained stability throughout their motion by maintaining at least three points of contact with the ground at all times. Since bipedal machines have only up to two points of contact with the ground, they must maintain stability through alternative means.

To gain insight into the methods of approaching balance control for a humanoid, the balance behaviour of human beings can be investigated. While walking or at standstill, humans attempt to remain stable by initially adjusting the distribution of pressure exerted by their feet. If the pressure application is not adequate for maintaining stability, the centre of gravity can be altered by performing a structural movement or by talking a step.

The balance behaviour of human beings is recognised as being the complex result of powerful and adaptive processes made in the brain (central processor), incorporating feedback from an extensive nervous system. The nervous system continually provides the brain with information from numerous sources through nerve impulses (messages). The information considered is provided from the eyes, inner ear, and the muscles of all body parts including the soles of the feet.

The design of a reliable and effective balance control system for a dynamically stable robot is an extremely difficult process. Designing a suitable model of the human balance system involves high levels of computation and control. Only few of the existing worldwide humanoid projects are able to adapt to adverse disturbances. Those that can, extensively rely on feedback systems from gyroscopic motion sensors and force-sensitive touch sensors.

1.4 Motivation

Motivation for the production of humanoid robots is a result of the recognised potential for their use in society. A robot of human-like dimensions would be capable of both interacting with human beings and also moving within human-based living environments.

Traditional robots exhibiting statically stable behaviours, such as wheel-based and frame-based machines, often have trouble traversing uneven terrain. Dynamically stable machines such as bipeds and humanoids, are much more versatile, demonstrating the ability to traverse uneven terrain through adaptive behavioural approaches.

The underlying goal in humanoid research is to develop a robot that is able to coexist with humans, while at the same time offering the ability to complete tasks that are classified as being either impossible or undesirable for humans to perform. Ideally, a humanoid robot offering the most additional value to society would offer a wide range of benefits, and not just specialised operations such as the vacuum cleaning robots that currently exist.

An obvious benefit for the development of humanoid robots is their usefulness in “search and rescue” and “decontamination” situations. Search and rescue robots have already been used in various major world events such as the World Trade Centre, Chernobyl, and the Three Mile Isle clean up. Further developments of such robots will inevitably reduce the requirements for fire-fighters, policemen and other emergency workers to risk their lives.

1.5 Project Outcomes and Achievements

The main goal associated with this thesis was the design and implementation of an active balance control system that would enable the GuRoo to remain upright despite adverse disturbances. Such disturbances included external forces and uneven terrain.

The scope of the thesis involved the initial design of software and inertial systems, comprehensive testing through simulation, design implementation and results from real hardware. With limited resources and time available, it became evident that the scope was largely overestimated.

A successful design was developed, simulated and evaluated. The proposed design forms a solid platform that will form the basis of further testing, development and implementation on real hardware.

1.6 Chapter outline

The remainder of this document is divided into 7 chapters. A description of the contents of each chapter is as follows:

Chapter 2 - Introduces relevant information within the field of study. The GuRoo is introduced, as are other humanoid projects around the world. Key concepts regarding Balance Control are defined and stability criteria specified.

Chapter 3 - Describes the specifications involved with the thesis project. It illustrates how the project design was broken into stages, and describes the way in which specifications were derived.

Chapter 4 - Gives a description of the mechanical design of the GuRoo. The assumptions proposed for modelling the system are described, and the feedback from the control architecture is illustrated.

Chapter 5 - Discusses the mathematical models and concepts used in the design of the active balance system.

Chapter 6 - Describes how the proposed balance control system is implemented in software.

Chapter 7 - Illustrate the success of the project with respect to the criteria outlined in Chapter 3. The criteria used for the evaluation of each model are specified and the results discussed accordingly.

Chapter 8 - Reflects on the goals of this thesis and the extent to which they were achieved. Suggestions for future developments of the balance control system are also described.

Chapter 2

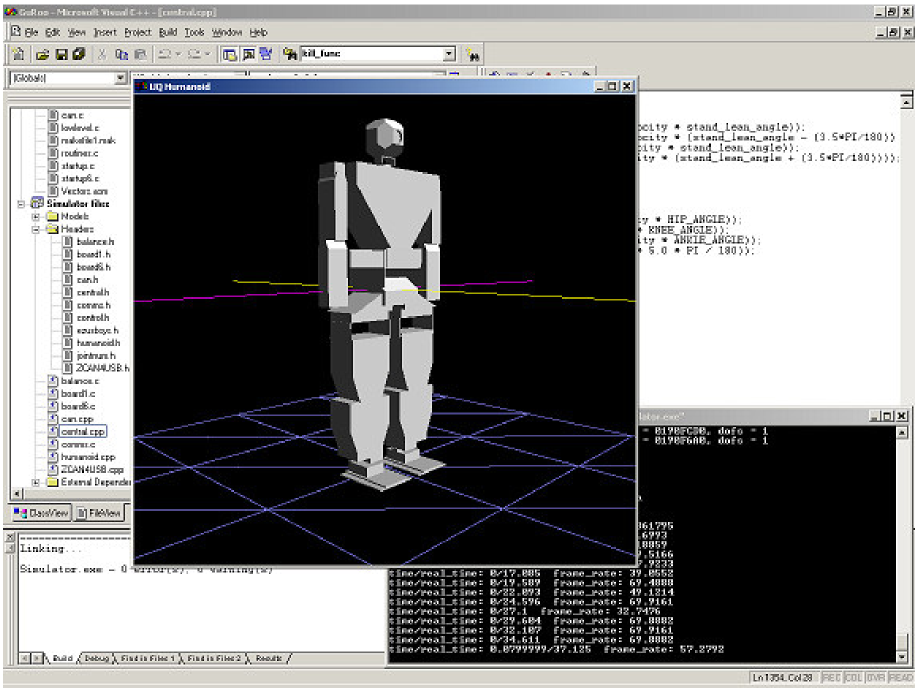
Literature Review

This purpose of this chapter is to introduce relevant information within the field of study. It introduces the University of Queensland's GuRoo project, and then briefly describes other humanoid projects around the world. Key concepts regarding Balance Control are defined and stability criteria specified.

2.1 Previous Work (GuRoo)

The University of Queensland's GuRoo humanoid project is currently in its second year of development. In 2001, a mechanical design based on human-like proportions was developed. The individual design and implementation of power, vision and joint controller systems was also initiated. The DynaMechs Simulator, developed by McMillan [3] was adapted and successfully used to simulate the distributed control structure of the GuRoo. Smith [4] successfully demonstrated various structural movements such a $0.03 ms^{-1}$ static walking gait (Figure 1).

In 2002, work began with all robot hardware being constructed and debugged. Development and implementation of high- and low-level software has been demonstrated through successful control of the real robot (Figure 2).



```
ocity * stand_lean_angle));
ocity * (stand_lean_angle - (3.5*PI/180))
city * stand_lean_angle));
ity * (stand_lean_angle + (3.5*PI/180))));

ty * HIP_ANGLE));
* KNEE_ANGLE));
ity * ANKLE_ANGLE));
* 5.0 * PI / 180));
```

```
---------------------
Linking...

Simulator.exe - 0 error(s), 0 warning(s)
```

```
= 8198FC88, dofs = 1
= 8198F6A8, dofs = 1

061795
.6993
18859
.5166
.7233
time/real_time: 0/17.885  frame_rate: 39.8552
time/real_time: 0/19.589  frame_rate: 69.4888
time/real_time: 0/22.093  frame_rate: 49.1214
time/real_time: 0/24.596  frame_rate: 69.9161
time/real_time: 0/27.1  frame_rate: 32.7476
time/real_time: 0/29.604  frame_rate: 69.8882
time/real_time: 0/32.107  frame_rate: 69.9161
time/real_time: 0/34.611  frame_rate: 69.8882
time/real_time: 0.0799999/37.125  frame_rate: 57.2792
```

Figure 1: GuRoo Simulator

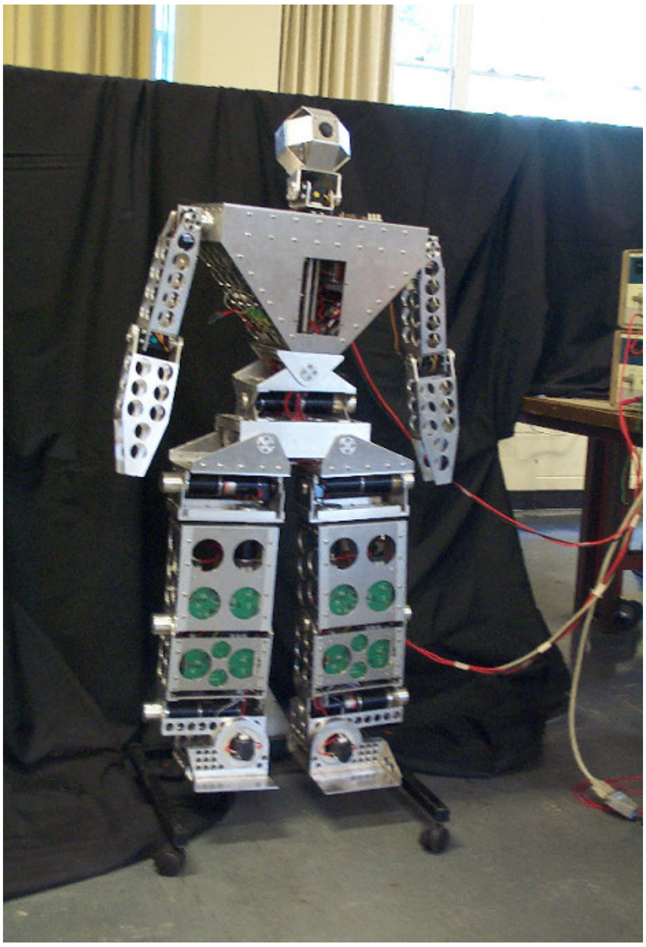


Figure 2: GuRoo Hardware

2.2 Other Humanoids in General

Humanoid development is a relatively new field in robotics research and few results are publicly recognised. The most publicised humanoid robots are Honda's "ASIMO" and Sony's "SDR-4X". These robots illustrate the forefront in humanoid technology, particularly in the areas of active balance control.

Honda's ASIMO robot (Figure 3a) [5] is 120cm in height and was originally conceived to function in an actual human living environment. It has the ability to walk continuously and smoothly while changing direction, and can travel at up to 0.44m/s. By predicting it's next movement in real time, ASIMO shifts its centre of gravity in anticipation of its path. For balance control, ASIMO uses gyroscopic and accelerative sensors in the torso, as well as 6-axis foot area sensors.

Sony's "SDR-4X" (Figure 3b) [6] is a small biped robot measuring 50cm in height. It is able to walk at 0.33m/s as well as demonstrate basic movements such as walking and changing direction, standing up, balancing on one leg, kicking a ball and dancing. Its movement allows it to walk on irregular surfaces and in the presence of external forces. For feedback regarding posture and position control it uses acceleration sensors in the torso, and four power sensors on each foot.

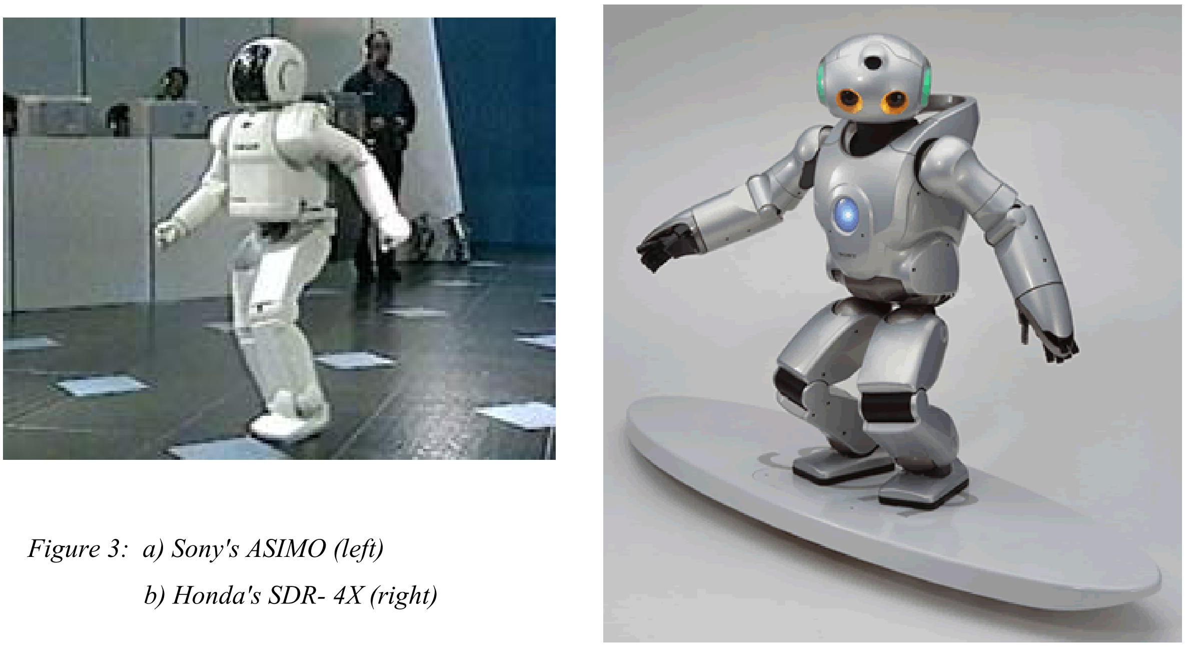


Figure 3: a) Sony's ASIMO (left)
b) Honda's SDR- 4X (right)

2.3 Description of Existing Humanoid Balance Systems

Humanoid robot design and bipedal locomotive control are currently two of the most challenging fields in robotic research. While some of the currently existing humanoid projects exhibit very reliable dynamic biped walking (such as Honda's ASIMO and Sony's SDR-4X), none are capable of exhibiting adequate responses to sudden interactions with unknown disturbances, or to unexpected decisions such as emergency "stop" or "avoid" [7].

Previous works in motion generation of humanoids can be classified into two categories [7]:

Trajectory replaying – A joint-motion trajectory is prepared in advance, and applied to the real robot with little on-line modification. The problem is divided into two sub-problems: planning and control.

Real-time Generation – Joint-motions are generated in real-time. The present state of the system is fed back to the controller and considered with the pre-provided goal of the motion. Planning and control and executed in a unified manner.

In pursuing the development of a highly-mobile robot, the technique of real-time motion generation offers the greatest amount of potential. However, due to the large amounts of computation required, adequate responses to sudden disturbances such as those mentioned are not yet realizable.

2.3.1 Defining Key Concepts

In considering stability conditions and the control of balance in biped locomotion, several dynamic-based criteria have been defined. The criteria most commonly used are the Centre of Pressure (C.o.P.), Zero Moment Point (Z.M.P.), Foot

Rotation Indicator (F.R.I.), and the Ground Projection of the Centre of Mass (G.C.o.M.). These terms are defined in Table 1.

Term	**Definition**
Centre of Pressure (C.o.P.)	The point on the foot / ground contact surface where the net ground reaction force actually acts.
Zero Moment Point (Z.M.P.)	The point on the foot / ground surface where the total forces and moments acting on the robot are zero.
Foot Rotation Indicator (F.R.I.)	The point on the foot / ground contact surface, within or outside the support polygon, where the net ground reaction force would have to act to keep the supporting base of the robot stationary [8]
Ground Projection of the Centre of Mass (G.C.o.M.)	The point on the foot / ground contact surface directly below (in the direction of gravitational acceleration) the location of the centre of mass of the robot.

Table 1: Balance Control Criteria

If ideal conditions are considered whereby neither the foot nor ground can deform under load, the C.o.P. and Z.M.P. locations will always coincide.

The *support polygon* of the robot is defined as the area of physical interaction between the robot and the ground surface. During single-support phases, it is the area of the supporting foot, and during double-support phases it is defined as the polygon created by the boundary of the two feet. The case of a double-support phase is shown in Figure 4.

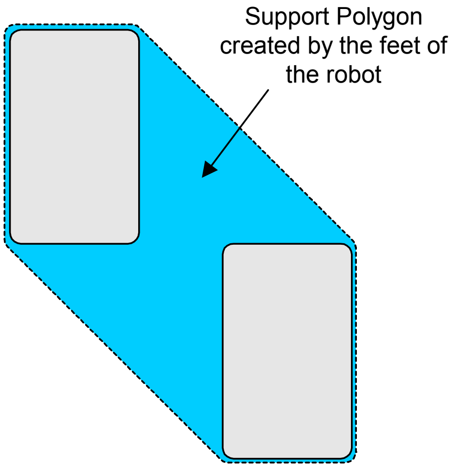


Figure 4: Support Polygon for Double Support Phase

2.3.2 Defining Stability Conditions

The concepts defined in section 2.3.1, are used to develop a general set of conditions, which can be used to describe the stability of an arbitrary biped robot.

When a robot is in a statically stable state, there are no inertial forces present. During such phases, the CoP, ZMP, FRI and GCoM points are all located within the support polygon and have coincident locations.

If the state of the robot is considered static, but is in an unstable state, the FRI and GCoM points are coincident and located outside the supporting polygon of the robot. During these states, the CoP is located at the boundary of the supporting base. By definition (Table 1), the physical constraints associated with the supporting base prevent the CoP from leaving the boundary at any time.

In dynamic situations where the motion of the robot needs to be considered, the GCoM is non-coincident with the FRI or ZMP. This is a direct result of the inertial forces associated with the link masses. If the FRI point is situated within the support polygon, it is coincident with the CoP and ZMP, and the robot exhibits postural balance. If at any time during motion, the FRI and ZMP are located outside the boundary of the support polygon, the robot will be dynamically unstable. The reason for this principle is derived from Figure 5, which shows a 2-dimentional view of the foot in the *x-z* plane.

The moment experienced by all points on the ground surface is linearly proportional to the distance of the point from the ZMP (or FRI). The arrow lengths in Figure 5 indicate the magnitudes of the moments. By considering moments taken about point *b*, a point on the boundary of the supporting base, the stability of the robot can be visualised.

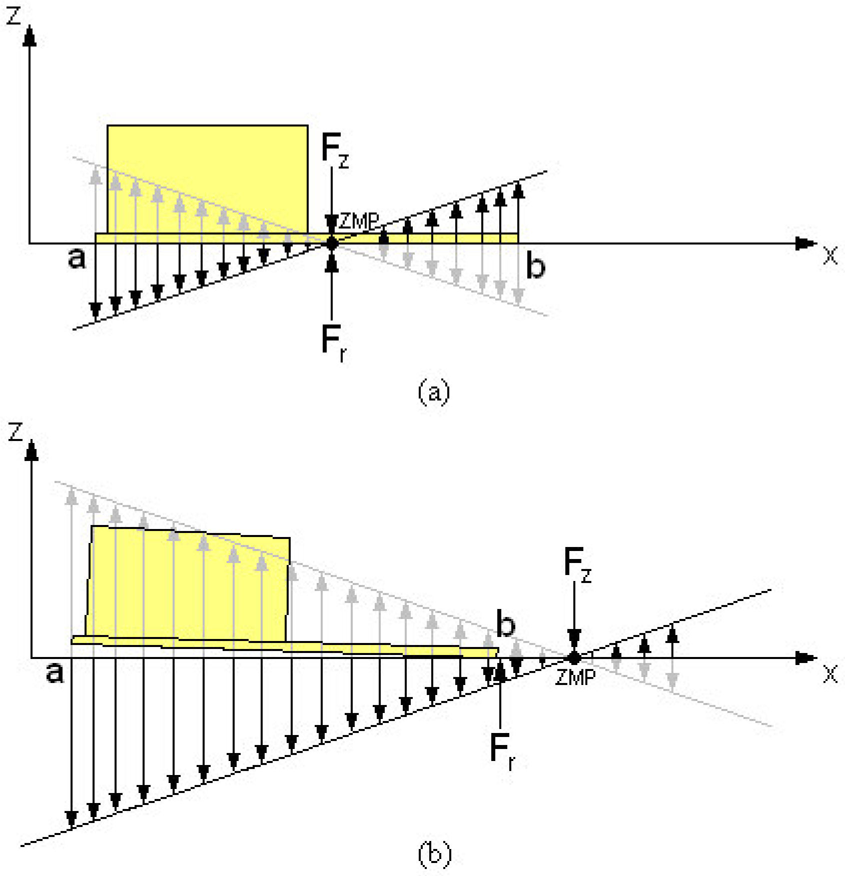


Figure 5: Interaction of the Supporting Base with the Ground Surface

In Figure 5a, the reaction force (F_r), counteracts the resultant moment due to the rotation of the robot, and the robot is dynamically stable. If the robot rotates about a point outside the support polygon (as in Figure 5b), the reaction force cannot compensate for the resultant moment about the ZMP and the supporting base will be either pushed into the ground or lifted from it. The result is an unstable pose.

The *Stability Margin* is a measure of the quality of stability, and is defined as the minimum distance between the location of the ZMP (or FRI) and the boundary of the support polygon [9]. If the stability margin is *high*, the robot is dubbed to have high stability. The concept of stability margin is shown in Figure 6.

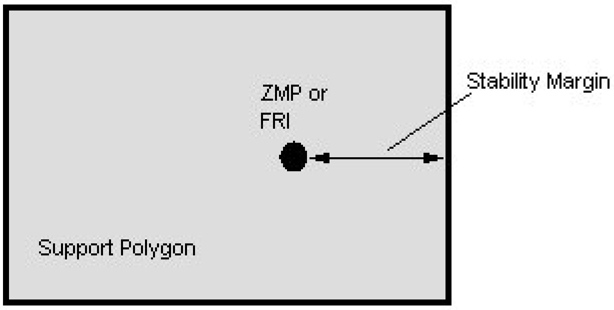


Figure 6: Stability Margin

2.3.3 Review of Methods of Balance Control

Many researchers have worked on various methods for generating motion trajectories and stabilizing systems for biped walking robots. Most of the existing systems aim to maintain stability by considering the relative position of the ZMP with the support polygon, and by defining compensating motions for the robot upper body.

Park et al. [10] presented a ZMP trajectory control scheme which was determined using fuzzy logic on the leg trajectories. The trunk and swing leg motions were compensated to stabilize the locomotion.

Fukuda et al. [11] used touch sensors on the feet of the robot to obtain the actual ZMP trajectory. The joint motion was then determined using recurrent neural networks with the constraint that the ZMP could move out of the support polygon.

A common model used in existing balance control systems is that involved with manipulating the motion of an *inverted pendulum.*

2.3.4 Inverted Pendulum Modes

Kajita et al. [12] introduced the Three-Dimensional Linear Inverted Pendulum Mode (3D-LIPM) which is a simplified model for a biped robot. The model was derived from a general three-dimensional inverted pendulum whose motion was constrained to move along an arbitrarily defined plane [12]. The model allows for the separate controller design of sagittal (*x-z*) and lateral (*y-z*) motion, significantly simplifying the analysis of dynamic motion. The inverted pendulum model consists of a point mass and a massless telescopic leg extending from the centre of the supporting base to the location of the centre of mass of the robot (Figure 7).

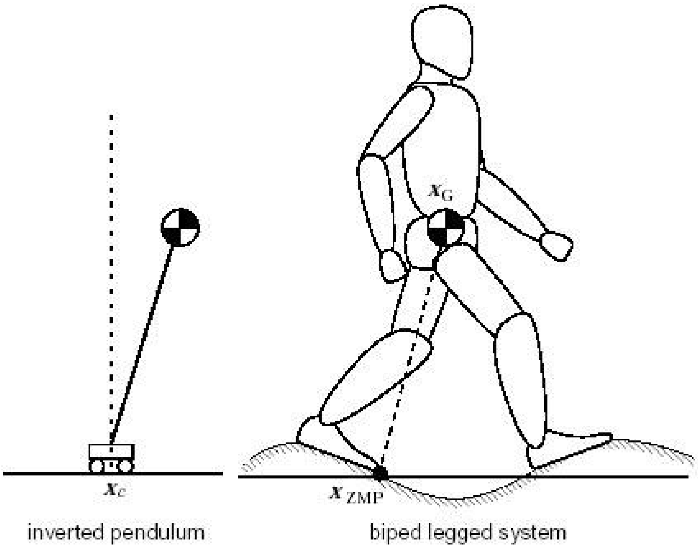


Figure 7: Inverted Pendulum of a Legged System [7]

For a humanoid robot, Li et al. [13] considered the model to be composed of two separate moving masses – one representing the torso and the other representing the legs. Motion generation specific to the surface structure of the ground was predetermined, while the motion of the torso was adapted in real-time to ensure postural stability. For traditional biped robots, the absence of a torso allows the mass distribution to be modelled as a single point mass. Stability is then maintained by relying on the overall motion of both the supporting leg and the swinging leg. The inverted pendulum model proposed by Kajita et al. [12], consisted only of a concentrated mass at the torso and neglects the inertial effects of the legs.

An extension to the basic 3D-LIPM was developed by Jong H. Park et al. [14], who developed a model called the "Gravity Compensated Inverted Pendulum Mode (GCIPM)". The model included the predetermined effects of the dynamics of the free (swing) leg on the ZMP by modelling the robot as two separate inverted pendulums. The concept is shown in Figure 8 [14]. The model assumes that the swing leg consists of mass concentrated at the location of the foot, and that its dynamics are dominated by gravitational acceleration. In this sense, only the static effect of the swinging leg is considered.

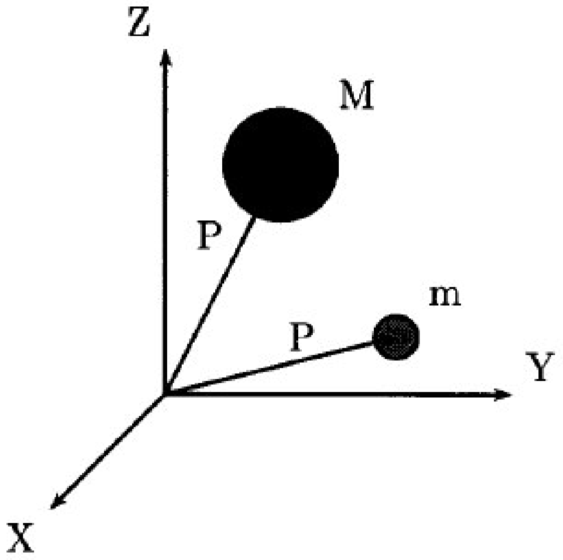


Figure 8: Gravity Compensation Inverted Pendulum [14]

Caballero et al. [15] developed a further extension that was applicable to humanoid robots. The model represents the robot as one inverted pendulum and two quasi-static coupled pendulums. Using ZMP stability theory, this model was successfully used to generate stable geometric gaits. The sagittal plane model of the pendulums is shown in Figure 9 [15].

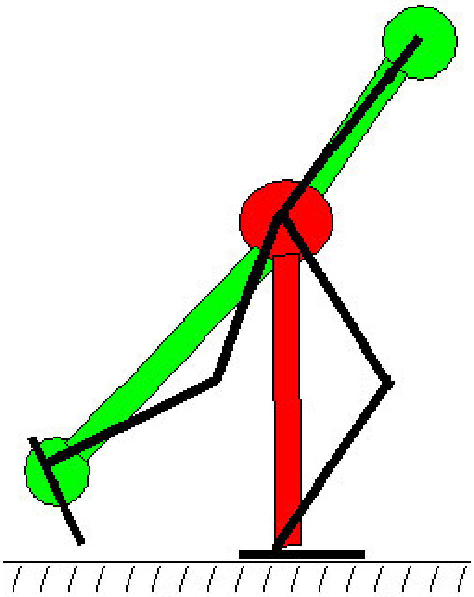


Figure 9: One Inverted Pendulum and Two Quasi-Static Coupled Pendulums

A. Albert and W. Gerth [16], then proposed two models which considered the dynamic effects of the swinging leg. The models are as follows:

Two Masses Inverted Pendulum Mode (TMIPM) – Robot consists of two masses – one mass representing the torso, and one mass representing the swinging leg. The complete dynamic effect of the swinging leg is considered for the generation of the torso motion.

Multiple Masses Inverted Pendulum Mode (MMIPM) – Robot consists of many masses - one mass representing the torso, and an arbitrary number of masses modelling the swinging leg. The foot motion of the swinging leg is pre-defined, and all other trajectories are calculated iteratively.

Chapter 3

Specifications

This chapter describes the specifications involved with the thesis project. It illustrates how the project design was broken into stages, and describes the way in which specifications were derived.

3.1 Introduction

The project objectives can be divided into two stages:

1. Preparation of the GuRoo for the Solo Competition in the Humanoid Division of RoboCup 2002
2. Further development of an Active Balance Control System

Only the development of the Balance Control System is documented in the remaining chapters of this thesis.

3.2 Preparation for RoboCup

The first primary objective for the entire GuRoo team was to adequately develop the robot for competition in RoboCup 2002.

The Solo Competition of the Humanoid Division involved three challenges [2]:

1. *Standing on One Leg* - Robot shall remain stationary and stable for 1 minute while standing with one leg raised.

2. *Humanoid Walk* – Robot is to walk 6m in a straight line, around a red marker, and then back to the starting line.
3. *Penalty Shot* – Robot is to walk 1.8m and then kick a ball 3.6m into a 2.4m wide goal

The ability to successfully accomplish all of these challenges relied on the adequate completion of all humanoid subsystems including the development of a walking gait, balance control, vision system, joint control, and hardware design and implementation.

The preliminary work required in forming a basic platform for the development of higher order systems involved:

- Modifications of the Simulator to reflect new hardware proposals
- Restructuring of the Simulator
- Assembly and Debugging of Hardware
- Debugging of Software
- Generating various Motion Trajectories for the GuRoo

The success of the work completed in this stage can be evaluated by assessing the performance of the GuRoo in the RoboCup Competition. All work required completion by the date of departure on 14th June 2002.

3.3 Development of an Active Balance Control System

With the tasks of section 3.1 completed, a basic platform suitable for the development of higher order systems such as Active Balance Control was created.

The development of the Balance System was broken into the following stages:

- Research Current Balance Control Methodology
- Develop and Test a method for determining the Supporting Base of the Robot
- Develop and Test a Centre of Mass Model

- Develop and Test an Attitude Control Algorithm

The testing phases involved two stages:

1. Simulator implementation and testing
2. Real Hardware implementation and testing

3.3.1 Research of Balance Control Methodology

Outcomes

- Complete Understanding of GuRoo Software
- Familiarity with current humanoid balance control technology

3.3.2 Supporting Base Model

Outcomes

- Determine the points of contact of the robot with the ground at all times

Depends on

- Feedback of Joint Positions
- Foot Pressure (force-sensitive) Sensors

Evaluated by

- Comparison of model output with ideal results predicted from trajectory analysis

Desired Results

- Model accurately reflects the interaction of the robot with the walking surface for unspecified movements
- Since this system is crucial to the overall success of the balance system, correct results regarding the support state are desired for ~100% of the time

3.3.3 Centre of Mass Model

Outcomes

- Determine the coordinates of the location for the centre of mass of the robot

Depends on

- Supporting Base Model
- Accuracy of Hardware Model
- Feedback of Joint Positions

Evaluated by

- Comparison of model with approximated results obtained statically from solid edge
- Comparison of model with results obtained directly from simulator

Desired Results

- Model accurately determines the location for the centre of mass of the robot.
- The required accuracy of this model is 90%

3.3.4 Attitude Control

Outcomes

- Method for defining structural movements that maintain stability

Depends on

- Supporting Base Model
- Centre of Mass Model
- Gyroscopes / Accelerometers
- Foot Pressure (force-sensitive) sensors
- Joint Control
- Vision System
- Processing Speed of central controller

Evaluated by

- The extent to which the robot can resists adverse disturbances
- The smoothness of joint motion
- Minimum load torques at each of the joints

Desired Results

- Robot can perform simple movements, resist external forces and compensate for inaccuracies such as backlash in the gearboxes
- Robot can demonstrate smoothness and stability while walking

The measure of success for the Attitude Control system is not quantifiable.

The feasibility of having an active balance control system implemented in hardware relies on the accuracy of the developed models and is also dependent on work of other team members. The adequacy of other humanoid subsystems such as joint control will be essential.

Chapter 4

Analysing the Hardware

The purpose of this chapter is to give a description of the mechanical design of the GuRoo. The assumptions proposed for modelling the system are described, and the feedback from the control architecture is illustrated.

4.1 Mechanical Design

The mechanical design of the GuRoo consists of 24 rigid body links and 23 actuated revolute joints. The frame of the robot is made from machined aluminium and weighs 38kg in total. It consists of a centre body (torso), connecting a head and neck to four separate limbs - two legs and two arms. The distribution of the degrees of freedom (DOF) throughout the robot is given in Figure 10. A detailed description of all link parameters is given in Appendix A.

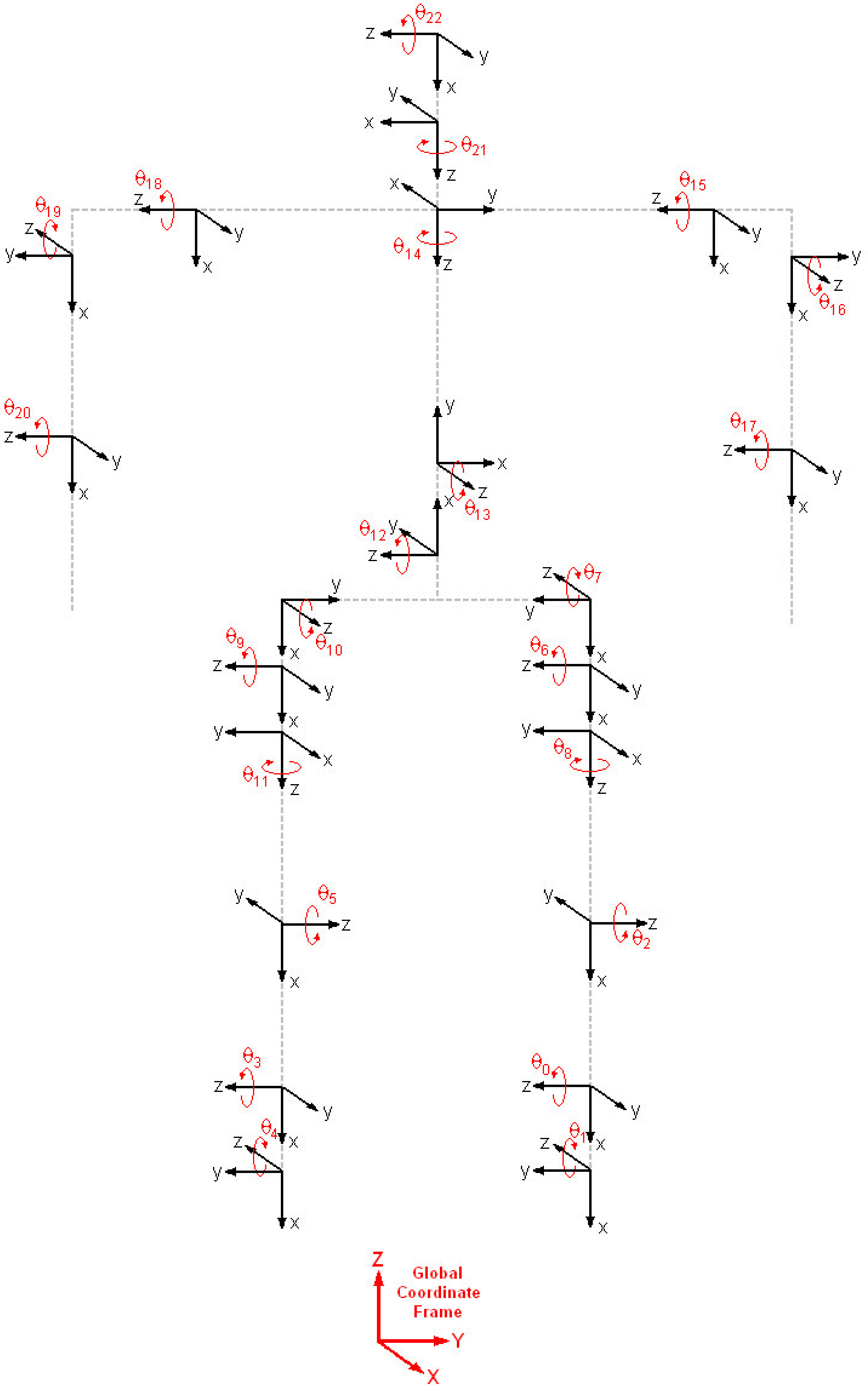


Figure 10: GuRoo Degrees of Freedom and Joint Coordinate Axes

In analysing the mechanical structure of the robot, each rigid body link needs to be considered separately. By doing this, all inertial forces can be considered and their dynamic effects on the motion of the robot adequately modelled.

A typical link is shown in Figure 11. In the analysis that follows, the distribution of mass of the link is considered to be solely concentrated at the location of the centre of mass. With this approximation, a link is defined as a rigid connection between coordinate frames and a point mass located somewhere between these frames. The reference frame for each link is located at the centre of the proximal joint.

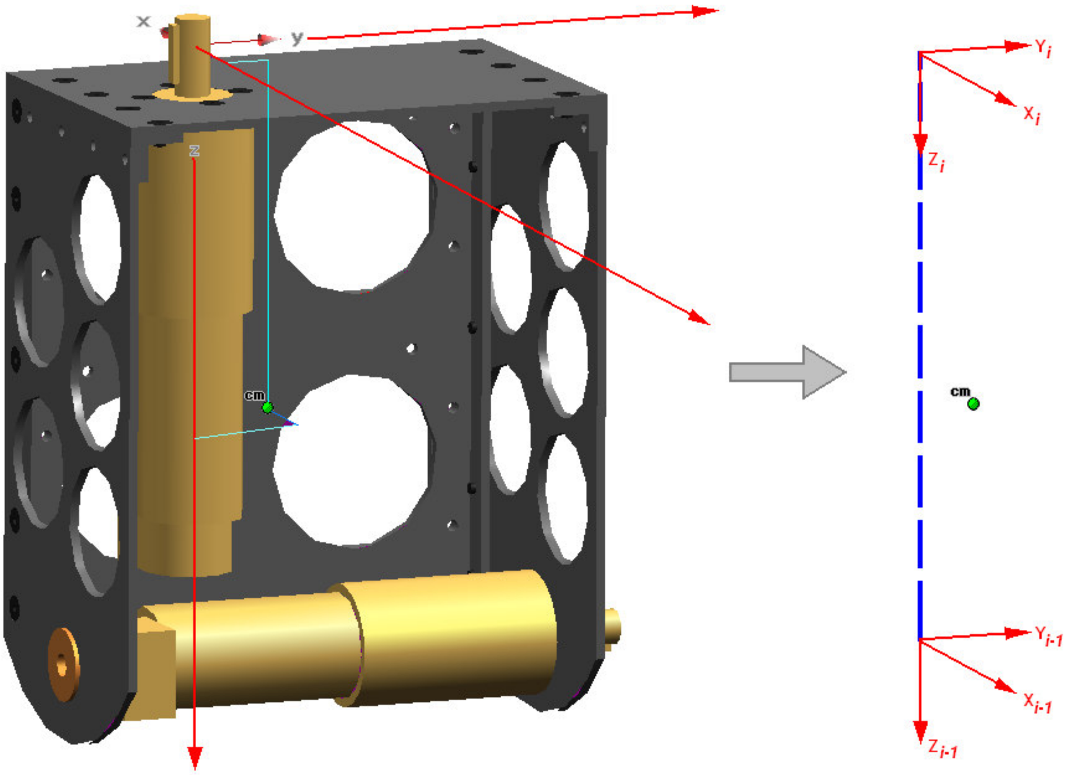


Figure 11: Model of a Typical Link

The base referential coordinate frame of the robot is located at the centre of the robots supporting base, and changes as the state of the robot changes. For single foot support phases, the reference frame is located at the centre of the base of the grounded foot. For double foot support phases, the reference frame is located at the centre of the polygon formed by the boundaries of the feet. The position and location of the reference coordinate frame is shown in Figure 12.

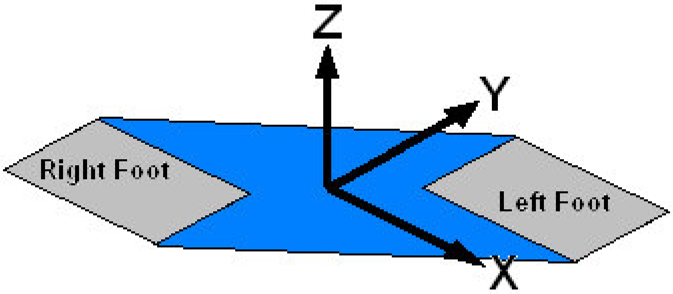


Figure 12: Position of Reference Coordinate Frame

Maxon RE32 series dc motors with 156:1 planetary gear heads are used for actuation of the lower limb and torso joints. The upper limb, neck and head joints are actuated using Hi-Tech HS705-MG RC servo motors.

Each link, defined as a transformation between coordinate frames, is described using either a combination of rotational and translational transformations, or by the *Modified Denavit-Hartenberg (mdh)* parameters. The *mdh*-parameters are defined in Table 2 [17]:

Description	mdh Parameter	Definition
Link Length	a_i	The offset distance between the z_{i-1} and z_i axes along the x_i axis
Link Twist	α_i	The angle from the z_{i-1} axis to the z_i axis about the x_i axis
Link Offset	d_i	The distance from the origin of frame $i-1$ to the x_i axis along the z_{i-1} axis
Joint Angle	θ_i	The angle between the x_{i-1} and x_i axis about the z_{i-1} axis

Table 2: mdh Parameters

The *mdh*-transformation matrix is then given by:

$$
{}^{i-1}A_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

4.2 Distributed Control Network

The control architecture of the GuRoo is shown in Figure 13. It consists of six distributed joint controller boards and either an iPAQ or Laptop. The central controller on the iPAQ generates desired joint velocities for each of the actuators in real time, and sends these via a 50Hz serial link to servo controller board. The servo board then distributes the messages among the five DC motor boards via a Controller Area Network (CAN) bus.

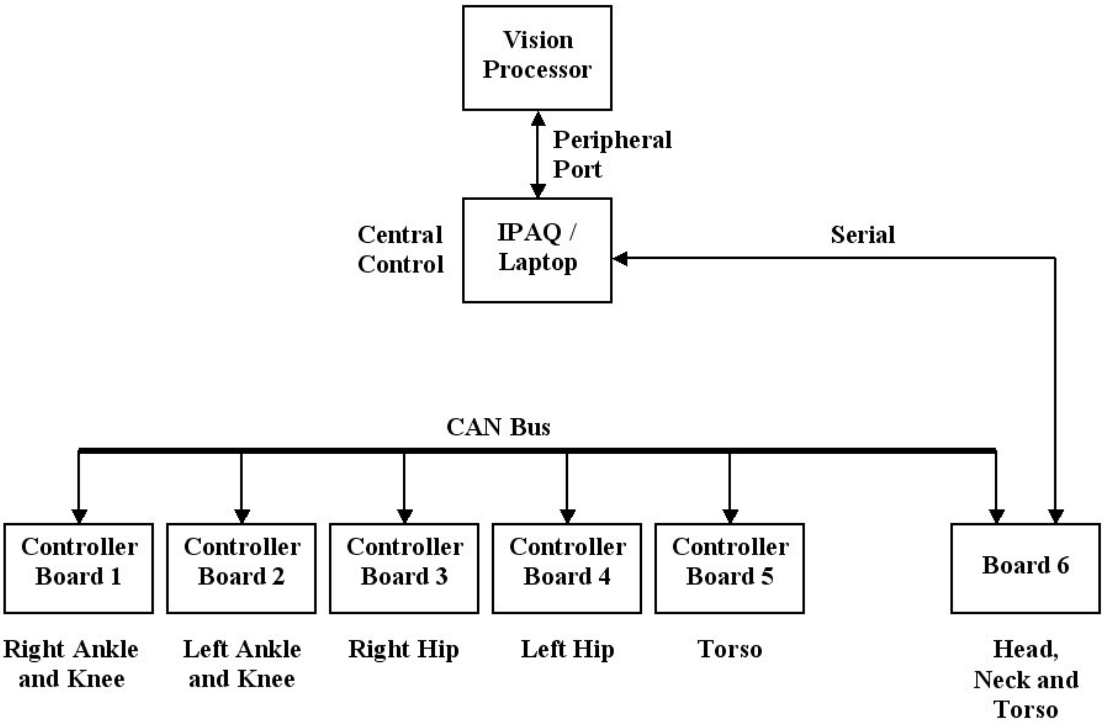


Figure 13: GuRoo Control Architecture

The only information available regarding the current state of the robot is the actual position of each actuated joint. On each board, this is achieved using two external quadrature decoders and an internal quadrature decoder on the TMS320F243 Digital Signal Processor (DSP) to read the 500 count per revolution optical encoders coupled to each motor. The actual position data is then fed back to the central controller via the serial link. The data is in the units of encoder counts per control loop, and needs to be converted to radians by first integrating the velocity and then applying the `ENC2RAD()` function [18].

Chapter 5

Mathematical Models

This chapter discusses the mathematical models and concepts used in the design of the active balance system. Specifically, the concepts derived are:

- The Linear Inverted Pendulum Model
- Zero Moment Point

5.1 Linear Inverted Pendulum

The 3D Linear Inverted Pendulum model is shown in Figure 14.

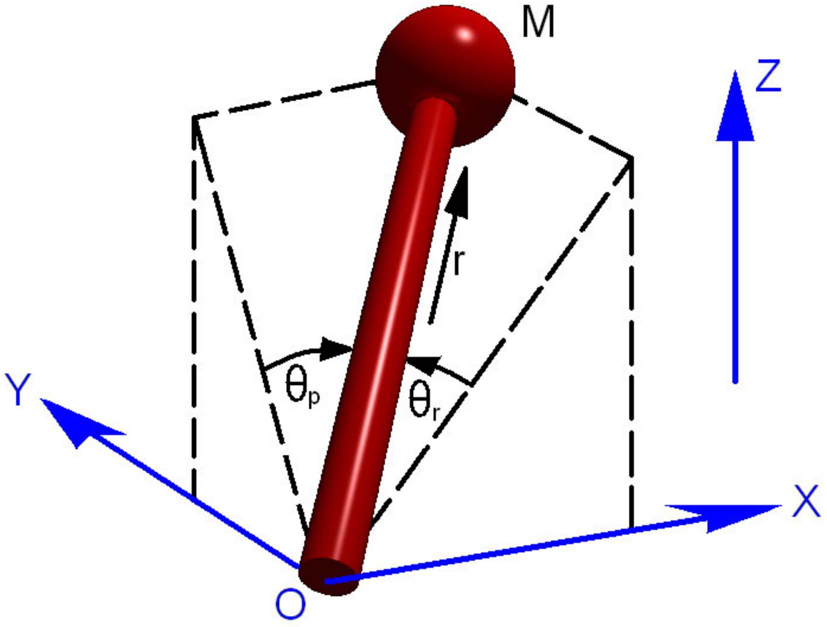


Figure 14: 3D Linear Inverted Pendulum Model

The following discussion is based on the research performed by Kajita et al. [12]. The position of the point mass, *M*, is specified by a set of state variables, $(\theta_{roll}, \theta_{pitch}, r)$ related to the Cartesian coordinates by:

$$x = r \sin \theta_{pitch}$$

$$y = -r \sin \theta_{roll}$$

$$z = r\sqrt{1 - (\sin \theta_{roll})^2 - (\sin \theta_{pitch})^2}$$

If τ_{roll}, τ_{pitch} and f are the actuator torque and force associated with these state variables, then the equation of motion is given by:

$$m\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \left(J^T\right)^{-1}\begin{pmatrix} \tau_{roll} \\ \tau_{pitch} \\ f \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix}$$

In the above equation, the Jacobian J is given by:

$$J = \frac{\partial p}{\partial q} = \begin{pmatrix} 0 & r\cos\theta_{pitch} & \sin\theta_{pitch} \\ -r\cos\theta_{roll} & 0 & -\sin\theta_{roll} \\ -r\cos\theta_{roll}\sin\theta_{roll}/D & -r\cos\theta_{pitch}\sin\theta_{pitch}/D & D \end{pmatrix}$$

The following equations can then be derived for the dynamics of the pendulum:

$$m(-z\ddot{y} + y\ddot{z}) = \frac{D}{\cos\theta_{roll}}\tau_{roll} - mgy$$

$$m(z\ddot{x} - x\ddot{z}) = \frac{D}{\cos\theta_{pitch}}\tau_{pitch} + mgx$$

If the motion of the pendulum is limited to motion in the horizontal plane, the following equations for the motion can be derived,

$$\ddot{y} = \frac{g}{z} y$$

$$\ddot{x} = \frac{g}{z} x$$

5.2 Zero Moment Point

The concept of the Zero Moment Point (ZMP) was introduced in section 2.3.1. For convenience, its definition is repeated here:

> *"The Zero Moment Point (ZMP) is the point on the ground surface about which the sum of all the moments of active forces is equal to zero."* [9]

Mathematically, it can be defined as:

$$ZMP = \frac{\sum_i M_i}{\sum_i F_i} \quad [19]$$

where,

$\sum_i M_i$ represents the resultant moment about the ZMP

$\sum_i F_i$ represents the resultant force exerted at the foot from the ground

Given a state vector defining a set of joint angles for the GuRoo, the location of the ZMP can be computed using inverse dynamics. Link transformation matrices can be used to determine the position and orientation of each link with respect to the global coordinate frame.

The robot is treated as a general *i*-segment extended rigid-body kinematic chain. For dynamic situations, the external forces acting on each link of the robot are the gravitational, inertial and reaction forces. A free body diagram illustrating the forces and moments acting on an arbitrary link is shown in Figure 15. Point ***P*** is an arbitrary point on the ground surface.

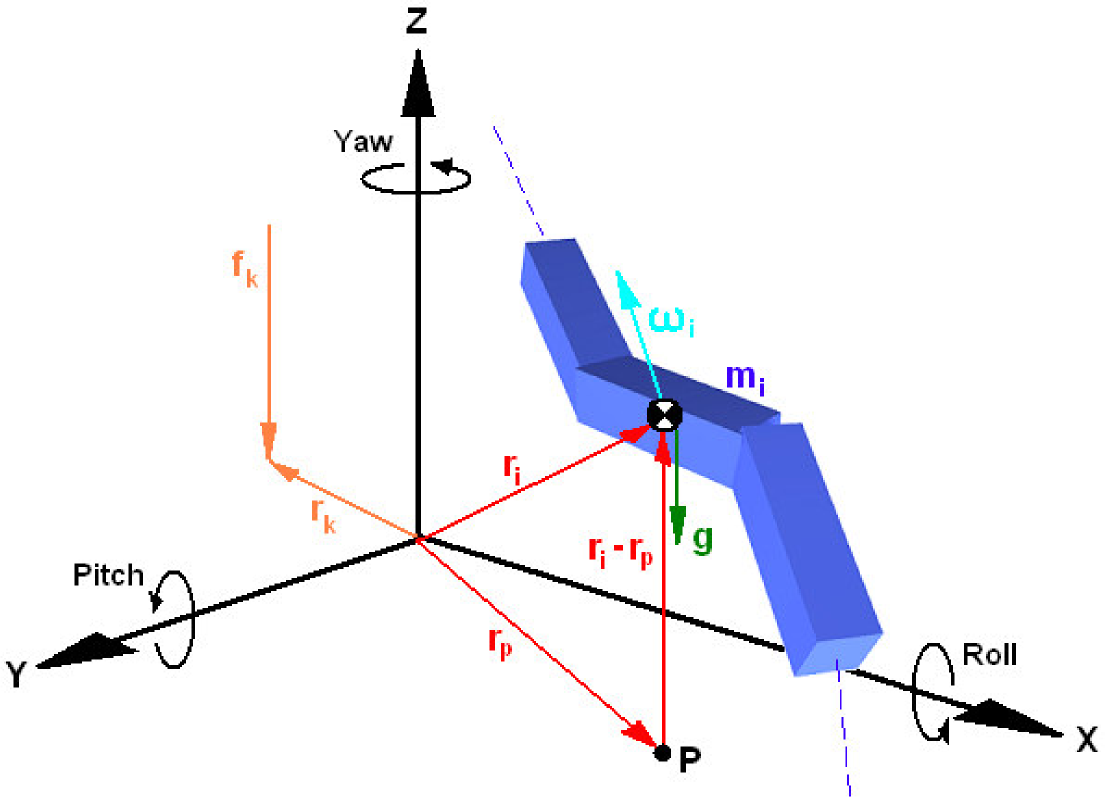


Figure 15: Free Body Diagram of an Arbitrary Link

According to D'Alembert's principle the total of all forces and moments acting on the robot is zero. Using this principle, and recognising that each body segment contributes to the net moment at the foot, all possible moment contributions about point ***P*** are considered.

The moment contribution (M_c) due to the acceleration of ***P*** in the reference coordinate frame is given by:

$$M_c = \frac{1}{m}\sum_{i=1}^{n}\left(m_i\left(r_i - r_p\right)\times m\ddot{r}_p\right) = \sum_{i=1}^{n}\left(\left(r_i - r_p\right)\times m_i\ddot{r}_p\right)$$

From the application of simple dynamics and vector addition, the total ground reaction force (F_r) is given by the combination of inertial and gravitational effects:

$$F_r = \sum_{i=1}^{n}\left(F_i + m_i\left(g - \ddot{r}_p\right)\right) = \sum_{i=1}^{n} m_i\left(\ddot{r}_i + g - \ddot{r}_p\right)$$

The action of this force causes a ground reaction moment (M_r) about point ***P*** given by:

$$M_r = \sum_{i=1}^{n} \left(r_i - r_p\right) \times m_i \left(\ddot{r}_i + g - \ddot{r}_p\right)$$

The total inertial moment (M_e) of the link masses due to rotational motion is:

$$M_e = \sum_{i=1}^{n} \left(M_i\right) = \sum_{i=1}^{n} \left[I_i \cdot \dot{\omega}_i + \omega_i \times I_i \cdot \omega_i\right]$$

The moment contribution (M_{ext}) due to the action of external forces (f_k) is given by:

$$M_{ext} = -\sum_j M_j - \sum_k \left(r_k - r_p\right) \times f_k$$

The total moment (M_P) about point ***P***, given by the sum of these moment contributions, is then:

$$M_P = M_c + M_r + M_e + M_{ext}$$

An assumption is made that the coefficient of static friction between the foot and the ground surface is sufficiently large so that the foot of the robot does not slide. The acceleration vector $\ddot{r}_p$ will then be zero, and since no additional external forces are present, the resultant moments M_{ext} and M_P will be zero.

The total moment about point ***P*** then becomes:

$$M_P = M_r + M_e = \sum_{i=1}^{n} \left(r_i - r_p\right) \times m_i \left(\ddot{r}_i + g\right) + \sum_{i=1}^{n} \left[I_i \cdot \dot{\omega}_i + \omega_i \times I_i \cdot \omega_i\right]$$

At the location of the zero moment point, the resultant moment, M_{ZMP}, will be:

$$M_{ZMP} = \begin{bmatrix} 0 \\ 0 \\ M_z \end{bmatrix}$$

Since the ZMP is restricted to be on the walking surface (ground plane), the z component will be zero and it's location is given by:

$$r_{ZMP} = \begin{bmatrix} x_{ZMP} \\ y_{ZMP} \\ z_{ZMP} \end{bmatrix} = \begin{bmatrix} x_{ZMP} \\ y_{ZMP} \\ 0 \end{bmatrix}$$

The above equation for moments about ***P*** can then be solved to give the location of the zero moment point with respect to the reference coordinate frame O:

$$x_{ZMP} = \frac{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)x_i - \sum_{i=1}^{n} m_i \ddot{x}_i z_i - \sum_{i=1}^{n} (M_y)_i}{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)}$$

$$y_{ZMP} = \frac{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)y_i - \sum_{i=1}^{n} m_i \ddot{y}_i z_i - \sum_{i=1}^{n} (M_x)_i}{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)}$$

By rearranging these equations, the vertical ground reaction force F_{rz} can be isolated:

$$x_{ZMP} = \frac{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)x_i}{F_{rz}} - \frac{\sum_{i=1}^{n} m_i \ddot{x}_i z_i}{F_{rz}} - \frac{\sum_{i=1}^{n} (M_y)_i}{F_{rz}}$$

$$y_{ZMP} = \frac{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)y_i}{F_{rz}} - \frac{\sum_{i=1}^{n} m_i \ddot{y}_i z_i}{F_{rz}} - \frac{\sum_{i=1}^{n} (M_x)_i}{F_{rz}}$$

Shih et al. [20] showed through the analysis of various biped motions that both M_x and M_y are of much less magnitude that the vertical ground reaction force, F_{rz}. For this reason, the final terms in the above equations can be neglected.

The ZMP locations are then given by:

$$x_{ZMP} = \frac{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)x_i - \sum_{i=1}^{n} m_i \ddot{x}_i z_i}{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)}$$

$$y_{ZMP} = \frac{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)y_i - \sum_{i=1}^{n} m_i \ddot{y}_i z_i}{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)}$$

Chapter 6

Software Design and Implementation

The purpose of this chapter is to describe how the balance control system is implemented in software. Where applicable, flow-charts and block diagrams are used. Entire code listings are included in the Appendices.

6.1 Introduction

The segregation of software at the simplest level is depicted in Figure 16. The high level code responsible for gait generation, balance control algorithms and data logging is implemented on the system's central processor.

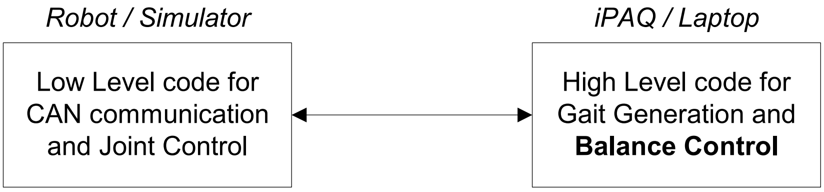

Robot / Simulator
iPAQ / Laptop
Low Level code for CAN communication and Joint Control
High Level code for Gait Generation and **Balance Control**

Figure 16: Software Segregation

The basic structure of the balance control system is shown in Figure 17. It consists of three main modules interacting with the low-level joint controllers via a 50Hz serial link. The primary functions involved with each of the modules are described in Table 3. Detailed descriptions regarding the software implementations are given in the proceeding sections.

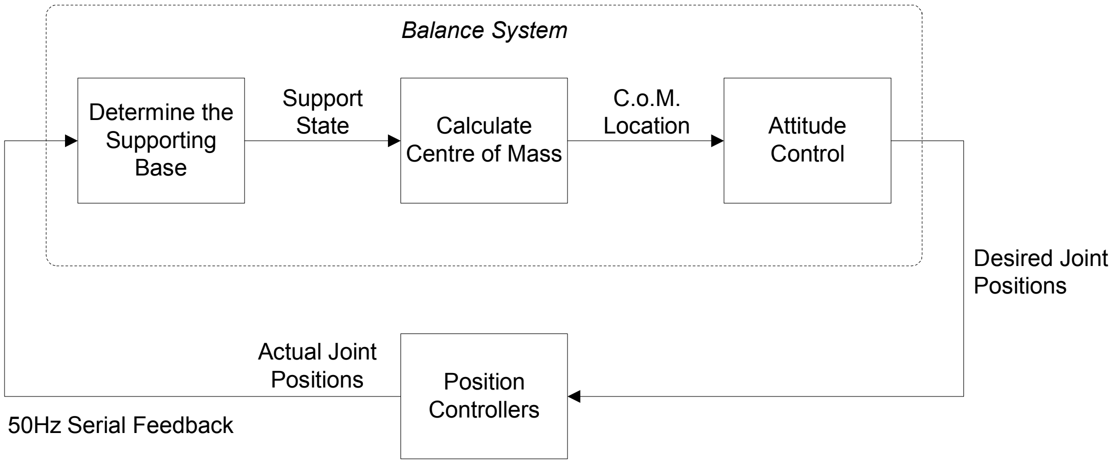


Figure 17: Balance System Block Diagram

Function Name	***Primary Function***
`get_support (void)`	Determines the supporting base of the robot
`calc_com (bal_state)`	Calculates the centre of mass of the robot with respect to the centre of the supporting base
`balance_attitude (bal_state)`	Specifies the desired joint velocities required for dynamic stability. Uses ZMP criteria.

Table 3: Balance System Software Functions

6.2 Initialisation and Matrix Transformations

The balance control code is primarily located in the file `balance.c` within the GuRoo humanoid dual workspace. Since the balance code involves a high level of mathematical computation, efficiency was a large factor influencing the structure of the code.

An `init_balance()` function is called during initialisation of the GuRoo from the `main()` function of `humanoid.cpp`. The purpose of the function is to permanently allocate memory blocks to the storage of 4×4-transformation matrices of type *double*. This is accomplished using the `malloc` library function. The memory locations are constantly accessed throughout the balance code and used for the mapping of points between link coordinate frames. Commonly used transformation matrices such as rotation by 90°, 180° and 270° about the *x*, *y*, and *z*-axes are also calculated and permanently stored in memory.

At the completion of execution of the robot, the `end_balance()` function is run to *free* the allocated memory blocks.

A series of `mat_mult()` functions used for the multiplication of matrices have also been developed. The arguments passed to the functions are pointers to desired output memory locations, as well as pointers to the location of the input matrices. The functions are capable of efficiently multiplying up to four 4×4 matrices.

A full code listing of the functions is shown in Appendix B.

6.3 Determining the Supporting Base

In determining the supporting base of the robot, it is assumed that at all times, the robot is orientated in a stable position with at least one foot *flat* on the ground.

At any time, the supporting base is defined as one of the following three states:

- Right foot on the ground
- Both feet on the ground
- Left foot on the ground

With the absence of foot pressure sensors and global feedback on the GuRoo there is no way of determining the exact orientation of the robot during motion. The current state of the robot with respect to its interaction with the ground can only be estimated through analyse of the joint positions. For this reason, it is only possible to develop a rather crude model.

The proposed method for determining the supporting base of the robot relies solely on analysing joint angles to determine the displacement of each foot with respect to a point at the hips of the robot. The principle is illustrated in Figure 18.

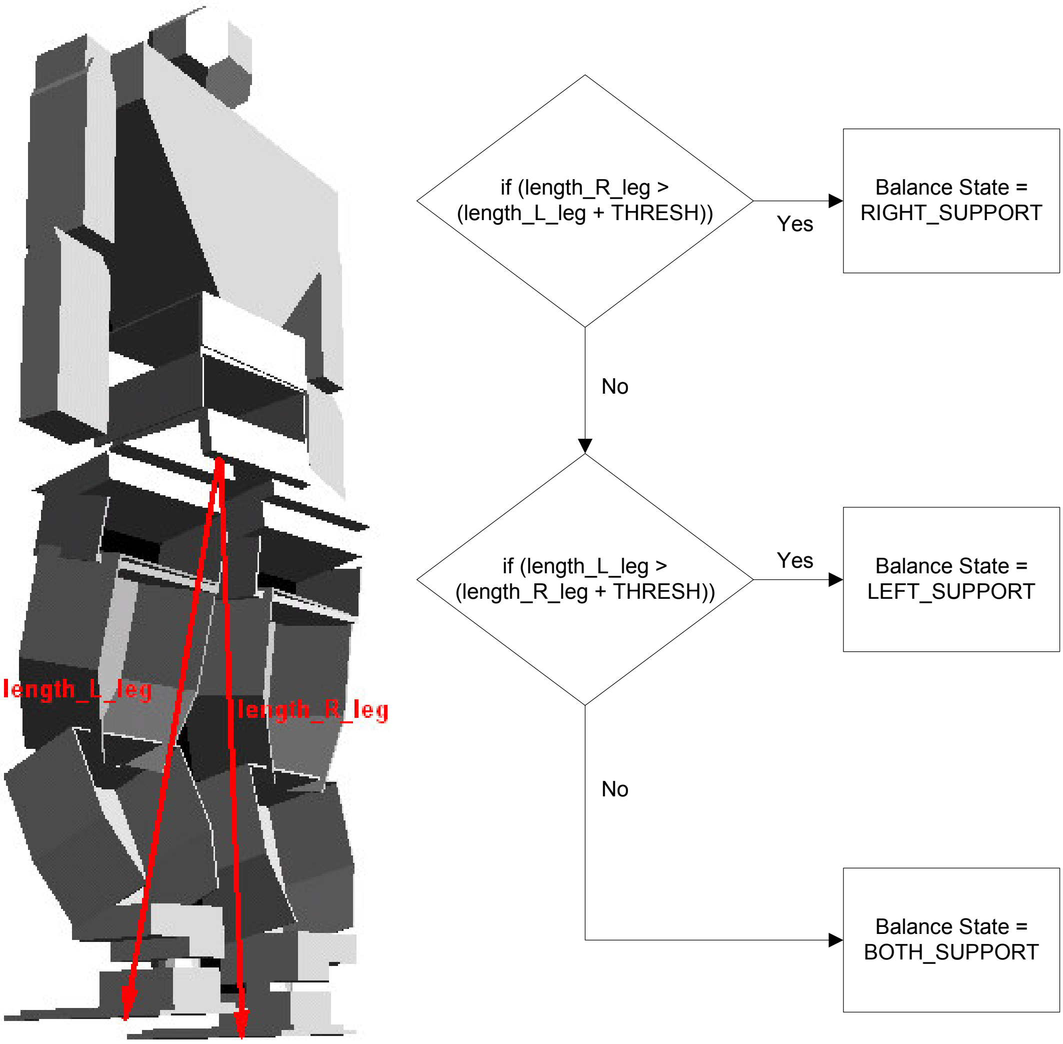

length_L_leg
length_R_leg
if (length_R_leg > (length_L_leg + THRESH))
Yes
Balance State = RIGHT_SUPPORT
No
if (length_L_leg > (length_R_leg + THRESH))
Yes
Balance State = LEFT_SUPPORT
No
Balance State = BOTH_SUPPORT

Figure 18: Method for Determining the Supporting Base

An assumption was made that at any given time, the supporting leg is defined as the one with the largest magnitude for the foot displacement vector. A threshold was defined for the minimum allowable difference in distances for single-support phase. Through experimentation, the threshold was selected to be 0.035. If the difference between the leg lengths is less than the threshold, the robot is defined as being in the double-support phase.

A flowchart describing the method is illustrated in Figure 18. A full code listing of the `get_support()` function is given in Appendix B.

6.4 Centre of Mass

The function of this module is to determine the location of the centre of mass (C.o.M.) of the robot with respect to the reference coordinate frame, which is located at the centre of the supporting base. When determining the location of the centre of mass, the actual position of all 23 actuated joints needs to be considered. The calculations involved require tremendous amounts of computation, as transformation matrices accurately modelling all link and joint parameters need to be considered.

Two centre of mass models have been developed for the GuRoo:

1. Robot C.o.M. Model
2. Simulator C.o.M. Model

6.4.1 Robot Centre of Mass Model

The `calc_com()` function calculates the location of the centre of mass of the robot with respect to the reference coordinate frame. The function accepts one argument defining the supporting base of the robot. The argument passed is the result determined by the `get_support()` function, and can be either `BOTH_SUPPORT`, `RIGHT_SUPPORT` or `LEFT_SUPPORT`.

The calculation of the centre of mass with respect to a fixed coordinate frame for a traditional chain-like structured robot is essentially a straightforward process. The location of the centre of mass of each link can be determined by simple geometry. Using the simplified model of Figure 19 in the *x-z* plane and then in the *y-z* plane, trigonometric relationships are used to derive the following equations:

$$p_{com,x} = \frac{1}{m_T}\sum_{i=1}^{n}\left(m_i b_i + \sum_{j=i+1}^{n} m_j a_i\right)\cos\theta_i$$

$$p_{com,y} = \frac{1}{m_T}\sum_{i=1}^{n}\left(m_i b_i + \sum_{j=i+1}^{n} m_j a_i\right)\sin\theta_i$$

$$p_{com,z} = \frac{1}{m_T}\sum_{i=1}^{n}\left(m_i b_i + \sum_{j=i+1}^{n} m_j a_i\right)\sin\theta_i$$

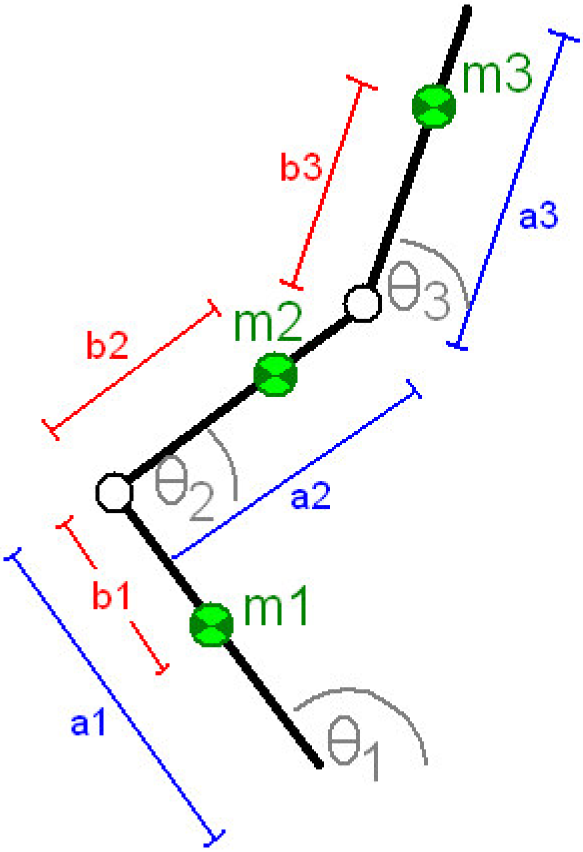


Figure 19: Simplified Model for calculating C.o.M.

Since the GuRoo's mechanical structure is arranged in a star-like configuration, the above equations cannot be directly applied in a single step. The presence of the yaw joints in the legs and torso also add to the difficulty of the analysis since the motion can no longer be easily mapped to planes. For these reasons, it is essential to use a full analysis of rotational and translational transformation matrices as opposed to simple trigonometric relationships to calculate the weighted mass contributions of links.

To reduce the complexity of the system, the C.o.M. is calculated in two stages:

1. Determine the C.o.M. for the legs and waist
2. Determine the C.o.M. for the torso, head and arms.

These masses are then weighted and combined to give the overall C.o.M. for the entire robot. The basic operation of the `calc_com()` function is shown in Figure 20. A full code listing is given in Appendix B.

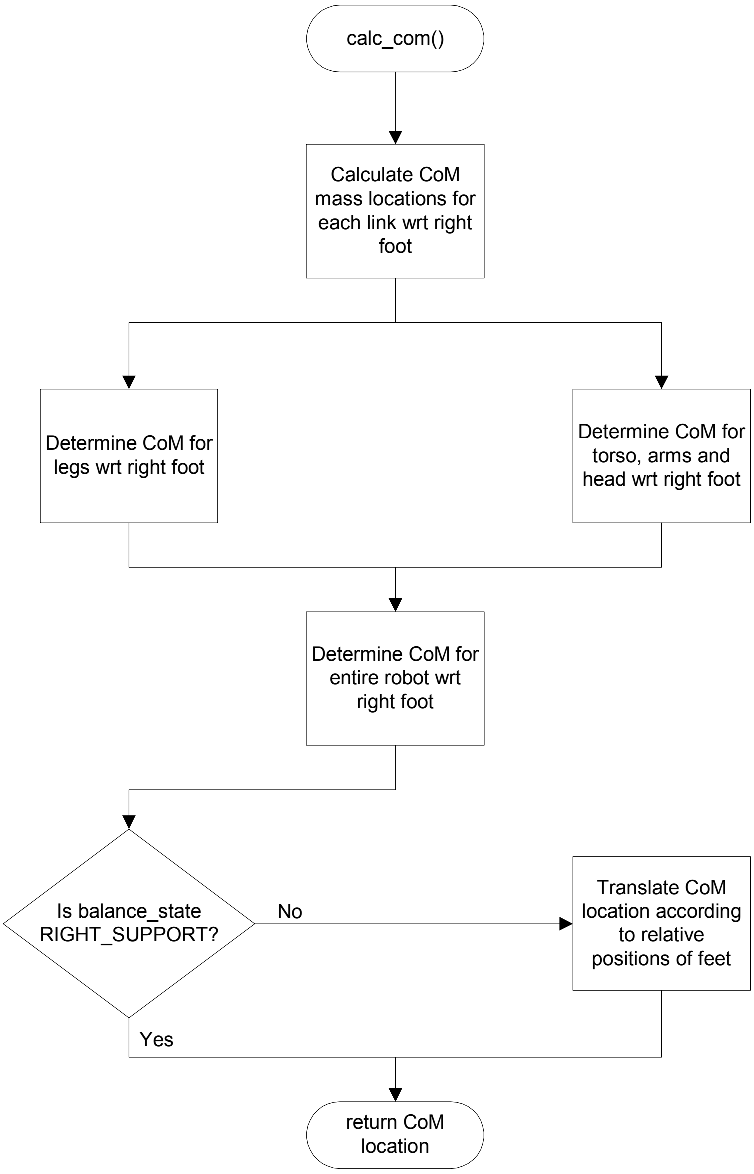

calc_com()
Calculate CoM mass locations for each link wrt right foot
Determine CoM for legs wrt right foot
Determine CoM for torso, arms and head wrt right foot
Determine CoM for entire robot wrt right foot
Is balance_state RIGHT_SUPPORT?
No
Translate CoM location according to relative positions of feet
Yes
return CoM location

Figure 20: Calc_Com() Block Diagram

6.4.2 Simulator Centre of Mass Model

The simulator model, developed with the assistance of C.S.H. from the University of New South Wales, uses the `dmRigidBody` class of the DynaMechs Simulation Library to extract the location of the centre of mass for each link from the simulator. By summing the weighted contributions of the masses of each link, the location of the C.o.M. is computed each time the `UpdateSim()` function of `humanoid.cpp` is run.

A flow-chart illustrating the basic operation of the `getCentreOfGravity()` function is shown in Figure 21. A full code listing of the function is given in Appendix C.

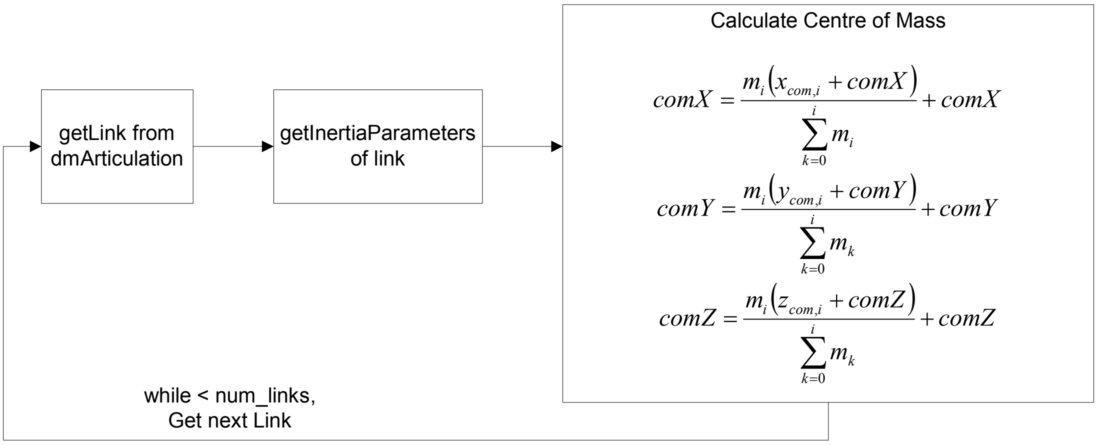

getLink from dmArticulation
getInertiaParameters of link
Calculate Centre of Mass
$comX = \frac{m_i\left(x_{com,i} + comX\right)}{\sum_{k=0}^{i} m_i} + comX$
$comY = \frac{m_i\left(y_{com,i} + comY\right)}{\sum_{k=0}^{i} m_k} + comY$
$comZ = \frac{m_i\left(z_{com,i} + comZ\right)}{\sum_{k=0}^{i} m_k} + comZ$
while < num_links, Get next Link

Figure 21: Simulator C.o.M model Block Diagram

6.5 Attitude Control

The `balance_attitude()` function is responsible for defining the structural movements required for maintaining stability. It does this by specifying desired values for the velocity control of the relevant joints.

The location of the zero moment point is used to specify stability criteria. The location of the ZMP was derived in section 5.2. For the GuRoo, it is given by:

$$x_{ZMP} = \frac{\sum_{i=1}^{23} m_i(\ddot{z}_i + g)x_i - \sum_{i=1}^{23} m_i \ddot{x}_i z_i}{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)}$$

$$y_{ZMP} = \frac{\sum_{i=1}^{23} m_i(\ddot{z}_i + g)y_i - \sum_{i=1}^{23} m_i \ddot{y}_i z_i}{\sum_{i=1}^{n} m_i(\ddot{z}_i + g)}$$

$$z_{ZMP} = 0$$

where,

m_i is the mass of link *i*

(x, y, z) is the position of link *i* wrt the base reference coordinate frame

$(\ddot{x}, \ddot{y}, \ddot{z})$ is the acceleration of link *i* wrt the base reference coordinate frame

g is the acceleration due to gravity (9.81ms^{-2})

Since the only information available regarding the state of the robot is the joint angle position data, the acceleration of link masses needs to be estimated. By updating the position data at each iteration of the 50Hz control loop, the acceleration of link *i* can be determined by first determining its velocity as follows:

$$vel\_com\_x[i] = \frac{(pos\_com\_x[i] - old\_pos\_com[i])}{CENTRAL\_SPEED}$$

$$acc\_com\_x[i] = \frac{(vel\_com\_x[i] - old\_vel\_com[i])}{CENTRAL\_SPEED}$$

where

`CENTRAL_SPEED` = 1 / 50Hz = 0.02s,

the time period for the control loop.

A simple balance scheme was developed to control the desired velocities sent to the ankle joints. The velocities are defined depending on the position of the zero moment point with respect to the centre of the supporting base. The design is based around a simple sign-based control law, which is illustrated in Figure 22.

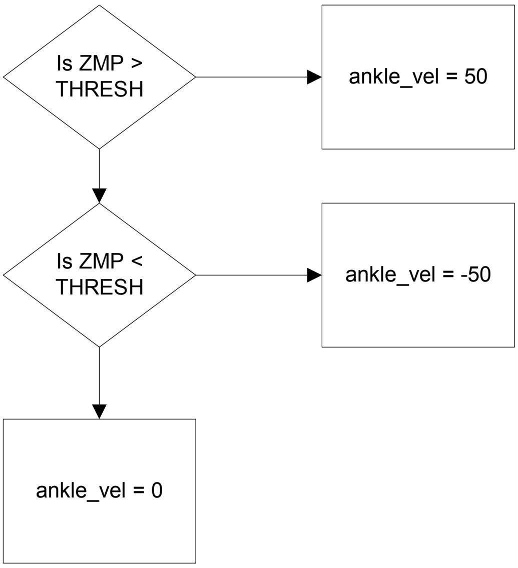


Figure 22: Sign-based Control Law for Balance Attitude

This control algorithm is implemented in the sagittal (pitch) plane where the `ANKLE_FWD` joints are controlled, and in the coronal (roll) plane where the `ANKLE_SIDE` joints are controlled.

A full code listing of the `balance_attitude()` function is given in Appendix B.

Chapter 7

System Evaluation

The purpose of this chapter is to illustrate the success of the project with respect to the criteria outlined in Chapter 3. The criteria used for the evaluation of each model are specified and the results discussed accordingly.

7.1 Determining the Supporting Base

As indicated in section 6.3, the accuracy of the model determining the supporting base of the robot was greatly constrained by the limited amount of available state information. Without the presence of foot pressure sensors and joint torque feedback no definite conclusions could be made regarding the actual interaction of the robot with the ground surface.

The suitability of the model can be investigated by comparing its results to that of an ideal model. Figure 23 illustrates the ideal sequence of support phases for the walking gait developed by Drury [18] and the sequence of support phases derived from the proposed model of section 6.3 in the simulator. The ideal waveform was determined from trajectory analysis.

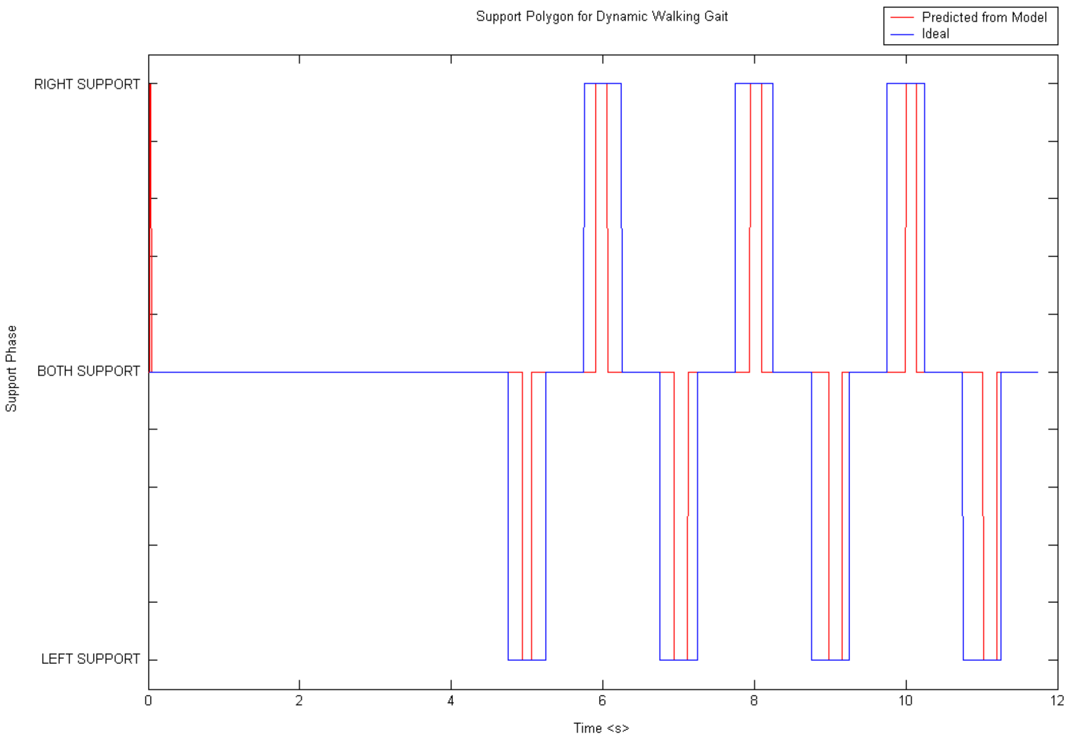


Figure 23: Sequences of Ideal and Predicted Support Phases for the Dynamic Walking Gait

In Figure 23, the robot begins its steady walking cycle at approximately 4.5s. The model correctly predicts the sequence of support phases, but doesn't accurately predict the timing of the phases. Accurate prediction of the support phases is crucial when implementing a balance system, since incorrect assumptions regarding the interaction with the ground will quickly lead to undesired attitude behaviours and severe instability problems. Ideally, it would be desirable for the model to pre-determine future robot positions so the motion could be altered in anticipation of the movements to come.

The purely geometrical approach employed in the model restricts the suitability of its application. The problem is caused by the existence of the time-delay between the moment the robot enters a support-phase and the moment it realises it is actually in that phase. For the walking motions in Figure 23, the maximum value for this time delay is 290ms.

For the purpose of quantifying the accuracy of the model, it can be derived from Figure 23, that the model predicts the correct state of the system for 87% of the time. In Section 3.3.2, it was stated that the desired result for the system was to predict the results obtained from an ideal trajectory analysis for close to 100% of the time.

The model doesn't quite meet the desired specifications, but is the best result that can be achieved with the current hardware configuration. The incorporation and use of force-sensitive pressure sensors in the feet would significantly increase the accuracy of the model.

7.2 Centre of Mass Calculations

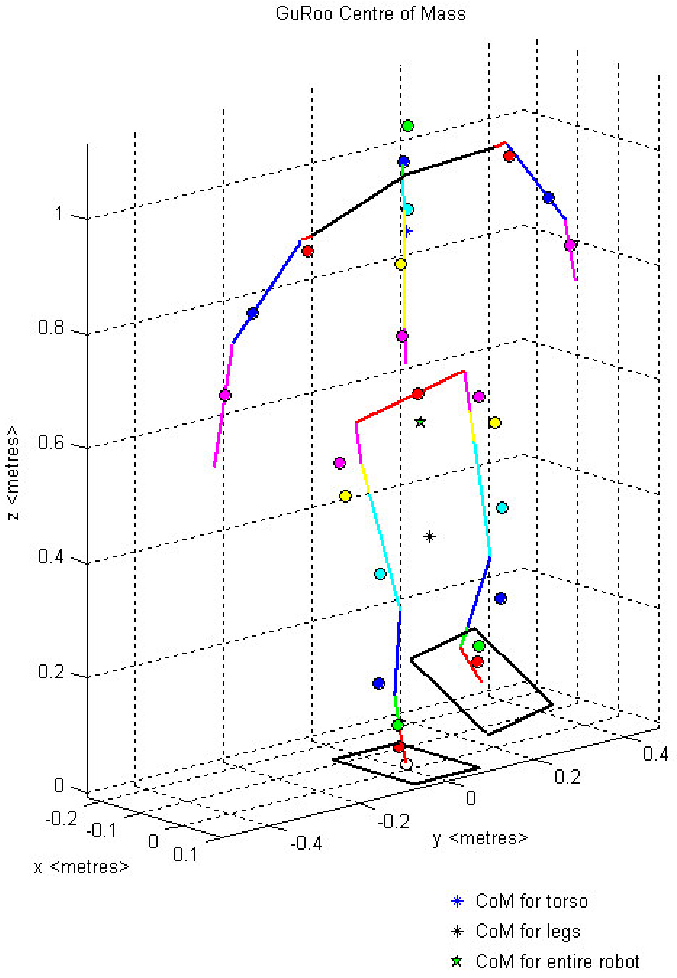
The Centre of Mass (C.o.M.) model developed in section 6.4 gives the location of the C.o.M. for the robot with respect to the centre of its current support polygon. In order to visualise the results, a MATLAB function was developed. The MATLAB function accepts a single vector argument listing all joint angles for the robot, and determines the location of the C.o.M. for each link. The results are plotted graphically using the MATLAB `plot3` function. A sample plot is shown in Figure 24.

The correctness of the centre of mass model can be evaluated by comparing its results with the CAD representation of the GuRoo in *Solid Edge*. Figure 25 compares the location of the C.o.M. for various structural positions. In the figure, all distances are relative to the centre point of the right foot base.

Figure 25a illustrates the GuRoo in a standing position. As seen, the results from the centre of mass model closely match those obtained from the CAD model. The largest discrepancy is the *z*-component, which indicates an absolute error of 32.2mm. The *y*-component exhibited an absolute error of only 0.2mm.

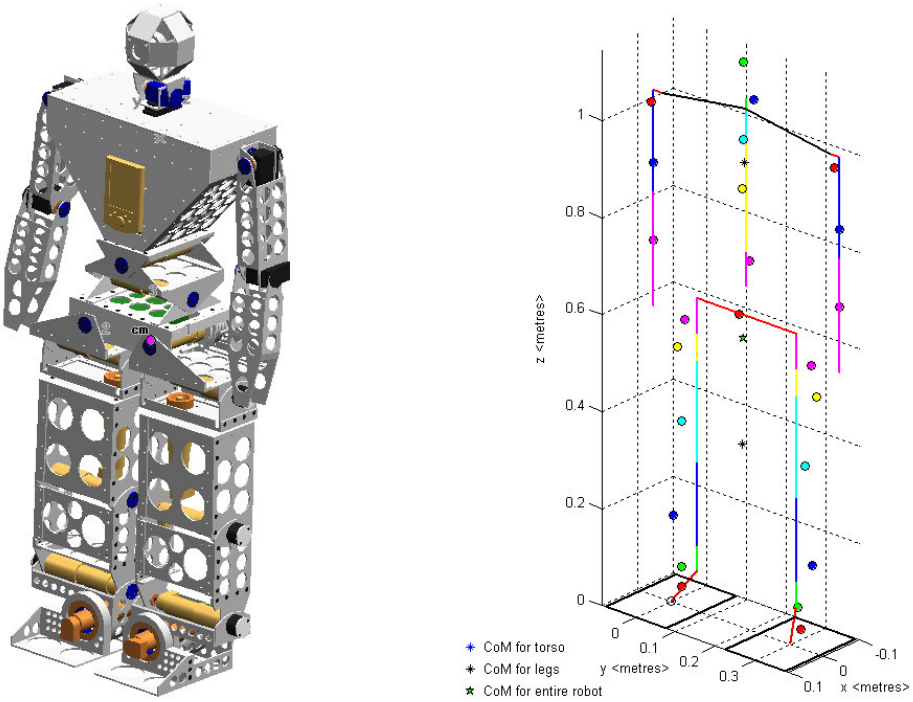
In Figure 25b, the `LEFT_HIP_SIDE` joint has been rotated by -60° and the `LEFT_KNEE` joint has been rotated by 90°. Again, the largest absolute error was in the *z*-direction, and was 42.6mm. The best result was in the *x*-direction with an absolute error of 0.9mm.

The discrepancies in the results can be attributed to the simplifications made when defining the *dm*-parameters. The absence of some parts from the model such as the iPAQ, circuit boards, servo-motors and batteries directly correlate to the slight variation that was observed in the results.



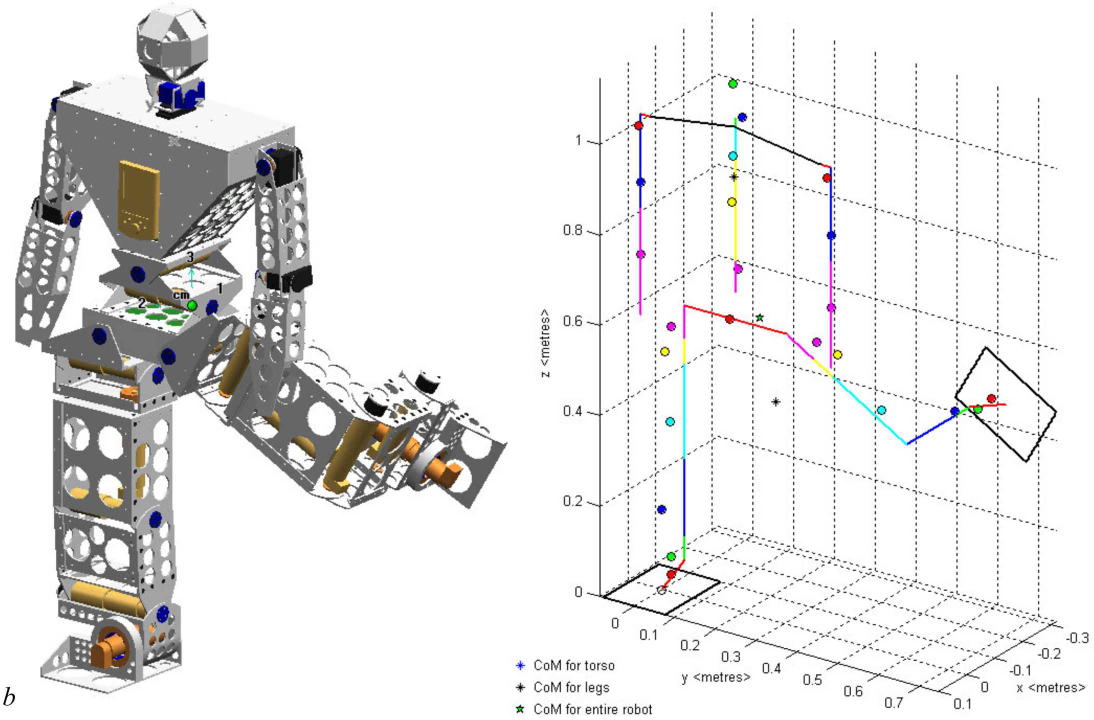
Joint	**Angle <degrees>**	**Joint**	**Angle <degrees>**
LEFT_ANKLE_SIDE	0	RIGHT_ANKLE_SIDE	-10
LEFT_ANKLE_FWD	0	RIGHT_ANKLE_FWD	-15
LEFT_KNEE	30	RIGHT_KNEE	-25
LEFT_LEG_TWIST	0	RIGHT_LEG_TWIST	0
LEFT_HIP_FWD	0	RIGHT_HIP_FWD	-10
LEFT_HIP_SIDE	0	RIGHT_HIP_SIDE	0
LEFT_SHOULDER	-10	RIGHT_SHOULDER	0
LEFT_UPPER_ARM	30	RIGHT_UPPER_ARM	50
LEFT_LOWER_ARM	30	RIGHT_LOWER_ARM	-30
TORSO_FWD	-10	NECK	0
TORSO_SIDE	0	HEAD	0
TORSO_TWIST	0		

Figure 24: 3D C.o.M. Plot in MATLAB



a

Centre of Mass Location	Solid Edge Model	C.o.M. Model
x	-0.0414	-0.0362
y	0.1497	0.1495
z	0.6057	0.5735



b

Centre of Mass Location	Solid Edge Model	C.o.M. Model
x	-0.0588	-0.0579
y	0.1981	0.2008
z	0.6736	0.6310

Figure 25: Evaluation of C.o.M. Model

For the walking gait proposed by Drury [18], the location of the centre of mass can be traced and plotted as shown in Figure 26. The figure compares the C.o.M. trajectories determined from desired and actual joint angles to the results determined directly from the simulator using the method described in section 6.4.2. The differences between the trajectories of the desired joint angles and the actual joint angles can be attributed to the static errors in the C.o.M. model as previously discussed, as well as the non-ideal behaviour of the joint control system of the robot. The problems in the joint control system include the method by which data is logged.

The deviation of the C.o.M. trajectory determined directly from the simulator is a direct result of the simulated robot environment. The main reason is due to the interactions between links not being correctly damped.

In section 3.3.3, it was stated that the success of the model could be evaluated by considering the accuracy of the results when compared to the ideal results obtained from the simulator. From the data of Figure 26, the standard deviation of the results is given in Table 4. The average of the values is 0.4114 and is clearly within the desired specifications. The model gives a valid approximation for the location of the centre of mass for the robot.

Centre of Mass Location	***Standard Deviation***
x	0.3407
y	0.7578
z	0.1357

Table 4: Standard Deviation of C.o.M. Comparison

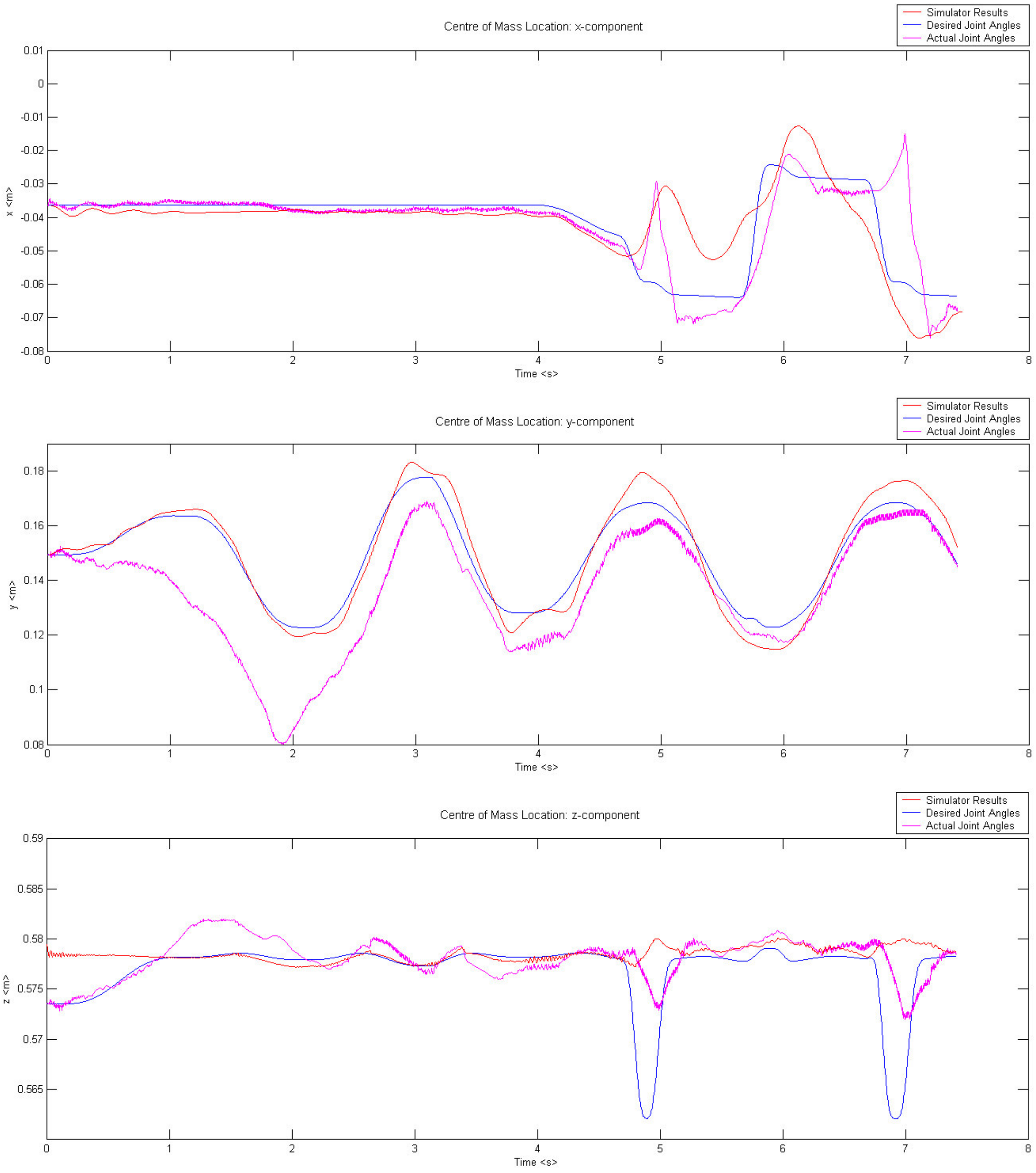


Figure 26: C.o.M. Locations for Walking Gait

7.3 Attitude Control

It is difficult to evaluate the success of the attitude control model, since its operation greatly depends on the performance of other systems of the GuRoo. These systems include the other balance modules discussed in the preceding sections of this chapter, and the joint control system developed by Drury [18].

As indicated in section 3.3.4, the balance control system can be evaluated by considering the extent to which the robot can resist adverse disturbances, the smoothness of the joint motion and by minimising the load torques on each of the joints. Since only a simple model was designed with the intention of investigating the success of the other balance modules, the following procedure was used:

1. Shift the weight of the robot over the right foot by defining desired velocities for the ankles and hips
2. Raise the left leg using the procedure proposed by Dury [18]
3. Raise the left leg a further 30° by moving the LEFT_HIP_FWD joint
4. Move the left leg to the side by 40° using the LEFT_HIP_SIDE joint
5. Bend the right knee by 15°
6. Tilt to torso by moving the `TORSO_SIDE` joint -20°
7. Twist the torso by moving the `TORSO_TWIST` joint -20°
8. Hold the final position

The control code specifying these movements is given in Appendix B.3.

During stage 1, velocities are pre-specified for the ankle joints. This is because the “supporting base model” is unable to accurately identify the supporting base of the robot. The reason for this is the error caused by the time delay described in section 7.1. During the remaining stages of the test, the model is correctly able to identify the supporting base of the robot, and the attitude control module specifies relevant joint velocities for the ankles depending on the location of the zero moment point.

The location of the zero moment point throughout the sequence of movements is shown in Figure 27. The oscillations that are visible in the waveforms are due to a combination of the lack of damping in the simulator environment and the way in which the desired velocities are defined.

Figure 28 shows the desired ankle joint velocities that are specified by the attitude control module. The final position of the robot at the end of the series of movements is shown in Figure 29.

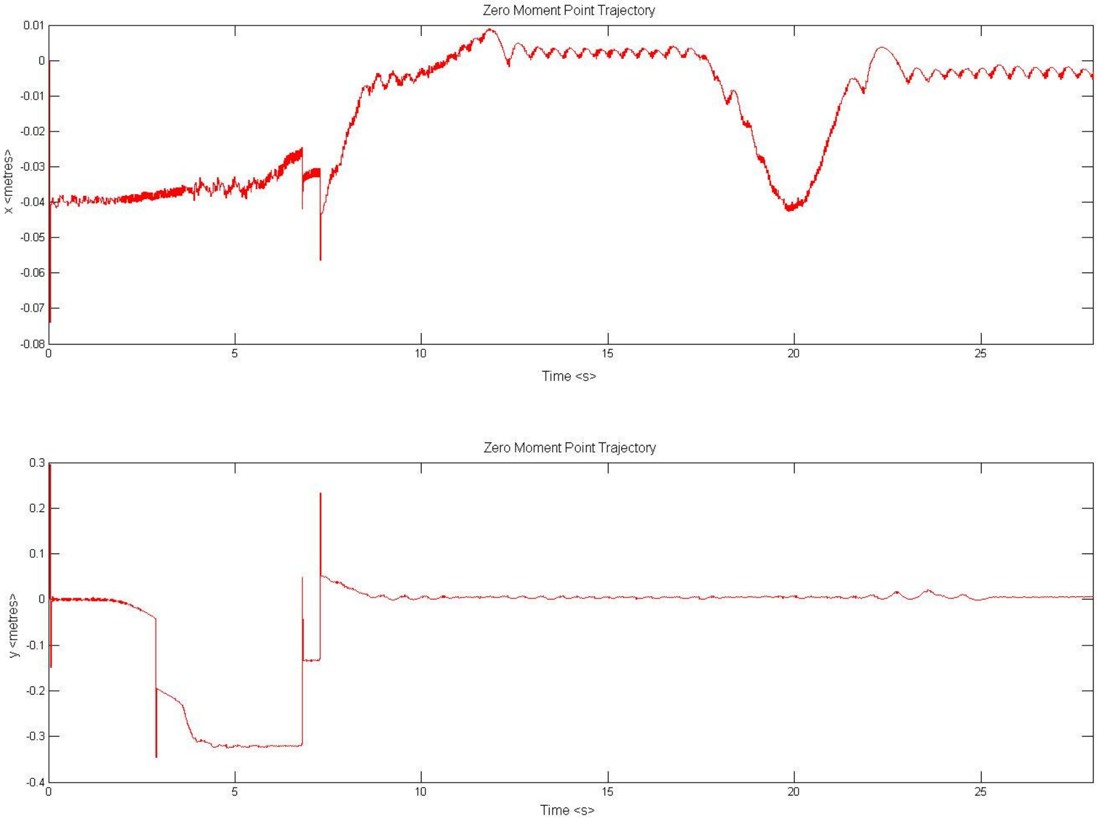
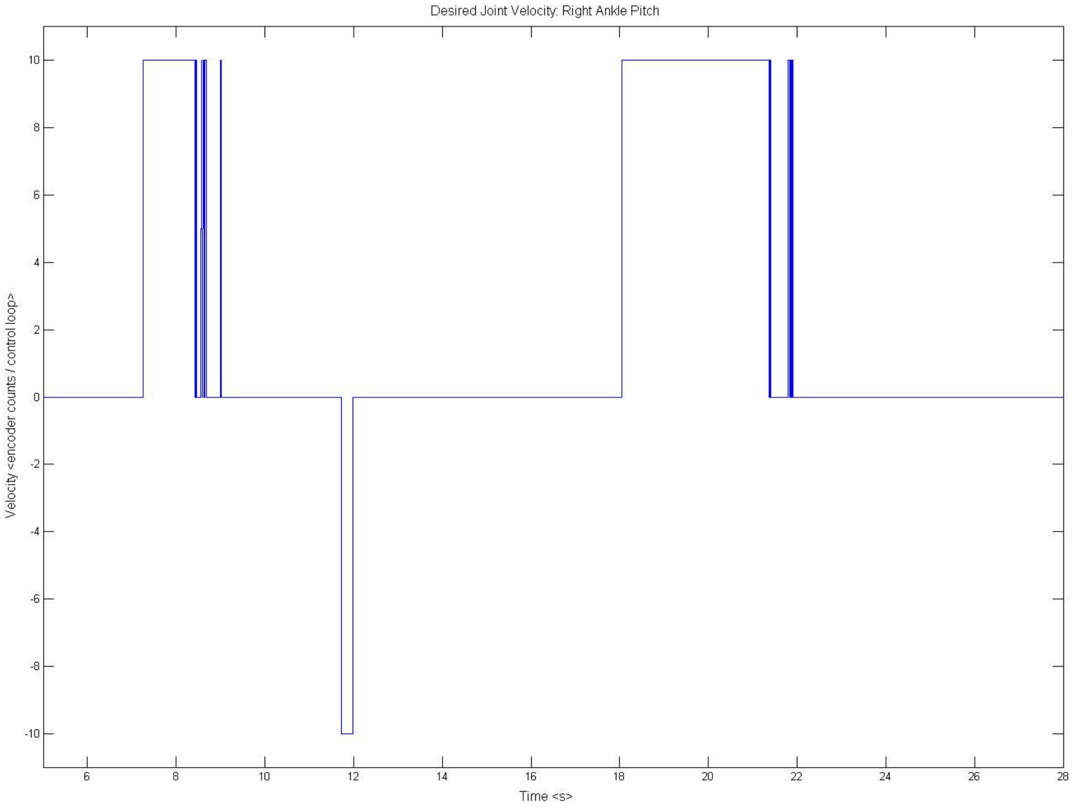


Figure 27: Location of the ZMP for the Dynamic Walking Gait



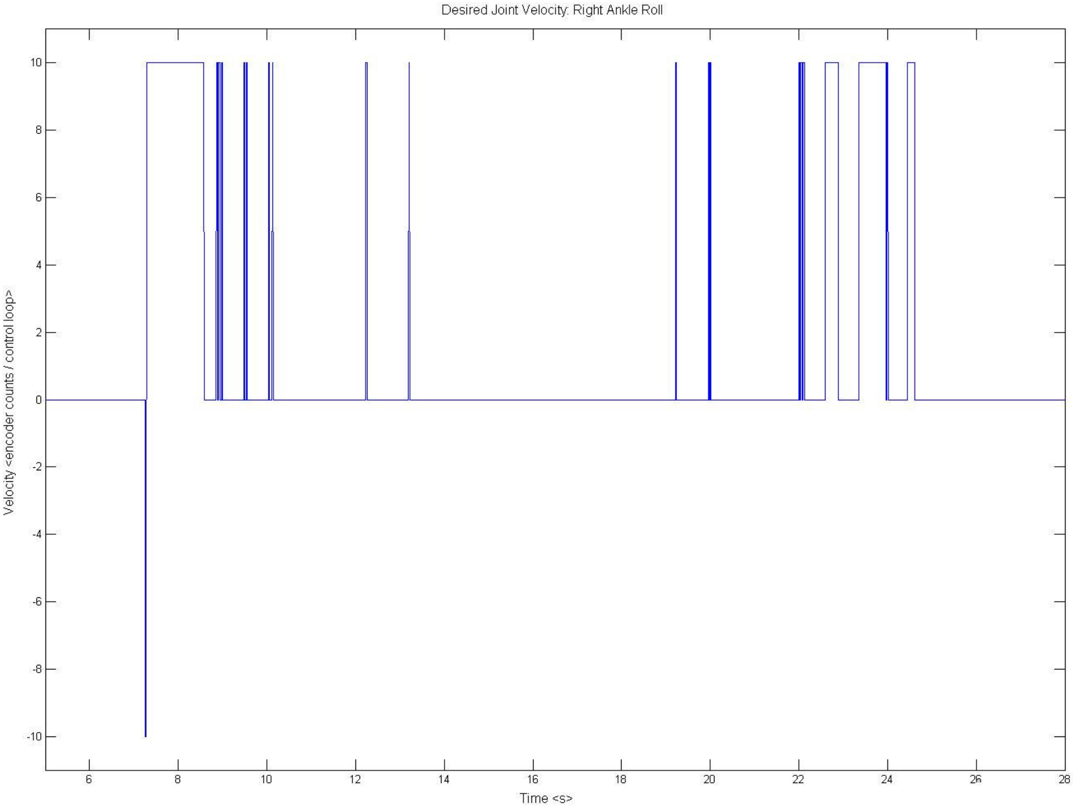


Figure 28: Desired Joint Velocities specified by the Attitude Control System

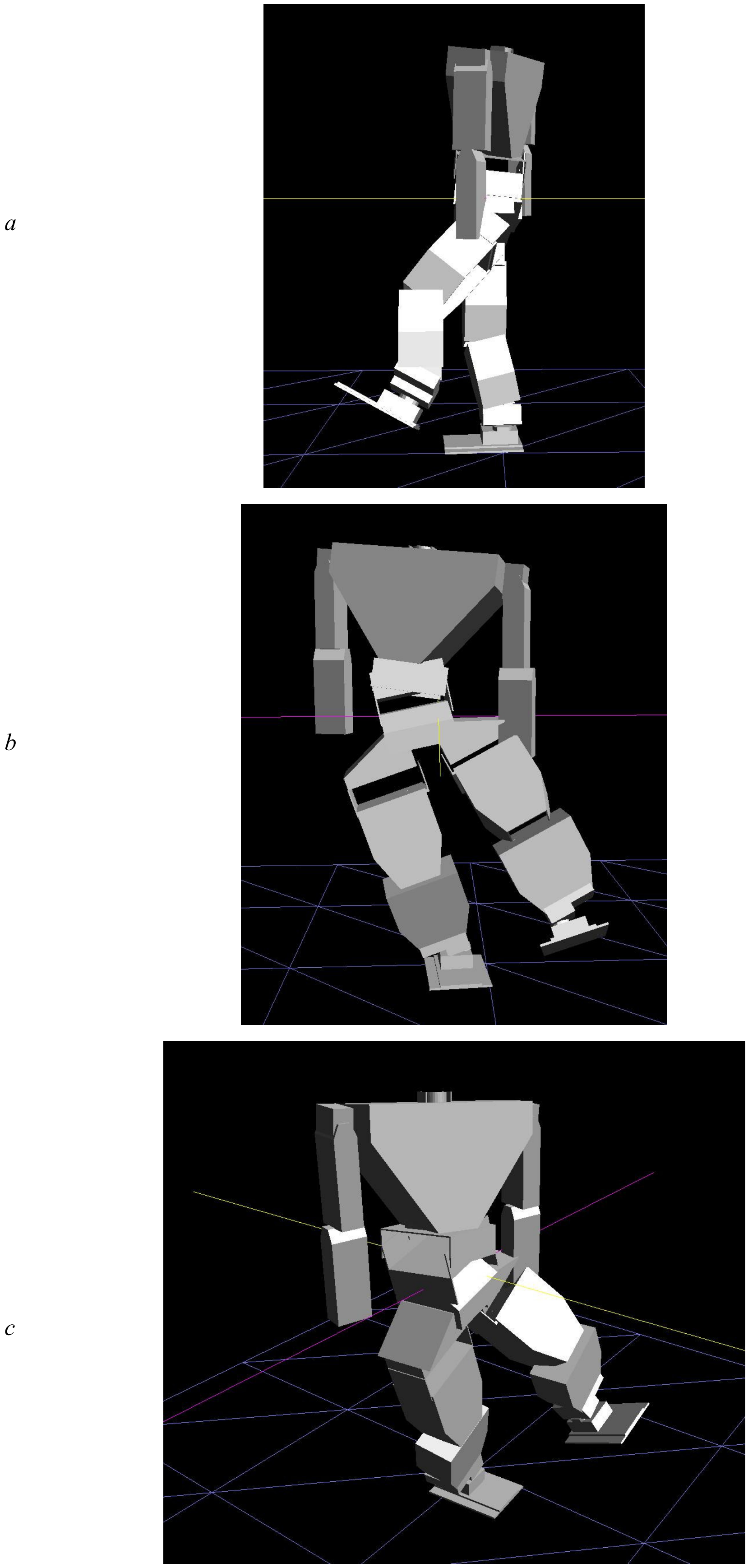


Figure 29: Stable Position of the GuRoo

a) Side View, *b) Front View,* *c) Isometric View*

The extent to which the attitude control module could be successful was significantly limited by the lack of feedback information. Testing was unable to be performed on the real robot because the serial feedback of all joint angles was not achieved. The reason for this was attributed to memory and timing problems - the speed of the connection was too slow and the allocation of memory was inadequate. The absence of additional sensors such as gyroscopes, accelerometers and force-sensitive sensors also limited the scope of the design.

The design that was developed, although not meeting all of the original specifications, can be considered successful, since control of the ankle joints did allow the robot to balance in the simulator. Compensation for a wide range of motions was successfully demonstrated.

Chapter 8

Conclusions and Future Developments

This chapter reflects on the goals of this thesis and the extent to which they were achieved. Suggestions for future developments of the balance control system are also described.

8.1 Conclusions

The aim of this thesis was to develop an active balance control system for a humanoid robot and implement it in hardware as part of the University of Queensland's GuRoo project. Ideally, the balance system would allow the robot to remain upright despite adverse disturbances such as those due external forces and uneven terrain.

The balance control system is a high-level behavioural controller for the robot and is greatly dependent on lower-level systems such as the joint controllers and feedback networks. The course of the project saw the development of the robot from a simulated on-paper design to a functional hardware prototype capable of performing simple movements such as crouching, standing on one leg and open-loop walking. Despite making impressive progress throughout the year, the robot was not developed to a stage where an active balance control system could effectively be implemented in real hardware. The reason for this is attributed to the fact that no global feedback was achieved.

With the resources and time available, the initial goal of developing and demonstrating a fully functional balance system on the GuRoo was an unreasonable task. However, the project can be considered a success since specifications were achieved through simulation. Additionally, a platform has been formed in a continuable way to allow further testing, development and implementation of the balance system on real hardware.

8.2 Future Developments

The future developments associated with implementing a robust and reliable balance control system for the GuRoo are quite clear.

Initially, a system providing correct feedback of the actual joint positions from the optical encoders needs to be implemented. It is evident that the existing serial communications link between the central controller and the distributed control network on the CAN bus is inadequate for the transmission of large amounts of data. The proposed high-speed USB-to-CAN interface would allow for more accurate and efficient data transmission.

In order for the attitude controller to correctly respond to the current state of the robot, the inadequacies of the “Supporting Base Model” need to be resolved. This cannot be achieved until force-sensitive touch sensors are incorporated into the feet of the robot. Feedback from these sensors would allow for the exact interaction between the robot and the walking surface to be recognised. The information would also allow for the precise computation of the actual zero moment point and provide additional knowledge for the specification of attitude behaviours.

The largest scope for development of the existing model lies in the “Attitude Control System” – one of the most exciting areas in modern robotics research. Once the abovementioned restrictions are overcome, the extent to which the attitude controller can be developed is limitless. An initial development would be in the modelling of a double-inverted-pendulum, whereby one pendulum represents the legs of the robot, and a second models the torso. The current "Centre of Mass Model" developed for the GuRoo is already structured for this development. Implementation of the double-inverted pendulum would allow for posture control of the robot. If the data-logging problems are corrected, it is hoped that the double-inverted-pendulum model will be implemented on the real robot by demo-day on the 29th October, 2002.

An improved method of deriving the desired joint movements needed to maintain stability can be developed. With the development of a torque-joint control (stiffness control) system, desired velocities for each joint would not need to be defined. A simple search algorithm investigating the current applied torques and all possible combinations of predicted future torques for each of the joints could be used. By minimising the applied torques, new attitudes can be specified. The software that has been written for the transformation mathematics has been structured with this development in mind.

A further development would be the incorporation of multiple accelerometers or a gyroscope package into the robot hardware design. These can be used to measure the direction and magnitude of the acceleration of the centre of mass of the robot. Combined with use of force-sensitive touch sensors and torque control, advanced techniques for impact absorption and trajectory planning can be developed.

References

[1] C. Willis, "Android World - Anthropomorphic Robots & Animatronics," vol. 2002: http://www.androidworld.com/, 2002.

[2] Robocup, "RoboCup 2002 - General Information," http://www.robocup2002.org/info/index.html, 2002.

[3] S. McMillan, "Computational Dynamics for Robotic Systems on Land and Under Water," in *Department of Electrical Engineering*. Ohio: The Ohio State University, 1995.

[4] A. W. Smith, "Simulator Adaption and Gait Pattern Creation for a Humanoid Robot," in *School of Information Technology and Electrical Engineering*: University of Queensland, 2001.

[5] Honda Motor Co Ltd, "The Honda Humanoid Robot ASIMO," vol. 2002: http://world.honda.com/ASIMO, 2002.

[6] Sony Corporation, "SDR-4X Press Release," vol. 2002: http://www.sony.co.jp/en/SonyInfoNews/Press/200203/02-0319E/, 2002.

[7] Y. N. Tomomichi Sugihara, Hirochika Inoue, "Realtime Humanoid Motion Generation through ZMP Manipulation based on Inverted Pendulum Control," presented at International Conference on Robotics & Automation, Washington DC, 2002.

[8] A. Goswami, "Foot rotation indicator (FRI) point: A new gait planning tool to evaluate postural stability of biped robots."

[9] S. K. Qiang Huang, Noriho Koyachi, Kenji Kaneko, Kazuhito Yokoi, Hirohiko Arai, Kiyoshi Komoriya, Kazuo Tanie, "A High Stability, Smooth Walking Pattern for a Biped Robot," presented at International Conference on Robotics & Automation, 1999.

[10] Y. K. R. Jong H. Park, "ZMP Trajectory Generation for Reduced Trunk Motions of Biped Robots," presented at IEEE International Conference on Intelligent Robots and Systems, 1998.

[11] Y. K. T. Fukuda, T. Arakawa, "Stabilization control of biped locomotion robot based learning with gas having self-adaptive mutation and recurrent neural networks," presented at Robotics and Automation, 1997.

[12] F. K. Shuuji Kajita, Kenji Kaneko, Kazuhito Yokoi, Hirohisa Hirukawa, "The 3D Linear Inverted Pendulum Mode: A simple modeling for a biped walking pattern generation," presented at International Conference on Intelligent Robots and Systems, Maui, Hawaii, USA, 2001.

[13] A. T. Q. Li, I. Kato, "Learning of walking stabilization for a biped robot with a trunk," presented at 2nd Asian Conference on Robotics and Its Applications, Beijing, 1994.

[14] K. D. K. Jong H. Park, "Biped Robot Walking Using Gravity-Compensated Inverted Pendulum Mode and Computed Torque Control," presented at International Conference on Robotics & Automation, Leuven, Belgium, 1998.

[15] M. A. R. Caballero, V. Sanchez, "Extending Zero Moment Point to a Segment Using Reduced Order Biped Model."

[16] W. G. A. Albert, "New path planning algorithms for higher gait stability of a bipedal robot," presented at 4th International Conference on Climbing and Walking Robots, Germany, 2001.

[17] P. I. Corke, *Robotics Toolbox*. Preston: CSIRO, 1996.

[18] A. Drury, "Gait Generation and Control Algorithms for a Humanoid Robot," University of Queensland, 2002.

[19] H. C. C. Jong H. Park, "An On-Line Trajectory Modifier for the Base Link of Biped Robots To Enhance Locomotion Stability," presented at Internation Conference on Robotics & Automation, San Fransisco, CA, 2000.

[20] Y. Z. L. C. L. Shih, S. Churng, T. T. Lee, W. A. Gruver, "Trajectory Synthesis and Physical Admissibility for a Biped Robot During the Single-Support Phase," presented at IEEE International Conference on Robotics & Automation, Cincinnati, Ohio, 1990.

Appendix A: Centre of Mass Locations

This table describes the physical parameters for each link of the GuRoo. The mass and centre of mass locations were extracted from the human_rev6_1.dm file of the simulator. The data was originally sourced from the solid edge workspace.

Link Number	***Link Description***	***Mass <kg>***	***Centre of Mass Location <(x, y, z) in metres>***
R	Right Foot	0.826	(0.03, 0.02, -0.02)
4	Right Ankle	1.42	(0.04, 0.02, 0.02)
3	Right Lower Leg	1.791	(0.12, -0.01, -0.05)
5	Right Upper Leg	2.856	(0.01, 0.03, 0.13)
11	Right Lower Hip	0.583	(0.04, 0.00, 0.05)
9	Right Hipper Hip	1.617	(0.06, -0.04, -0.01)
10	Waste	4.77	(-0.01, 0.125, -0.02)
7	Left Upper Hip	1.616	(-0.015, -0.01, -0.03)
6	Left Lower Hip	0.581	(0.00, -0.05, -0.015)
8	Left Upper Leg	2.849	(-0.08, -0.01, 0.03)
2	Left Lower Leg	1.789	(-0.052, 0.01, -0.05)
0	Left Ankle	1.443	(-0.011, -0.02, -0.02)
L	Left Foot	0.826	(-0.025, 0.005, 0.025)
13	Lower Torso	1.47	(0.00, -0.02, -0.01)
14	Upper Torso	1.579	(0.00, -0.01, 0.10)
T	Outer Torso	8.39	(0.06, 0.00, -0.01)
21	Neck	0.215	(-0.01, 0.01, -0.02)
22	Head	0.421	(-0.08, 0.01, 0.00)
15	Left Shoulder	0.225	(0.02, -0.01, 0.02)
16	Left Upper Arm	0.361	(0.15, 0.00, 0.00)
17	Left Lower Arm	0.208	(0.10, 0.00, 0.00)
18	Right Shoulder	0.224	(0.02, -0.01, -0.02)
19	Right Upper Arm	0.361	(0.15, 0.00, 0.00)
20	Right Lower Arm	0.208	(0.10, 0.00, 0.00)

Appendix B: C/C++ CODE

B.1 - Balance.h

```
// constant definitions
#define g              9.81
#define LEFT_SUPPORT   -1
#define BOTH_SUPPORT   0
#define RIGHT_SUPPORT  1
#define CENTRAL_SPEED  0.02
#define BAL_THRESH     0.0035
#define ZMP_THRESH     0.001

// specify function prototypes
void balance_control (void);

int get_support (void);
double *calc_com (int balance_state);
void balance_attitude (int balance_state);

void init_balance(void);
double **create_matrix(void);
void unpack_joint_angles (char* incoming);
void end_balance(void);
void release_matrix(double **matrix);

void mdh (double **ptr_out, double a, double alpha, double d, double theta);
void trans (double **ptr_out, double x, double y, double z);
void Rx (double **ptr_out, double theta);
void Ry (double **ptr_out, double theta);
void Rz (double **ptr_out, double theta);

void mat_mult2 (double **ptr_out, double **ptr_a, double **ptr_b);
void mat_mult3 (double **ptr_out, double **ptr_a, double **ptr_b, double **ptr_c);
void mat_mult4 (double **ptr_out, double **ptr_a, double **ptr_b, double **ptr_c,
double **ptr_d);
void vec_mult (double *ptr_out, double **matrix, double *ptr_vector);
```

B.2 - Balance.c

```
/*
 *      Author: Ian Marshall
 *
 *      Code for Balance System
 *
 */
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <dmLink.hpp>

#include "balance.h"
#include "../Common/humanoid.h"
#include "jointnum.h"


// location of centre of mass of each link with respect to robot coordinate frame
double comR[4];
double com4[4];
double com3[4];
double com5[4];
double com11[4];
double com9[4];
double com10[4];
double com7[4];
double com6[4];
double com8[4];
double com2[4];
double com0[4];
double comL[4];
double com13[4];
double com14[4];
double comT[4];
double com21[4];
double com22[4];
double com15[4];
double com16[4];
double com17[4];
double com18[4];
double com19[4];
double com20[4];


//int tmp_bal_state;
double tmp_zmp_x, tmp_zmp_y, tmp_zmp_z;


// Masses of each link
double mR = 0.826;
double m4 = 1.42;
double m3 = 1.791;
double m5 = 2.856;
double m11 = 0.583;
double m9 = 1.617;
double m10 = 4.77;
double m7 = 1.616;
double m6 = 0.581;
double m8 = 2.849;
double m2 = 1.789;
double m0 = 1.443;
double mL = 0.826;
double m13 = 1.47;
double m14 = 1.579;
double mT  = 8.39;
double m21 = 0.215;
double m22 = 0.421;
double m15 = 0.225;
double m16 = 0.361;
double m17 = 0.208;
double m18 = 0.224;
double m19 = 0.361;
double m20 = 0.208;


// initialise arrays for storing calculated centre of mass
double LwrtR[4];
double com_legs[3];
double com_torso[3];
double com_total[3];   // centre of mass for the entire robot
double state_com[3];   // centre of mass in 3D polar coordinates
                                              // [theta_roll, theta_pitch, r]
// initialise link transformation matrices
double **T_RwrtOrgn;
double **T_4wrtR;
double **T_3wrt4;
double **T_5wrt3;
double **T_11wrt5;
double **T_9wrt11;
double **T_10wrt9;
double **T_7wrt10;
double **T_6wrt7;
double **T_8wrt6;
double **T_2wrt8;
double **T_0wrt2;
double **T_Lwrt0;

double **T_13wrtOrgn;
double **T_14wrt13;
double **T_Twrt14;
double **T_21wrtT;
double **T_22wrt21;

double **T_15wrtT;
double **T_16wrt15;
double **T_17wrt16;

double **T_18wrtT;
double **T_19wrt18;
double **T_20wrt19;

double **T_Rorigin;
double **T_4origin;
double **T_3origin;
double **T_5origin;
double **T_11origin;
double **T_9origin;
double **T_10origin;
double **T_7origin;
double **T_6origin;
double **T_8origin;
double **T_2origin;
double **T_0origin;
double **T_Lorigin;

double **T_13origin;
double **T_14origin;
double **T_Torigin;
double **T_21origin;
double **T_22origin;
double **T_15origin;
double **T_16origin;
double **T_17origin;
double **T_18origin;
double **T_19origin;
double **T_20origin;


// initialise arrays to store commonly used transformation matrices
double **RxnegPIdiv2;
double **RxPIdiv2;
double **RxPI;

double **RynegPIdiv2;
double **RyPIdiv2;
double **RyPI;

double **RznegPIdiv2;
double **RzPIdiv2;
double **RzPI;


// initialise matrix arrays to store temporary calculations
double **tmp1;
double **tmp2;
double **tmp3;
double **tmp4;
double **tmp5;
double **tmp6;


// define error vector
double error_ptr[] = {-1, -1, -1, -1};


// location of centre of mass of each link with respect to it's own coordinate axes
// develop: get directly from dm file
double local_comR[] = {0.03, 0.02, -0.02, 1};
double local_com4[] = {0.04, 0.02, 0.02, 1};
double local_com3[] = {0.12, -0.01, -0.05, 1};
double local_com5[] = {0.01, 0.03, 0.13, 1};
double local_com11[] = {0.04, 0, 0.05, 1};
double local_com9[] = {0.06, -0.04, -0.01, 1};
double local_com10[] = {-0.01, 0.125, -0.02, 1};
double local_com7[] = {-0.015, -0.01, -0.03, 1};
double local_com6[] = {0.00, -0.05, -0.015, 1};
double local_com8[] = {-0.08, -0.01, 0.03, 1};
double local_com2[] = {-0.052, 0.01, -0.05, 1};
double local_com0[] = {-0.011, -0.02, -0.02, 1};
double local_comL[] = {-0.025, 0.005, 0.025, 1};
double local_com13[] = {0.00, -0.02, -0.01, 1};
double local_com14[] = {0.00, -0.01, 0.10, 1};
double local_comT[] = {0.06, 0.00, -0.01, 1};
double local_com21[] = {-0.01, 0.01, -0.02, 1};
double local_com22[] = {-0.08, 0.01, 0.00, 1};
double local_com15[] = {0.02, -0.01, 0.02, 1};
double local_com16[] = {0.15, 0.00, 0.00, 1};
double local_com17[] = {0.10, 0.00, 0.00, 1};
double local_com18[] = {0.02, -0.01, -0.02, 1};
double local_com19[] = {0.15, 0.00, 0.00, 1};
double local_com20[] = {0.10, 0.00, 0.00, 1};

double old_pos_com_x[24];
double old_pos_com_y[24];
double old_pos_com_z[24];

double old_vel_com_x[24];
double old_vel_com_y[24];
double old_vel_com_z[24];


extern int* desired_joint_vel[TOTAL_MOTORS];

typedef struct {
        double theta0;
        double theta1;
        double theta2;
        double theta3;
        double theta4;
        double theta5;
        double theta6;
        double theta7;
        double theta8;
        double theta9;
        double theta10;
        double theta11;
        double theta12;
        double theta13;
        double theta14;
        double theta15;
        double theta16;
        double theta17;
        double theta18;
        double theta19;
        double theta20;
        double theta21;
        double theta22;
        double theta23;

} joint_angles_t;

joint_angles_t angles;


void balance_control (void) {
        int k, j;
        int bal_state;
        float joint_pos[1];
        float joint_vel[1];
        extern dmLink **robot_link;
        float joint_angles[23];

#ifndef TX
        for (k = 0; k < 23; k++) {
                j = Joint_Conversion(k);
                robot_link[j]->getState(joint_pos, joint_vel);

                if (j == SIM_RIGHT_KNEE || j == SIM_LEFT_KNEE )
                        joint_pos[0] -= PI/2;

                joint_angles[k] = joint_pos[0];
        }
                angles.theta0 = joint_angles[0];
                angles.theta1 = joint_angles[1];
                angles.theta2 = joint_angles[2];
```

```
		angles.theta3 = -joint_angles[3];
		angles.theta4 = -joint_angles[4];
		angles.theta5 = -joint_angles[5];
		angles.theta6 = joint_angles[6];
		angles.theta7 = joint_angles[7];
		angles.theta8 = joint_angles[8];
		angles.theta9 = -joint_angles[9];
		angles.theta10 = -joint_angles[10];
		angles.theta11 = -joint_angles[11];
		angles.theta12 = -joint_angles[12];
		angles.theta13 = joint_angles[13];
		angles.theta14 = joint_angles[14];
		angles.theta15 = joint_angles[15];
		angles.theta16 = joint_angles[16];
		angles.theta17 = joint_angles[17];
		angles.theta18 = joint_angles[18];
		angles.theta19 = joint_angles[19];
		angles.theta20 = joint_angles[20];
		angles.theta21 = joint_angles[21];
		angles.theta22 = joint_angles[22];


		// uncomment and use these angles for testing purposes
/*
		angles.theta0 = 30 * PI/180;
		angles.theta1 = 0;
		angles.theta2 = 49.47 * PI/180;
		angles.theta3 = 0;
		angles.theta4 = -30 * PI/180;
		angles.theta5 = 0;
		angles.theta6 = 19.47 * PI/180;
		angles.theta7 = 0;
		angles.theta8 = 0;
		angles.theta9 = 0;
		angles.theta10 = 0;
		angles.theta11 = 0;
		angles.theta12 = 0;
		angles.theta13 = 0;
		angles.theta14 = 0;
		angles.theta15 = 0;
		angles.theta16 = 0;
		angles.theta17 = 0;
		angles.theta18 = 0;
		angles.theta19 = 0;
		angles.theta20 = 0;
		angles.theta21 = 0;
		angles.theta22 = 0;
*/

#endif

	bal_state = get_support();

	calc_com(bal_state);

	balance_attitude(bal_state);

//	printf("COM: %f\t%f\t%f\n", com_total[0], com_total[1], com_total[2]);

}
```

```
/*
 * get_support
 *
 * This function determines the supporting base of the robot.
 * The base is either: LEFT_SUPPORT, BOTH_SUPPORT, or RIGHT_SUPPORT
 *
 */
int get_support (void) {

	// variables for the length of the legs
	double R_leg[4];
	double L_leg[4];
	double length_R_leg;
	double length_L_leg;
	int balance_state;

	// initialise other variables
	double zero_vector[] = {0, 0, 0, 1};

	// determine length of right leg
	vec_mult(R_leg, T_9origin, zero_vector);
	length_R_leg = sqrt(pow(R_leg[0], 2) + pow(R_leg[1], 2) + pow(R_leg[2], 2));


	// determine length of left leg
/*	vec_mult(tmp_vec, T_Lorigin, zero_vector);
	for (i=0; i<3; i++) {
		L_leg[i] = R_leg[i] - tmp_vec[i];
	}
*/


	// Make negative (due to errors in simulator): (0, 1, 2, 3, 4, 5, 6, 7)

	// Link L:  Left Foot
	trans(tmp1, -0.045, -0.025, 0.055);
	Rz(tmp2, PI);
	Ry(tmp3, PI/2);
	mat_mult3(tmp4, tmp1, tmp2, tmp3);

	// Link 1:  Left Ankle - Joint 1
	mdh(tmp1, -0.051, -PI/2, 0, -angles.theta1);
	mat_mult2(tmp2, tmp4, tmp1);

	// Link 0:  Left Lower Leg (Shin) - Joint 0
	mdh(tmp1, -0.172, PI, 0, -angles.theta0);
	mat_mult2(tmp3, tmp2, tmp1);

	// Link 2:  Left Upper Leg (Thigh) - Joint 2
	Rz(tmp1, -angles.theta2);
	trans(tmp2, -0.21, 0, 0);
	Ry(tmp4, PI/2);
	Rz(tmp5, -PI/2);
	mat_mult4(tmp6, tmp1, tmp2, tmp4, tmp5);
	mat_mult2(tmp1, tmp3, tmp6);

	// Link 8:  Left Lower Hip (Pitch) - Joint 8
	Rz(tmp5, -angles.theta8);
	trans(tmp2, 0, 0, -0.055);
	Rz(tmp3, -PI/2);
	Ry(tmp4, -PI/2);
	mat_mult4(tmp6, tmp5, tmp2, tmp3, tmp4);
	mat_mult2(tmp2, tmp1, tmp6);

	// Link 6:  Left Upper Hip (Roll) - Joint 6
	mdh(tmp3, -0.075, PI/2, 0, -angles.theta6);
	mat_mult2(tmp1, tmp2, tmp3);

	// Link 7:     Hip - Joint 7
/*	Rz(tmp2, -angles.theta7);
	trans(tmp3, -0.06, 0.125, 0);
	mat_mult4(tmp5, tmp2, tmp3, RyPI, RxnegPIdiv2);
	mat_mult2(tmp2, tmp1, tmp5);
*/

	vec_mult(L_leg, tmp1, zero_vector);

	length_L_leg = sqrt(pow(L_leg[0], 2) + pow(L_leg[1], 2) + pow(L_leg[2], 2));


	if (length_R_leg > (length_L_leg + BAL_THRESH)) {
		// robot on right foot
		balance_state = RIGHT_SUPPORT;
		printf("\nRight support.\n");

	} else if (length_L_leg > (length_R_leg + BAL_THRESH)) {
		// robot on left foot
```

```
                balance_state = LEFT_SUPPORT;
                printf("\nLeft support.\n");

        } else {
                // robot on both feet
                balance_state = BOTH_SUPPORT;
                printf("\nBoth support.\n");

        }

        printf("\nLeft leg: %f", length_L_leg);
        printf("  Right leg: %f\n\n", length_R_leg);

        return balance_state;
}
```

```
/*
 * calc_com
 *
 * This function calculates the centre of mass of the robot.
 *
 * It uses one argument which specifies what the supporting base is.
 * Can be either RIGHT_SUPPORT, LEFT_SUPPORT, BOTH_SUPPORT.
 *
 * The function returns a pointer to an array containing the
 * location of the centre of mass of the robot in 3D poloar coordinates.
 *
 */
double *calc_com (int balance_state) {


        // total mass
        double mT_legs = mR + m4 + m3 + m5+ m11 + m9 + m10 + m7 + m6 + m8 + m2 + m0 +
mL;
        double mT_torso = m13 + m14 + mT + m21 + m22 + m15 + m16 + m17 + m18 + m19 +
m20;


        // initialise other variables
        double zero_vector[] = {0, 0, 0, 1};


        // evaluate the commonly used transformation matrices (need to only do this
once .... should use #ifdef's etc)
        Rx(RxnegPIdiv2, -PI/2);
        Rx(RxPIdiv2, PI/2);
        Rx(RxPI, PI);

        Ry(RynegPIdiv2, -PI/2);
        Ry(RyPIdiv2, PI/2);
        Ry(RyPI, PI);

        Rz(RznegPIdiv2, -PI/2);
        Rz(RzPIdiv2, PI/2);
        Rz(RzPI, PI);


        // evaluate transformation matrices for each rigid body link

        /*
         * com for legs wrt right foot
         */

        // Link R:  Right Foot
        trans(tmp1, -0.045, 0.025, 0.055);
        mat_mult3(T_Rorigin, tmp1, RzPI, RyPIdiv2);
        vec_mult(comR, T_Rorigin, local_comR);

        // Link 4:  Right Ankle - Joint 4
        mdh(T_4wrtR, -0.051, -PI/2, 0, angles.theta4);
        mat_mult2(T_4origin, T_Rorigin, T_4wrtR);
        vec_mult(com4, T_4origin, local_com4);

        // Link 3:  Right Lower Leg (Shin) - Joint 3
        mdh(T_3wrt4, -0.172, PI, 0, angles.theta3);
        mat_mult2(T_3origin, T_4origin, T_3wrt4);
        vec_mult(com3, T_3origin, local_com3);

        // Link 5:  Right Upper Leg (Thigh) - Joint 5
        trans(tmp1, -0.21, 0, 0);
        Rz(tmp4, angles.theta5);
        mat_mult4(T_5wrt3, tmp4, tmp1, RyPIdiv2, RznegPIdiv2);
        mat_mult2(T_5origin, T_3origin, T_5wrt3);
        vec_mult(com5, T_5origin, local_com5);

        // Link 11:  Right Lower Hip (Pitch) - Joint 11
        trans(tmp1, 0, 0, -0.055);
        Rz(tmp4, angles.theta11);
        mat_mult4(T_11wrt5, tmp4, tmp1, RznegPIdiv2, RynegPIdiv2);
        mat_mult2(T_11origin, T_5origin, T_11wrt5);
        vec_mult(com11, T_11origin, local_com11);

        // Link 9:  Right Upper Hip (Roll) - Joint 9
        mdh(T_9wrt11, -0.075, -PI/2, 0, angles.theta9);
        mat_mult2(T_9origin, T_11origin, T_9wrt11);
        vec_mult(com9, T_9origin, local_com9);

        // Link 10:  Hip
        Rz(tmp1, angles.theta10);
        trans(tmp2, 0, 0.25, 0);
        mat_mult3(T_10wrt9, tmp1, tmp2, RxPI);
        mat_mult2(T_10origin, T_9origin, T_10wrt9);
        vec_mult(com10, T_10origin, local_com10);
        // Link 7: Left Upper Hip (Roll)
        mdh(T_7wrt10, 0.075, -PI/2, 0, angles.theta7);
        mat_mult2(T_7origin, T_10origin, T_7wrt10);
        vec_mult(com7, T_7origin, local_com7);

        // Link 6: Left Lower Hip (Pitch)
        Rz(tmp1, angles.theta6);
        trans(tmp2,0.055, 0, 0);
        mat_mult4(T_6wrt7, tmp1, tmp2, RzPIdiv2, RxPIdiv2);
        mat_mult2(T_6origin, T_7origin, T_6wrt7);
        vec_mult(com6, T_6origin, local_com6);

        // Link 8: Left Upper Leg (Yaw)
        Rz(tmp1, angles.theta8);
        trans(tmp2, 0, 0, 0.21);
        mat_mult4(T_8wrt6, tmp1, tmp2, RynegPIdiv2, RxPIdiv2);
        mat_mult2(T_8origin, T_6origin, T_8wrt6);
        vec_mult(com8, T_8origin, local_com8);

        // Link 2: Left Lower Leg
        mdh(T_2wrt8, 0.172, PI, 0, angles.theta2);
        mat_mult2(T_2origin, T_8origin, T_2wrt8);
        vec_mult(com2, T_2origin, local_com2);

        // Link 0: Left Ankle
        mdh(T_0wrt2, 0.051, PI/2, 0, angles.theta0);
        mat_mult2(T_0origin, T_2origin, T_0wrt2);
        vec_mult(com0, T_0origin, local_com0);

        // Link L: Left Foot
        Rz(tmp1, angles.theta1);
        trans(tmp2, 0.055, -0.025, -0.045);
        mat_mult4(T_Lwrt0, tmp1, tmp2, RynegPIdiv2, RzPI);
        mat_mult2(T_Lorigin, T_0origin, T_Lwrt0);
        vec_mult(comL, T_Lorigin, local_comL);


        /*
         * com for Torso wrt right foot
         */

        // Link 10:  Hip (mass is considered in leg calcs). Required here for
position and orientation of torso.
        Rz(tmp1, angles.theta10);
        trans(tmp2, -0.06, 0.125, 0);
        mat_mult4(tmp5, tmp1, tmp2, RzPI, RxnegPIdiv2);
        mat_mult2(T_13wrtOrgn, T_9origin, tmp5);

        // Link 13: Upper Hip
        Rz(tmp1, angles.theta12);
        trans(tmp2, 0.07, 0, 0);
        mat_mult4(tmp5, tmp1, tmp2, RxPIdiv2, RznegPIdiv2);
        mat_mult2(T_13origin, T_13wrtOrgn, tmp5);
        vec_mult(com13, T_13origin, local_com13);

        // Link 14: Torso Twist (yaw)
        Rz(tmp1, angles.theta13);
        trans(tmp2, 0, 0.229, 0);
        mat_mult4(T_14wrt13, tmp1, tmp2, RyPIdiv2, RxPIdiv2);
        mat_mult2(T_14origin, T_13origin, T_14wrt13);
        vec_mult(com14, T_14origin, local_com14);

        // Link T: Torso
        Rz(tmp1, angles.theta14);
        trans(tmp2, 0, 0, -0.067);
        mat_mult4(T_Twrt14, tmp1, tmp2, RynegPIdiv2, RxPI);
        mat_mult2(T_Torigin, T_14origin, T_Twrt14);
        vec_mult(comT, T_Torigin, local_comT);

        // Link 21: Neck
        Rx(tmp1, angles.theta21);
        trans(tmp4, -0.00, 0.00, 0.00);
        mat_mult4(T_21wrtT, tmp1, RzPIdiv2, RxPIdiv2, tmp4);
        mat_mult2(T_21origin, T_Torigin, T_21wrtT);
        vec_mult(com21, T_21origin, local_com21);

        // Link 22: Head
        trans(tmp1, 0, 0, -0.021);
        Rx(tmp2, angles.theta22);
        mat_mult4(T_22wrt21, tmp1, tmp2, RynegPIdiv2, RxPI);
        mat_mult2(T_22origin, T_21origin, T_22wrt21);
        vec_mult(com22, T_22origin, local_com22);

        // Link 15:  Left Shoulder
        trans(tmp1, 0.03, -0.21, 0);
        Rz(tmp3, angles.theta15);
        mat_mult3(T_15wrtT, tmp1, RxPIdiv2, tmp3);
        mat_mult2(T_15origin, T_Torigin, T_15wrtT);
        vec_mult(com15, T_15origin, local_com15);

        // Link 16: Left Upper Arm
        trans(tmp1, 0.00, 0.00, 0.024);
        Rz(tmp3, angles.theta16);
        mat_mult3(T_16wrt15, tmp1, RxPIdiv2, tmp3);
        mat_mult2(T_16origin, T_15origin, T_16wrt15);
        vec_mult(com16, T_16origin, local_com16);

        // Link 17: Left Lower Arm
        trans(tmp1, 0.209, 0.00, 0.00);
        Rz(tmp3, angles.theta17);
        mat_mult3(T_17wrt16, tmp1, RxnegPIdiv2, tmp3);
        mat_mult2(T_17origin, T_16origin, T_17wrt16);
        vec_mult(com17, T_17origin, local_com17);

        // Link 18: Right Shoulder
        trans(tmp1, 0.03, 0.21, 0);
        Rz(tmp3, angles.theta18);
        mat_mult3(T_18wrtT, tmp1, RxPIdiv2, tmp3);
        mat_mult2(T_18origin, T_Torigin, T_18wrtT);
        vec_mult(com18, T_18origin, local_com18);

        // Link 19: Right Upper Arm
        trans(tmp1, 0.00, 0.00, -0.024);
        Rz(tmp3, angles.theta19);
        mat_mult3(T_19wrt18, tmp1, RxnegPIdiv2, tmp3);
        mat_mult2(T_19origin, T_18origin, T_19wrt18);
        vec_mult(com19, T_19origin, local_com19);

        // Link 20: Right Lower Arm
        trans(tmp1, 0.209, 0.00, 0.00);
        Rz(tmp3, angles.theta20);
        mat_mult3(T_20wrt19, tmp1, RxPIdiv2, tmp3);
        mat_mult2(T_20origin, T_19origin, T_20wrt19);
        vec_mult(com20, T_20origin, local_com20);


        // Calculate com for the legs wrt right foot
        com_legs[0] = (mR*comR[0] + m4*com4[0] + m3*com3[0] + m5*com5[0] +
m11*com11[0] + m9*com9[0] + m10*com10[0] + m7*com7[0] + m6*com6[0] + m8*com8[0] +
m2*com2[0] + m0*com0[0] + mL*comL[0]) / mT_legs;
        com_legs[1] = (mR*comR[1] + m4*com4[1] + m3*com3[1] + m5*com5[1] +
m11*com11[1] + m9*com9[1] + m10*com10[1] + m7*com7[1] + m6*com6[1] + m8*com8[1] +
m2*com2[1] + m0*com0[1] + mL*comL[1]) / mT_legs;
        com_legs[2] = (mR*comR[2] + m4*com4[2] + m3*com3[2] + m5*com5[2] +
m11*com11[2] + m9*com9[2] + m10*com10[2] + m7*com7[2] + m6*com6[2] + m8*com8[2] +
m2*com2[2] + m0*com0[2] + mL*comL[2]) / mT_legs;

        // Calculate com for the torso wrt right foot
        com_torso[0] = (m13*com13[0] + m14*com14[0] + mT*comT[0] + m21*com21[0] +
m22*com22[0] + m15*com15[0] + m16*com16[0] + m17*com17[0] + m18*com18[0] +
m19*com19[0] + m20*com20[0]) / mT_torso;
        com_torso[1] = (m13*com13[1] + m14*com14[1] + mT*comT[1] + m21*com21[1] +
m22*com22[1] + m15*com15[1] + m16*com16[1] + m17*com17[1] + m18*com18[1] +
m19*com19[1] + m20*com20[1]) / mT_torso;
        com_torso[2] = (m13*com13[2] + m14*com14[2] + mT*comT[2] + m21*com21[2] +
m22*com22[2] + m15*com15[2] + m16*com16[2] + m17*com17[2] + m18*com18[2] +
m19*com19[2] + m20*com20[2]) / mT_torso;

        com_total[0] = (mT_legs*com_legs[0] + mT_torso*com_torso[0]) / (mT_legs +
mT_torso);
        com_total[1] = (mT_legs*com_legs[1] + mT_torso*com_torso[1]) / (mT_legs +
mT_torso);
        com_total[2] = (mT_legs*com_legs[2] + mT_torso*com_torso[2]) / (mT_legs +
mT_torso);


        // Calculate centre of mass for robot!
        if (balance_state == LEFT_SUPPORT) {
                vec_mult(LwrtR, T_Lorigin, zero_vector);
                com_total[0] = com_total[0] - LwrtR[0];
                com_total[1] = com_total[1] - LwrtR[1];
                com_total[2] = com_total[2] - LwrtR[2];

        } else if (balance_state == BOTH_SUPPORT) {
                vec_mult(LwrtR, T_Lorigin, zero_vector);
                com_total[0] = com_total[0] - (LwrtR[0] / 2.0);
                com_total[1] = com_total[1] - (LwrtR[1] / 2.0);
                com_total[2] = com_total[2] - (LwrtR[2] / 2.0);

        } else if (balance_state == RIGHT_SUPPORT) {
                // return the value calculated above

        } else {
```

```
		// error condition
		return error_ptr;
	}


	// Convert com location to 3D polar coordinates (angles.theta_roll,
angles.theta_pitch, r);
	state_com[0] = atan(com_total[1] / com_total[2]);
	state_com[1] = atan(com_total[2] / com_total[0]);
	state_com[2] = sqrt(pow(com_total[0],2) + pow(com_total[1],2) +
pow(com_total[2],2));


	// for printing result of matrix
/*	for (i = 0; i < 16; i++) {
		printf("%f\t", *(T_Rorigin + i));
		if (i==3) {
			printf("\n");
		} else if (i==7) {
			printf("\n");
		} else if (i==11) {
			printf("\n");
		}
	}
	printf("\n\n");
*/


	// print result of vector
/*	printf("Centre for the entire robot:\n");
	for (int i = 0; i < 3; i++) {
		printf("%f\n", com_total[i]);
	}
	printf("\n\n");
*/

	return state_com;
}



void balance_attitude (int balance_state) {

	double m[24] = {mR, m4, m3, m5, m11, m9, m10, m7, m6, m8, m2, m0, mL, m13,
m14, mT, m21, m22, m15, m16, m17, m18, m19, m20};

	// initialise variables
	double pos_com_x[] = {comR[0], com4[0], com3[0], com5[0], com11[0], com9[0],
com10[0], com7[0], com6[0], com8[0], com2[0], com0[0], comL[0], com13[0], com14[0],
comT[0], com21[0], com22[0], com15[0], com16[0], com17[0], com18[0], com19[0],
com20[0]};
	double pos_com_y[] = {comR[1], com4[1], com3[1], com5[1], com11[1], com9[1],
com10[1], com7[1], com6[1], com8[1], com2[1], com0[1], comL[1], com13[1], com14[1],
comT[1], com21[1], com22[1], com15[1], com16[1], com17[1], com18[1], com19[1],
com20[1]};
	double pos_com_z[] = {comR[2], com4[2], com3[2], com5[2], com11[2], com9[2],
com10[2], com7[2], com6[2], com8[2], com2[2], com0[2], comL[2], com13[2], com14[2],
comT[2], com21[2], com22[2], com15[2], com16[2], com17[2], com18[2], com19[2],
com20[2]};

	double vel_com_x[24];
	double vel_com_y[24];
	double vel_com_z[24];

	double acc_com_x[24];
	double acc_com_y[24];
	double acc_com_z[24];

	double tmp_var1 = 0.0;
	double tmp_var2 = 0.0;
	double tmp_var3 = 0.0;
	double tmp_var4 = 0.0;
	double tmp_var5 = 0.0;

	double zero_vector[] = {0, 0, 0, 1};

	double ZMPx, ZMPy, ZMPz;

	int i;

	if (balance_state == LEFT_SUPPORT) {
		vec_mult(LwrtR, T_Lorigin, zero_vector);
		for (i = 1; i<24; i++) {
			pos_com_x[i] = pos_com_x[i] - LwrtR[0];
			pos_com_y[i] = pos_com_y[i] - LwrtR[1];
			pos_com_z[i] = pos_com_z[i] - LwrtR[2];
		}
	} else if (balance_state == BOTH_SUPPORT) {
		vec_mult(LwrtR, T_Lorigin, zero_vector);
		for (i = 1; i<24; i++) {
			pos_com_x[i] = pos_com_x[i] - (LwrtR[0] / 2.0);
			pos_com_y[i] = pos_com_y[i] - (LwrtR[1] / 2.0);
			pos_com_z[i] = pos_com_z[i] - (LwrtR[2] / 2.0);
		}
	} else if (balance_state == RIGHT_SUPPORT) {
		// return the value calculated above

	} else {
		// error condition
	}


	// calculate the velocity and acceleration of the com for each link
	for (i = 0; i<24; i++) {

		// x direction
		vel_com_x[i] = (pos_com_x[i] - old_pos_com_x[i]) / CENTRAL_SPEED;
		acc_com_x[i] = (vel_com_x[i] - old_vel_com_x[i]) / CENTRAL_SPEED;

		// y direction
		vel_com_y[i] = (pos_com_y[i] - old_pos_com_y[i]) / CENTRAL_SPEED;
		acc_com_y[i] = (vel_com_y[i] - old_vel_com_y[i]) / CENTRAL_SPEED;

		// z direction
		vel_com_z[i] = (pos_com_z[i] - old_pos_com_z[i]) / CENTRAL_SPEED;
		acc_com_z[i] = (vel_com_z[i] - old_vel_com_z[i]) / CENTRAL_SPEED;


		// variables to be used in the calculation of the zmp
		tmp_var1 = tmp_var1 + m[i] * pos_com_x[i] * (acc_com_z[i] + g);
		tmp_var2 = tmp_var2 + m[i] * acc_com_x[i] * pos_com_z[i];
		tmp_var3 = tmp_var3 + m[i] * (acc_com_z[i] + g);

		tmp_var4 = tmp_var4 + m[i] * pos_com_y[i] * (acc_com_z[i] + g);
		tmp_var5 = tmp_var5 + m[i] * acc_com_y[i] * pos_com_z[i];
	}


	// calculate the location of the zero moment point (zmp)
	ZMPx = (tmp_var1 - tmp_var2) / tmp_var3;
	ZMPy = (tmp_var4 - tmp_var5) / tmp_var3;
	ZMPz = 0;


//	printf("ZMP: %f\t%f\t%f\n", ZMPx, ZMPy, ZMPz);
//	printf("CoM: %f\t%f\t%f\n", com_total[0], com_total[1], com_total[2]);

	// define movement for stability (specify an angle and a max acceleration?)

	// temporary variable used for data_logging
//	tmp_zmp_x = ZMPx;
//	tmp_zmp_y = ZMPy;
//	tmp_zmp_z = ZMPz;


	if (balance_state == LEFT_SUPPORT) {
		// pitch (x direction)
		if (ZMPx > ZMP_THRESH) {
			*desired_joint_vel[LEFT_ANKLE_FWD] =  -10;
			printf("pos\t\t");

		} else if (ZMPx < -ZMP_THRESH) {
			*desired_joint_vel[LEFT_ANKLE_FWD] = 10;
			printf("neg\t\t");

		} else {
			*desired_joint_vel[LEFT_ANKLE_FWD] = 0;
			printf("zero\t\t");
		}


		// roll (y direction)
		if (ZMPy > ZMP_THRESH) {
			*desired_joint_vel[LEFT_ANKLE_SIDE] = 10;
			printf("pos\n");

		} else if (ZMPy < -ZMP_THRESH) {
			*desired_joint_vel[LEFT_ANKLE_SIDE] =  -10;
			printf("neg\n");

		} else {
			*desired_joint_vel[LEFT_ANKLE_SIDE] = 0;
			printf("zero\n");
		}
```

```
        } else if (balance_state == RIGHT_SUPPORT) {

                // pitch (x direction)
                if (ZMPx > ZMP_THRESH) {
                        *desired_joint_vel[RIGHT_ANKLE_FWD] =  -10;
                        printf("pos\t\t");

                } else if (ZMPx < -ZMP_THRESH) {
                        *desired_joint_vel[RIGHT_ANKLE_FWD] = 10;
                        printf("neg\t\t");

                } else {
                        *desired_joint_vel[RIGHT_ANKLE_FWD] = 0;
                        printf("zero\t\t");
                }


                // roll (y direction)
                if (ZMPy > ZMP_THRESH) {
                        *desired_joint_vel[RIGHT_ANKLE_SIDE] = 10;
                        printf("pos\n");

                } else if (ZMPy < -ZMP_THRESH) {
                        *desired_joint_vel[RIGHT_ANKLE_SIDE] =  -10;
                        printf("neg\n");

                } else {
                        *desired_joint_vel[RIGHT_ANKLE_SIDE] = 0;
                        printf("zero\n");
                }

        } else if (balance_state == BOTH_SUPPORT) {

                // pitch (x direction)
                if (ZMPx > ZMP_THRESH) {
                        *desired_joint_vel[RIGHT_ANKLE_FWD] =  -10;
                        *desired_joint_vel[LEFT_ANKLE_FWD] =  -10;
                        printf("pos\t\t");

                } else if (ZMPx < -ZMP_THRESH) {
                        *desired_joint_vel[RIGHT_ANKLE_FWD] = 10;
                        *desired_joint_vel[LEFT_ANKLE_FWD] = 10;
                        printf("neg\t\t");

                } else {
                        *desired_joint_vel[RIGHT_ANKLE_FWD] = 0;
                        *desired_joint_vel[LEFT_ANKLE_FWD] = 0;
                        printf("zero\t\t");
                }


                // roll (y direction)
                if (ZMPy > ZMP_THRESH) {
                        *desired_joint_vel[RIGHT_ANKLE_SIDE] = -10;
                        *desired_joint_vel[LEFT_ANKLE_SIDE] = -10;
                        printf("pos\n");

                } else if (ZMPy < -ZMP_THRESH) {
                        *desired_joint_vel[RIGHT_ANKLE_SIDE] =  10;
                        *desired_joint_vel[LEFT_ANKLE_SIDE] =  10;
                        printf("neg\n");

                } else {
                        *desired_joint_vel[RIGHT_ANKLE_SIDE] = 0;
                        *desired_joint_vel[LEFT_ANKLE_SIDE] = 0;
                        printf("zero\n");
                }

        }


        // update variables
        for (i=0; i<24; i++) {
                old_pos_com_x[i] = pos_com_x[i];
                old_pos_com_y[i] = pos_com_y[i];

                old_vel_com_x[i] = vel_com_x[i];
                old_vel_com_y[i] = vel_com_y[i];
        }
}
```

```
/*
 * init_balance -
 *
 * This function initialises the balance system by allocating memory
 * for each matrix that is required.  It also does preliminary
 * calculations to determine the supporting base of the robot.
 *
 */
void init_balance(void) {

	int i;

	// initialise link transformation matrices
	T_RwrtOrgn = create_matrix();
	T_4wrtR = create_matrix();
	T_3wrt4 = create_matrix();
	T_5wrt3 = create_matrix();
	T_11wrt5 = create_matrix();
	T_9wrt11 = create_matrix();
	T_10wrt9 = create_matrix();
	T_7wrt10 = create_matrix();
	T_6wrt7 = create_matrix();
	T_8wrt6 = create_matrix();
	T_2wrt8 = create_matrix();
	T_0wrt2 = create_matrix();
	T_Lwrt0 = create_matrix();

	T_13wrtOrgn = create_matrix();
	T_14wrt13 = create_matrix();
	T_Twrt14 = create_matrix();
	T_21wrtT = create_matrix();
	T_22wrt21 = create_matrix();

	T_15wrtT = create_matrix();
	T_16wrt15 = create_matrix();
	T_17wrt16 = create_matrix();

	T_18wrtT = create_matrix();
	T_19wrt18 = create_matrix();
	T_20wrt19 = create_matrix();

	T_Rorigin = create_matrix();
	T_4origin = create_matrix();
	T_3origin = create_matrix();
	T_5origin = create_matrix();
	T_11origin = create_matrix();
	T_9origin = create_matrix();
	T_10origin = create_matrix();
	T_7origin = create_matrix();
	T_6origin = create_matrix();
	T_8origin = create_matrix();
	T_2origin = create_matrix();
	T_0origin = create_matrix();
	T_Lorigin = create_matrix();

	T_13origin = create_matrix();
	T_14origin = create_matrix();
	T_Torigin = create_matrix();
	T_21origin = create_matrix();
	T_22origin = create_matrix();
	T_15origin = create_matrix();
	T_16origin = create_matrix();
	T_17origin = create_matrix();
	T_18origin = create_matrix();
	T_19origin = create_matrix();
	T_20origin = create_matrix();

	// initialise arrays to store commonly used transformation matrices
	RxnegPIdiv2 = create_matrix();
	RxPIdiv2 = create_matrix();
	RxPI = create_matrix();

	RynegPIdiv2 = create_matrix();
	RyPIdiv2 = create_matrix();
	RyPI = create_matrix();

	RznegPIdiv2 = create_matrix();
	RzPIdiv2 = create_matrix();
	RzPI = create_matrix();

	tmp1 = create_matrix();
	tmp2 = create_matrix();
	tmp3 = create_matrix();
	tmp4 = create_matrix();
	tmp5 = create_matrix();
	tmp6 = create_matrix();
```

```
        for (i = 0; i<24; i++) {
                old_pos_com_x[i] = 0;
                old_pos_com_y[i] = 0;
                old_pos_com_z[i] = 0;

                old_vel_com_x[i] = 0;
                old_vel_com_y[i] = 0;
                old_vel_com_z[i] = 0;
        }

        // preliminary calculations required for get_support

        // Link R:  Right Foot
        trans(tmp1, -0.045, 0.025, 0.055);
        mat_mult3(T_Rorigin, tmp1, RzPI, RyPIdiv2);
        vec_mult(comR, T_Rorigin, local_comR);

        // Link 4:  Right Ankle - Joint 4
        mdh(T_4wrtR, -0.051, -PI/2, 0, angles.theta4);
        mat_mult2(T_4origin, T_Rorigin, T_4wrtR);
        vec_mult(com4, T_4origin, local_com4);

        // Link 3:  Right Lower Leg (Shin) - Joint 3
        mdh(T_3wrt4, -0.172, PI, 0, angles.theta3);
        mat_mult2(T_3origin, T_4origin, T_3wrt4);
        vec_mult(com3, T_3origin, local_com3);

        // Link 5:  Right Upper Leg (Thigh) - Joint 5
        trans(tmp1, -0.21, 0, 0);
        Rz(tmp4, angles.theta5);
        mat_mult4(T_5wrt3, tmp4, tmp1, RyPIdiv2, RznegPIdiv2);
        mat_mult2(T_5origin, T_3origin, T_5wrt3);
        vec_mult(com5, T_5origin, local_com5);

        // Link 11:  Right Lower Hip (Pitch) - Joint 11
        trans(tmp1, 0, 0, -0.055);
        Rz(tmp4, angles.theta11);
        mat_mult4(T_11wrt5, tmp4, tmp1, RznegPIdiv2, RynegPIdiv2);
        mat_mult2(T_11origin, T_5origin, T_11wrt5);
        vec_mult(com11, T_11origin, local_com11);

        // Link 9:  Right Upper Hip (Roll) - Joint 9
        mdh(T_9wrt11, -0.075, -PI/2, 0, angles.theta9);
        mat_mult2(T_9origin, T_11origin, T_9wrt11);
        vec_mult(com9, T_9origin, local_com9);

        // Link 10:  Hip
        Rz(tmp1, angles.theta10);
        trans(tmp2, -0.06, 0.125, 0);
        mat_mult4(tmp5, tmp1, tmp2, RzPI, RxnegPIdiv2);
        mat_mult2(T_13wrtOrgn, T_9origin, tmp5);

}
```

```
/*
 * create_matrix -
 *
 * This function allocates memory for the storage of a matrix
 *
 */
double **create_matrix(void)
{
       double **matrix;
       double *column;
       int m;

       matrix = (double **) malloc(sizeof(double *) * 4);

       for (m = 0; m < 4; m++) {
              column = (double *) malloc(sizeof(double) * 4);
              matrix[m] = column;
       }

       return matrix;
}
```

```
/*
 * unpack_joint_angles -
 *
 * This function retrieves data for actual positions from the
 * incoming bit-stream
 *
 */
void unpack_joint_angles (char* incoming) {

       angles.theta0 = ENC2RAD((incoming[0] << 8) + incoming[1]);
       angles.theta1 = ENC2RAD((incoming[2] << 8) + incoming[3]);
       angles.theta2 = ENC2RAD((incoming[4] << 8) + incoming[5]);
       angles.theta3 = ENC2RAD((incoming[6] << 8) + incoming[7]);
       angles.theta4 = ENC2RAD((incoming[8] << 8) + incoming[9]);
       angles.theta5 = ENC2RAD((incoming[10] << 8) + incoming[11]);
       angles.theta6 = ENC2RAD((incoming[12] << 8) + incoming[13]);
       angles.theta7 = ENC2RAD((incoming[14] << 8) + incoming[15]);
       angles.theta8 = ENC2RAD((incoming[16] << 8) + incoming[17]);
       angles.theta9 = ENC2RAD((incoming[18] << 8) + incoming[19]);
       angles.theta10 = ENC2RAD((incoming[20] << 8) + incoming[21]);
       angles.theta11 = ENC2RAD((incoming[22] << 8) + incoming[23]);
       angles.theta12 = ENC2RAD((incoming[24] << 8) + incoming[25]);
       angles.theta13 = ENC2RAD((incoming[26] << 8) + incoming[27]);
       angles.theta14 = ENC2RAD((incoming[28] << 8) + incoming[29]);
       angles.theta15 = ENC2RAD((incoming[30] << 8) + incoming[31]);
       angles.theta16 = ENC2RAD((incoming[32] << 8) + incoming[33]);
       angles.theta17 = ENC2RAD((incoming[34] << 8) + incoming[35]);
       angles.theta18 = ENC2RAD((incoming[36] << 8) + incoming[37]);
       angles.theta19 = ENC2RAD((incoming[38] << 8) + incoming[39]);
       angles.theta20 = ENC2RAD((incoming[40] << 8) + incoming[41]);
       angles.theta21 = ENC2RAD((incoming[42] << 8) + incoming[43]);
       angles.theta22 = ENC2RAD((incoming[44] << 8) + incoming[45]);

}
```

```
/*
 * end_balance -
 *
 * This function releases the memory allocated to each of the matrices
 *
 */
void end_balance (void) {

        // initialise link transformation matrices
        release_matrix(T_RwrtOrgn);
        release_matrix(T_4wrtR);
        release_matrix(T_3wrt4);
        release_matrix(T_5wrt3);
        release_matrix(T_11wrt5);
        release_matrix(T_9wrt11);
        release_matrix(T_10wrt9);
        release_matrix(T_7wrt10);
        release_matrix(T_6wrt7);
        release_matrix(T_8wrt6);
        release_matrix(T_2wrt8);
        release_matrix(T_0wrt2);
        release_matrix(T_Lwrt0);

        release_matrix(T_13wrtOrgn);
        release_matrix(T_14wrt13);
        release_matrix(T_Twrt14);
        release_matrix(T_21wrtT);
        release_matrix(T_22wrt21);

        release_matrix(T_15wrtT);
        release_matrix(T_16wrt15);
        release_matrix(T_17wrt16);

        release_matrix(T_18wrtT);
        release_matrix(T_19wrt18);
        release_matrix(T_20wrt19);

        release_matrix(T_Rorigin);
        release_matrix(T_4origin);
        release_matrix(T_3origin);
        release_matrix(T_5origin);
        release_matrix(T_11origin);
        release_matrix(T_9origin);
        release_matrix(T_10origin);
        release_matrix(T_7origin);
        release_matrix(T_6origin);
        release_matrix(T_8origin);
        release_matrix(T_2origin);
        release_matrix(T_0origin);
        release_matrix(T_Lorigin);

        release_matrix(T_13origin);
        release_matrix(T_14origin);
        release_matrix(T_Torigin);
        release_matrix(T_21origin);
        release_matrix(T_22origin);
        release_matrix(T_15origin);
        release_matrix(T_16origin);
        release_matrix(T_17origin);
        release_matrix(T_18origin);
        release_matrix(T_19origin);
        release_matrix(T_20origin);


        // initialise arrays to store commonly used transformation matrices
        release_matrix(RxnegPIdiv2);
        release_matrix(RxPIdiv2);
        release_matrix(RxPI);

        release_matrix(RynegPIdiv2);
        release_matrix(RyPIdiv2);
        release_matrix(RyPI);

        release_matrix(RznegPIdiv2);
        release_matrix(RzPIdiv2);
        release_matrix(RzPI);


        release_matrix(tmp1);
        release_matrix(tmp2);
        release_matrix(tmp3);
        release_matrix(tmp4);
        release_matrix(tmp5);
        release_matrix(tmp6);

}
```

```
/*
 * release_matrix -
 *
 * This function frees the memory a matrix
 *
 */
void release_matrix(double **matrix)
{
	int n;
	for (n = 0; n < 4; n++) {
		free(matrix[n]);
	}
	free(matrix);
}



/*
 * mdh -
 *
 * This function evaluates the general modified Denavit-Hartenberg
 * transformation matrix.
 *
 * Inputs are the mdh parameters: a, alpha, d, theta
 *
 */
void mdh (double **ptr_out, double a, double alpha, double d, double theta) {

	ptr_out[0][0] = cos(theta);				// mdh_data11
	ptr_out[0][1] = -sin(theta)*cos(alpha);		// mdh_data12
	ptr_out[0][2] = sin(theta)*sin(alpha);		// mdh_data13
	ptr_out[0][3] = a*cos(theta);			// mdh_data14

	ptr_out[1][0] = sin(theta);				// mdh_data21
	ptr_out[1][1] = cos(theta)*cos(alpha);		// mdh_data22
	ptr_out[1][2] = -cos(theta)*sin(alpha);		// mdh_data23
	ptr_out[1][3] = a*sin(theta);			// mdh_data24

	ptr_out[2][0] = 0;					// mdh_data31
	ptr_out[2][1] = sin(alpha);				// mdh_data32
	ptr_out[2][2] = cos(alpha);				// mdh_data33
	ptr_out[2][3] = d;					// mdh_data34

	ptr_out[3][0] = 0;					// mdh_data41
	ptr_out[3][1] = 0;					// mdh_data42
	ptr_out[3][2] = 0;					// mdh_data43
	ptr_out[3][3] = 1;					// mdh_data44

	return;

}


/*
 * trans -
 *
 * This function evaluates a general translation matrix.
 *
 * Input is displacement vector (x, y, z)
 *
 */
void trans (double **ptr_out, double x, double y, double z) {

	ptr_out[0][0] = 1;		// trans_data11
	ptr_out[0][1] = 0;		// trans_data12
	ptr_out[0][2] = 0;		// trans_data13
	ptr_out[0][3] = x;		// trans_data14

	ptr_out[1][0] = 0;		// trans_data21
	ptr_out[1][1] = 1;		// trans_data22
	ptr_out[1][2] = 0;		// trans_data23
	ptr_out[1][3] = y;		// trans_data24

	ptr_out[2][0] = 0;		// trans_data31
	ptr_out[2][1] = 0;		// trans_data32
	ptr_out[2][2] = 1;		// trans_data33
	ptr_out[2][3] = z;		// trans_data34

	ptr_out[3][0] = 0;		// trans_data41
	ptr_out[3][1] = 0;		// trans_data42
	ptr_out[3][2] = 0;		// trans_data43
	ptr_out[3][3] = 1;		// trans_data44

	return;

}
```

```
/*
 * Rx -
 *
 * This function creates a rotational transformation matrix about the x-axis.
 *
 * Input is an angle in radians.
 *
 */
void Rx (double **ptr_out, double theta) {

       ptr_out[0][0] = 1;                        // Rx_data11
       ptr_out[0][1] = 0;                        // Rx_data12
       ptr_out[0][2] = 0;                        // Rx_data13
       ptr_out[0][3] = 0;                        // Rx_data14

       ptr_out[1][0] = 0;                        // Rx_data21
       ptr_out[1][1] = cos(theta);        // Rx_data22
       ptr_out[1][2] = -sin(theta);  // Rx_data23
       ptr_out[1][3] = 0;                        // Rx_data24

       ptr_out[2][0] = 0;                        // Rx_data31
       ptr_out[2][1] = sin(theta);        // Rx_data32
       ptr_out[2][2] = cos(theta);        // Rx_data33
       ptr_out[2][3] = 0;                        // Rx_data34

       ptr_out[3][0] = 0;                        // Rx_data41
       ptr_out[3][1] = 0;                        // Rx_data42
       ptr_out[3][2] = 0;                        // Rx_data43
       ptr_out[3][3] = 1;                        // Rx_data44

       return;

}
```

```
/*
 * Ry -
 *
 * This function creates a rotational transformation matrix about the y-axis.
 *
 * Input is an angle in radians.
 *
 */
void Ry (double **ptr_out, double theta) {

       ptr_out[0][0] = cos(theta);
       ptr_out[0][1] = 0;                        // Ry_data12
       ptr_out[0][2] = sin(theta);        // Ry_data13
       ptr_out[0][3] = 0;                        // Ry_data14

       ptr_out[1][0] = 0;                        // Ry_data21
       ptr_out[1][1] = 1;                        // Ry_data22
       ptr_out[1][2] = 0;                        // Ry_data23
       ptr_out[1][3] = 0;                        // Ry_data24

       ptr_out[2][0] = -sin(theta);  // Ry_data31
       ptr_out[2][1] = 0;                        // Ry_data32
       ptr_out[2][2] = cos(theta);        // Ry_data33
       ptr_out[2][3] = 0;                        // Ry_data34

       ptr_out[3][0] = 0;                        // Ry_data41
       ptr_out[3][1] = 0;                        // Ry_data42
       ptr_out[3][2] = 0;                        // Ry_data43
       ptr_out[3][3] = 1;                        // Ry_data44

       return;
}
```

```
/*
 * Rz -
 *
 * This function creates a rotational transformation matrix about the z-axis.
 *
 * Input is an angle in radians.
 *
 */
void Rz (double **ptr_out, double theta) {

        ptr_out[0][0] = cos(theta);           // Rz_data11
        ptr_out[0][1] = -sin(theta);  // Rz_data12
        ptr_out[0][2] = 0;                           // Rz_data13
        ptr_out[0][3] = 0;                           // Rz_data14

        ptr_out[1][0] = sin(theta);           // Rz_data21
        ptr_out[1][1] = cos(theta);           // Rz_data22
        ptr_out[1][2] = 0;                           // Rz_data23
        ptr_out[1][3] = 0;                           // Rz_data24

        ptr_out[2][0] = 0;                           // Rz_data31
        ptr_out[2][1] = 0;                           // Rz_data32
        ptr_out[2][2] = 1;                           // Rz_data33
        ptr_out[2][3] = 0;                           // Rz_data34

        ptr_out[3][0] = 0;                           // Rz_data41
        ptr_out[3][1] = 0;                           // Rz_data42
        ptr_out[3][2] = 0;                           // Rz_data43
        ptr_out[3][3] = 1;                           // Rz_data44

        return;

}



/*
 * mat_mult2 -
 *
 * This function multiplies 4 x 4 matrices.
 *
 *      OUTPUT_MATRIX = MATRIX_A x MATRIX_B
 *
 */
void mat_mult2 (double **ptr_out, double **ptr_a, double **ptr_b) {

        ptr_out[0][0] = ptr_a[0][0]*ptr_b[0][0] + ptr_a[0][1]*ptr_b[1][0] +
ptr_a[0][2]*ptr_b[2][0] + ptr_a[0][3]*ptr_b[3][0];
        ptr_out[0][1] = ptr_a[0][0]*ptr_b[0][1] + ptr_a[0][1]*ptr_b[1][1] +
ptr_a[0][2]*ptr_b[2][1] + ptr_a[0][3]*ptr_b[3][1];
        ptr_out[0][2] = ptr_a[0][0]*ptr_b[0][2] + ptr_a[0][1]*ptr_b[1][2] +
ptr_a[0][2]*ptr_b[2][2] + ptr_a[0][3]*ptr_b[3][2];
        ptr_out[0][3] = ptr_a[0][0]*ptr_b[0][3] + ptr_a[0][1]*ptr_b[1][3] +
ptr_a[0][2]*ptr_b[2][3] + ptr_a[0][3]*ptr_b[3][3];

        ptr_out[1][0] = ptr_a[1][0]*ptr_b[0][0] + ptr_a[1][1]*ptr_b[1][0] +
ptr_a[1][2]*ptr_b[2][0] + ptr_a[1][3]*ptr_b[3][0];
        ptr_out[1][1] = ptr_a[1][0]*ptr_b[0][1] + ptr_a[1][1]*ptr_b[1][1] +
ptr_a[1][2]*ptr_b[2][1] + ptr_a[1][3]*ptr_b[3][1];
        ptr_out[1][2] = ptr_a[1][0]*ptr_b[0][2] + ptr_a[1][1]*ptr_b[1][2] +
ptr_a[1][2]*ptr_b[2][2] + ptr_a[1][3]*ptr_b[3][2];
        ptr_out[1][3] = ptr_a[1][0]*ptr_b[0][3] + ptr_a[1][1]*ptr_b[1][3] +
ptr_a[1][2]*ptr_b[2][3] + ptr_a[1][3]*ptr_b[3][3];

        ptr_out[2][0] = ptr_a[2][0]*ptr_b[0][0] + ptr_a[2][1]*ptr_b[1][0] +
ptr_a[2][2]*ptr_b[2][0] + ptr_a[2][3]*ptr_b[3][0];
        ptr_out[2][1] = ptr_a[2][0]*ptr_b[0][1] + ptr_a[2][1]*ptr_b[1][1] +
ptr_a[2][2]*ptr_b[2][1] + ptr_a[2][3]*ptr_b[3][1];
        ptr_out[2][2] = ptr_a[2][0]*ptr_b[0][2] + ptr_a[2][1]*ptr_b[1][2] +
ptr_a[2][2]*ptr_b[2][2] + ptr_a[2][3]*ptr_b[3][2];
        ptr_out[2][3] = ptr_a[2][0]*ptr_b[0][3] + ptr_a[2][1]*ptr_b[1][3] +
ptr_a[2][2]*ptr_b[2][3] + ptr_a[2][3]*ptr_b[3][3];

        ptr_out[3][0] = ptr_a[3][0]*ptr_b[0][0] + ptr_a[3][1]*ptr_b[1][0] +
ptr_a[3][2]*ptr_b[2][0] + ptr_a[3][3]*ptr_b[3][0];
        ptr_out[3][1] = ptr_a[3][0]*ptr_b[0][1] + ptr_a[3][1]*ptr_b[1][1] +
ptr_a[3][2]*ptr_b[2][1] + ptr_a[3][3]*ptr_b[3][1];
        ptr_out[3][2] = ptr_a[3][0]*ptr_b[0][2] + ptr_a[3][1]*ptr_b[1][2] +
ptr_a[3][2]*ptr_b[2][2] + ptr_a[3][3]*ptr_b[3][2];
        ptr_out[3][3] = ptr_a[3][0]*ptr_b[0][3] + ptr_a[3][1]*ptr_b[1][3] +
ptr_a[3][2]*ptr_b[2][3] + ptr_a[3][3]*ptr_b[3][3];

        return;
}
```

```
/*
 * mat_mult3 -
 *
 * This function multiplies 4 x 4 matrices.
 *
 *      OUTPUT_MATRIX = MATRIX_A x MATRIX_B x MATRIX_C
 */
void mat_mult3 (double **ptr_out, double **ptr_a, double **ptr_b, double **ptr_c) {
        double ptr_tmp[4][4];

        // MATRIX_A x MATRIX_B
        ptr_tmp[0][0] = ptr_a[0][0]*ptr_b[0][0] + ptr_a[0][1]*ptr_b[1][0] +
ptr_a[0][2]*ptr_b[2][0] + ptr_a[0][3]*ptr_b[3][0];
        ptr_tmp[0][1] = ptr_a[0][0]*ptr_b[0][1] + ptr_a[0][1]*ptr_b[1][1] +
ptr_a[0][2]*ptr_b[2][1] + ptr_a[0][3]*ptr_b[3][1];
        ptr_tmp[0][2] = ptr_a[0][0]*ptr_b[0][2] + ptr_a[0][1]*ptr_b[1][2] +
ptr_a[0][2]*ptr_b[2][2] + ptr_a[0][3]*ptr_b[3][2];
        ptr_tmp[0][3] = ptr_a[0][0]*ptr_b[0][3] + ptr_a[0][1]*ptr_b[1][3] +
ptr_a[0][2]*ptr_b[2][3] + ptr_a[0][3]*ptr_b[3][3];

        ptr_tmp[1][0] = ptr_a[1][0]*ptr_b[0][0] + ptr_a[1][1]*ptr_b[1][0] +
ptr_a[1][2]*ptr_b[2][0] + ptr_a[1][3]*ptr_b[3][0];
        ptr_tmp[1][1] = ptr_a[1][0]*ptr_b[0][1] + ptr_a[1][1]*ptr_b[1][1] +
ptr_a[1][2]*ptr_b[2][1] + ptr_a[1][3]*ptr_b[3][1];
        ptr_tmp[1][2] = ptr_a[1][0]*ptr_b[0][2] + ptr_a[1][1]*ptr_b[1][2] +
ptr_a[1][2]*ptr_b[2][2] + ptr_a[1][3]*ptr_b[3][2];
        ptr_tmp[1][3] = ptr_a[1][0]*ptr_b[0][3] + ptr_a[1][1]*ptr_b[1][3] +
ptr_a[1][2]*ptr_b[2][3] + ptr_a[1][3]*ptr_b[3][3];

        ptr_tmp[2][0] = ptr_a[2][0]*ptr_b[0][0] + ptr_a[2][1]*ptr_b[1][0] +
ptr_a[2][2]*ptr_b[2][0] + ptr_a[2][3]*ptr_b[3][0];
        ptr_tmp[2][1] = ptr_a[2][0]*ptr_b[0][1] + ptr_a[2][1]*ptr_b[1][1] +
ptr_a[2][2]*ptr_b[2][1] + ptr_a[2][3]*ptr_b[3][1];
        ptr_tmp[2][2] = ptr_a[2][0]*ptr_b[0][2] + ptr_a[2][1]*ptr_b[1][2] +
ptr_a[2][2]*ptr_b[2][2] + ptr_a[2][3]*ptr_b[3][2];
        ptr_tmp[2][3] = ptr_a[2][0]*ptr_b[0][3] + ptr_a[2][1]*ptr_b[1][3] +
ptr_a[2][2]*ptr_b[2][3] + ptr_a[2][3]*ptr_b[3][3];

        ptr_tmp[3][0] = ptr_a[3][0]*ptr_b[0][0] + ptr_a[3][1]*ptr_b[1][0] +
ptr_a[3][2]*ptr_b[2][0] + ptr_a[3][3]*ptr_b[3][0];
        ptr_tmp[3][1] = ptr_a[3][0]*ptr_b[0][1] + ptr_a[3][1]*ptr_b[1][1] +
ptr_a[3][2]*ptr_b[2][1] + ptr_a[3][3]*ptr_b[3][1];
        ptr_tmp[3][2] = ptr_a[3][0]*ptr_b[0][2] + ptr_a[3][1]*ptr_b[1][2] +
ptr_a[3][2]*ptr_b[2][2] + ptr_a[3][3]*ptr_b[3][2];
        ptr_tmp[3][3] = ptr_a[3][0]*ptr_b[0][3] + ptr_a[3][1]*ptr_b[1][3] +
ptr_a[3][2]*ptr_b[2][3] + ptr_a[3][3]*ptr_b[3][3];


        // MATRIX_A x MATRIX_B x MATRIX_C
        ptr_out[0][0] = ptr_tmp[0][0]*ptr_c[0][0] + ptr_tmp[0][1]*ptr_c[1][0] +
ptr_tmp[0][2]*ptr_c[2][0] + ptr_tmp[0][3]*ptr_c[3][0];
        ptr_out[0][1] = ptr_tmp[0][0]*ptr_c[0][1] + ptr_tmp[0][1]*ptr_c[1][1] +
ptr_tmp[0][2]*ptr_c[2][1] + ptr_tmp[0][3]*ptr_c[3][1];
        ptr_out[0][2] = ptr_tmp[0][0]*ptr_c[0][2] + ptr_tmp[0][1]*ptr_c[1][2] +
ptr_tmp[0][2]*ptr_c[2][2] + ptr_tmp[0][3]*ptr_c[3][2];
        ptr_out[0][3] = ptr_tmp[0][0]*ptr_c[0][3] + ptr_tmp[0][1]*ptr_c[1][3] +
ptr_tmp[0][2]*ptr_c[2][3] + ptr_tmp[0][3]*ptr_c[3][3];

        ptr_out[1][0] = ptr_tmp[1][0]*ptr_c[0][0] + ptr_tmp[1][1]*ptr_c[1][0] +
ptr_tmp[1][2]*ptr_c[2][0] + ptr_tmp[1][3]*ptr_c[3][0];
        ptr_out[1][1] = ptr_tmp[1][0]*ptr_c[0][1] + ptr_tmp[1][1]*ptr_c[1][1] +
ptr_tmp[1][2]*ptr_c[2][1] + ptr_tmp[1][3]*ptr_c[3][1];
        ptr_out[1][2] = ptr_tmp[1][0]*ptr_c[0][2] + ptr_tmp[1][1]*ptr_c[1][2] +
ptr_tmp[1][2]*ptr_c[2][2] + ptr_tmp[1][3]*ptr_c[3][2];
        ptr_out[1][3] = ptr_tmp[1][0]*ptr_c[0][3] + ptr_tmp[1][1]*ptr_c[1][3] +
ptr_tmp[1][2]*ptr_c[2][3] + ptr_tmp[1][3]*ptr_c[3][3];

        ptr_out[2][0] = ptr_tmp[2][0]*ptr_c[0][0] + ptr_tmp[2][1]*ptr_c[1][0] +
ptr_tmp[2][2]*ptr_c[2][0] + ptr_tmp[2][3]*ptr_c[3][0];
        ptr_out[2][1] = ptr_tmp[2][0]*ptr_c[0][1] + ptr_tmp[2][1]*ptr_c[1][1] +
ptr_tmp[2][2]*ptr_c[2][1] + ptr_tmp[2][3]*ptr_c[3][1];
        ptr_out[2][2] = ptr_tmp[2][0]*ptr_c[0][2] + ptr_tmp[2][1]*ptr_c[1][2] +
ptr_tmp[2][2]*ptr_c[2][2] + ptr_tmp[2][3]*ptr_c[3][2];
        ptr_out[2][3] = ptr_tmp[2][0]*ptr_c[0][3] + ptr_tmp[2][1]*ptr_c[1][3] +
ptr_tmp[2][2]*ptr_c[2][3] + ptr_tmp[2][3]*ptr_c[3][3];

        ptr_out[3][0] = ptr_tmp[3][0]*ptr_c[0][0] + ptr_tmp[3][1]*ptr_c[1][0] +
ptr_tmp[3][2]*ptr_c[2][0] + ptr_tmp[3][3]*ptr_c[3][0];
        ptr_out[3][1] = ptr_tmp[3][0]*ptr_c[0][1] + ptr_tmp[3][1]*ptr_c[1][1] +
ptr_tmp[3][2]*ptr_c[2][1] + ptr_tmp[3][3]*ptr_c[3][1];
        ptr_out[3][2] = ptr_tmp[3][0]*ptr_c[0][2] + ptr_tmp[3][1]*ptr_c[1][2] +
ptr_tmp[3][2]*ptr_c[2][2] + ptr_tmp[3][3]*ptr_c[3][2];
        ptr_out[3][3] = ptr_tmp[3][0]*ptr_c[0][3] + ptr_tmp[3][1]*ptr_c[1][3] +
ptr_tmp[3][2]*ptr_c[2][3] + ptr_tmp[3][3]*ptr_c[3][3];

        return;
}
```

```
/*
 * mat_mult4 -
 *
 * This function multiplies 4 x 4 matrices.
 *
 *      OUTPUT_MATRIX = MATRIX_A x MATRIX_B x MATRIX_C x MATRIX_D
 *
 */
void mat_mult4 (double **ptr_out, double **ptr_a, double **ptr_b, double **ptr_c,
double **ptr_d) {

        double ptr_tmp[4][4];
        double ptr_tmp1[4][4];

        // MATRIX_A x MATRIX_B = MATRIX_TMP
        ptr_tmp[0][0] = ptr_a[0][0]*ptr_b[0][0] + ptr_a[0][1]*ptr_b[1][0] +
ptr_a[0][2]*ptr_b[2][0] + ptr_a[0][3]*ptr_b[3][0];
        ptr_tmp[0][1] = ptr_a[0][0]*ptr_b[0][1] + ptr_a[0][1]*ptr_b[1][1] +
ptr_a[0][2]*ptr_b[2][1] + ptr_a[0][3]*ptr_b[3][1];
        ptr_tmp[0][2] = ptr_a[0][0]*ptr_b[0][2] + ptr_a[0][1]*ptr_b[1][2] +
ptr_a[0][2]*ptr_b[2][2] + ptr_a[0][3]*ptr_b[3][2];
        ptr_tmp[0][3] = ptr_a[0][0]*ptr_b[0][3] + ptr_a[0][1]*ptr_b[1][3] +
ptr_a[0][2]*ptr_b[2][3] + ptr_a[0][3]*ptr_b[3][3];

        ptr_tmp[1][0] = ptr_a[1][0]*ptr_b[0][0] + ptr_a[1][1]*ptr_b[1][0] +
ptr_a[1][2]*ptr_b[2][0] + ptr_a[1][3]*ptr_b[3][0];
        ptr_tmp[1][1] = ptr_a[1][0]*ptr_b[0][1] + ptr_a[1][1]*ptr_b[1][1] +
ptr_a[1][2]*ptr_b[2][1] + ptr_a[1][3]*ptr_b[3][1];
        ptr_tmp[1][2] = ptr_a[1][0]*ptr_b[0][2] + ptr_a[1][1]*ptr_b[1][2] +
ptr_a[1][2]*ptr_b[2][2] + ptr_a[1][3]*ptr_b[3][2];
        ptr_tmp[1][3] = ptr_a[1][0]*ptr_b[0][3] + ptr_a[1][1]*ptr_b[1][3] +
ptr_a[1][2]*ptr_b[2][3] + ptr_a[1][3]*ptr_b[3][3];

        ptr_tmp[2][0] = ptr_a[2][0]*ptr_b[0][0] + ptr_a[2][1]*ptr_b[1][0] +
ptr_a[2][2]*ptr_b[2][0] + ptr_a[2][3]*ptr_b[3][0];
        ptr_tmp[2][1] = ptr_a[2][0]*ptr_b[0][1] + ptr_a[2][1]*ptr_b[1][1] +
ptr_a[2][2]*ptr_b[2][1] + ptr_a[2][3]*ptr_b[3][1];
        ptr_tmp[2][2] = ptr_a[2][0]*ptr_b[0][2] + ptr_a[2][1]*ptr_b[1][2] +
ptr_a[2][2]*ptr_b[2][2] + ptr_a[2][3]*ptr_b[3][2];
        ptr_tmp[2][3] = ptr_a[2][0]*ptr_b[0][3] + ptr_a[2][1]*ptr_b[1][3] +
ptr_a[2][2]*ptr_b[2][3] + ptr_a[2][3]*ptr_b[3][3];

        ptr_tmp[3][0] = ptr_a[3][0]*ptr_b[0][0] + ptr_a[3][1]*ptr_b[1][0] +
ptr_a[3][2]*ptr_b[2][0] + ptr_a[3][3]*ptr_b[3][0];
        ptr_tmp[3][1] = ptr_a[3][0]*ptr_b[0][1] + ptr_a[3][1]*ptr_b[1][1] +
ptr_a[3][2]*ptr_b[2][1] + ptr_a[3][3]*ptr_b[3][1];
        ptr_tmp[3][2] = ptr_a[3][0]*ptr_b[0][2] + ptr_a[3][1]*ptr_b[1][2] +
ptr_a[3][2]*ptr_b[2][2] + ptr_a[3][3]*ptr_b[3][2];
        ptr_tmp[3][3] = ptr_a[3][0]*ptr_b[0][3] + ptr_a[3][1]*ptr_b[1][3] +
ptr_a[3][2]*ptr_b[2][3] + ptr_a[3][3]*ptr_b[3][3];


        // MATRIX_A x MATRIX_B x MATRIX_C = MATRIX_TMP1
        ptr_tmp1[0][0] = ptr_tmp[0][0]*ptr_c[0][0] + ptr_tmp[0][1]*ptr_c[1][0] +
ptr_tmp[0][2]*ptr_c[2][0] + ptr_tmp[0][3]*ptr_c[3][0];
        ptr_tmp1[0][1] = ptr_tmp[0][0]*ptr_c[0][1] + ptr_tmp[0][1]*ptr_c[1][1] +
ptr_tmp[0][2]*ptr_c[2][1] + ptr_tmp[0][3]*ptr_c[3][1];
        ptr_tmp1[0][2] = ptr_tmp[0][0]*ptr_c[0][2] + ptr_tmp[0][1]*ptr_c[1][2] +
ptr_tmp[0][2]*ptr_c[2][2] + ptr_tmp[0][3]*ptr_c[3][2];
        ptr_tmp1[0][3] = ptr_tmp[0][0]*ptr_c[0][3] + ptr_tmp[0][1]*ptr_c[1][3] +
ptr_tmp[0][2]*ptr_c[2][3] + ptr_tmp[0][3]*ptr_c[3][3];

        ptr_tmp1[1][0] = ptr_tmp[1][0]*ptr_c[0][0] + ptr_tmp[1][1]*ptr_c[1][0] +
ptr_tmp[1][2]*ptr_c[2][0] + ptr_tmp[1][3]*ptr_c[3][0];
        ptr_tmp1[1][1] = ptr_tmp[1][0]*ptr_c[0][1] + ptr_tmp[1][1]*ptr_c[1][1] +
ptr_tmp[1][2]*ptr_c[2][1] + ptr_tmp[1][3]*ptr_c[3][1];
        ptr_tmp1[1][2] = ptr_tmp[1][0]*ptr_c[0][2] + ptr_tmp[1][1]*ptr_c[1][2] +
ptr_tmp[1][2]*ptr_c[2][2] + ptr_tmp[1][3]*ptr_c[3][2];
        ptr_tmp1[1][3] = ptr_tmp[1][0]*ptr_c[0][3] + ptr_tmp[1][1]*ptr_c[1][3] +
ptr_tmp[1][2]*ptr_c[2][3] + ptr_tmp[1][3]*ptr_c[3][3];

        ptr_tmp1[2][0] = ptr_tmp[2][0]*ptr_c[0][0] + ptr_tmp[2][1]*ptr_c[1][0] +
ptr_tmp[2][2]*ptr_c[2][0] + ptr_tmp[2][3]*ptr_c[3][0];
        ptr_tmp1[2][1] = ptr_tmp[2][0]*ptr_c[0][1] + ptr_tmp[2][1]*ptr_c[1][1] +
ptr_tmp[2][2]*ptr_c[2][1] + ptr_tmp[2][3]*ptr_c[3][1];
        ptr_tmp1[2][2] = ptr_tmp[2][0]*ptr_c[0][2] + ptr_tmp[2][1]*ptr_c[1][2] +
ptr_tmp[2][2]*ptr_c[2][2] + ptr_tmp[2][3]*ptr_c[3][2];
        ptr_tmp1[2][3] = ptr_tmp[2][0]*ptr_c[0][3] + ptr_tmp[2][1]*ptr_c[1][3] +
ptr_tmp[2][2]*ptr_c[2][3] + ptr_tmp[2][3]*ptr_c[3][3];

        ptr_tmp1[3][0] = ptr_tmp[3][0]*ptr_c[0][0] + ptr_tmp[3][1]*ptr_c[1][0] +
ptr_tmp[3][2]*ptr_c[2][0] + ptr_tmp[3][3]*ptr_c[3][0];
        ptr_tmp1[3][1] = ptr_tmp[3][0]*ptr_c[0][1] + ptr_tmp[3][1]*ptr_c[1][1] +
ptr_tmp[3][2]*ptr_c[2][1] + ptr_tmp[3][3]*ptr_c[3][1];
        ptr_tmp1[3][2] = ptr_tmp[3][0]*ptr_c[0][2] + ptr_tmp[3][1]*ptr_c[1][2] +
ptr_tmp[3][2]*ptr_c[2][2] + ptr_tmp[3][3]*ptr_c[3][2];
```

```
        ptr_tmp1[3][3] = ptr_tmp[3][0]*ptr_c[0][3] + ptr_tmp[3][1]*ptr_c[1][3] +
ptr_tmp[3][2]*ptr_c[2][3] + ptr_tmp[3][3]*ptr_c[3][3];


        // MATRIX_A x MATRIX_B x MATRIX_C x MATRIX_D = MATRIX_OUT
        ptr_out[0][0] = ptr_tmp1[0][0]*ptr_d[0][0] + ptr_tmp1[0][1]*ptr_d[1][0] +
ptr_tmp1[0][2]*ptr_d[2][0] + ptr_tmp1[0][3]*ptr_d[3][0];
        ptr_out[0][1] = ptr_tmp1[0][0]*ptr_d[0][1] + ptr_tmp1[0][1]*ptr_d[1][1] +
ptr_tmp1[0][2]*ptr_d[2][1] + ptr_tmp1[0][3]*ptr_d[3][1];
        ptr_out[0][2] = ptr_tmp1[0][0]*ptr_d[0][2] + ptr_tmp1[0][1]*ptr_d[1][2] +
ptr_tmp1[0][2]*ptr_d[2][2] + ptr_tmp1[0][3]*ptr_d[3][2];
        ptr_out[0][3] = ptr_tmp1[0][0]*ptr_d[0][3] + ptr_tmp1[0][1]*ptr_d[1][3] +
ptr_tmp1[0][2]*ptr_d[2][3] + ptr_tmp1[0][3]*ptr_d[3][3];

        ptr_out[1][0] = ptr_tmp1[1][0]*ptr_d[0][0] + ptr_tmp1[1][1]*ptr_d[1][0] +
ptr_tmp1[1][2]*ptr_d[2][0] + ptr_tmp1[1][3]*ptr_d[3][0];
        ptr_out[1][1] = ptr_tmp1[1][0]*ptr_d[0][1] + ptr_tmp1[1][1]*ptr_d[1][1] +
ptr_tmp1[1][2]*ptr_d[2][1] + ptr_tmp1[1][3]*ptr_d[3][1];
        ptr_out[1][2] = ptr_tmp1[1][0]*ptr_d[0][2] + ptr_tmp1[1][1]*ptr_d[1][2] +
ptr_tmp1[1][2]*ptr_d[2][2] + ptr_tmp1[1][3]*ptr_d[3][2];
        ptr_out[1][3] = ptr_tmp1[1][0]*ptr_d[0][3] + ptr_tmp1[1][1]*ptr_d[1][3] +
ptr_tmp1[1][2]*ptr_d[2][3] + ptr_tmp1[1][3]*ptr_d[3][3];

        ptr_out[2][0] = ptr_tmp1[2][0]*ptr_d[0][0] + ptr_tmp1[2][1]*ptr_d[1][0] +
ptr_tmp1[2][2]*ptr_d[2][0] + ptr_tmp1[2][3]*ptr_d[3][0];
        ptr_out[2][1] = ptr_tmp1[2][0]*ptr_d[0][1] + ptr_tmp1[2][1]*ptr_d[1][1] +
ptr_tmp1[2][2]*ptr_d[2][1] + ptr_tmp1[2][3]*ptr_d[3][1];
        ptr_out[2][2] = ptr_tmp1[2][0]*ptr_d[0][2] + ptr_tmp1[2][1]*ptr_d[1][2] +
ptr_tmp1[2][2]*ptr_d[2][2] + ptr_tmp1[2][3]*ptr_d[3][2];
        ptr_out[2][3] = ptr_tmp1[2][0]*ptr_d[0][3] + ptr_tmp1[2][1]*ptr_d[1][3] +
ptr_tmp1[2][2]*ptr_d[2][3] + ptr_tmp1[2][3]*ptr_d[3][3];

        ptr_out[3][0] = ptr_tmp1[3][0]*ptr_d[0][0] + ptr_tmp1[3][1]*ptr_d[1][0] +
ptr_tmp1[3][2]*ptr_d[2][0] + ptr_tmp1[3][3]*ptr_d[3][0];
        ptr_out[3][1] = ptr_tmp1[3][0]*ptr_d[0][1] + ptr_tmp1[3][1]*ptr_d[1][1] +
ptr_tmp1[3][2]*ptr_d[2][1] + ptr_tmp1[3][3]*ptr_d[3][1];
        ptr_out[3][2] = ptr_tmp1[3][0]*ptr_d[0][2] + ptr_tmp1[3][1]*ptr_d[1][2] +
ptr_tmp1[3][2]*ptr_d[2][2] + ptr_tmp1[3][3]*ptr_d[3][2];
        ptr_out[3][3] = ptr_tmp1[3][0]*ptr_d[0][3] + ptr_tmp1[3][1]*ptr_d[1][3] +
ptr_tmp1[3][2]*ptr_d[2][3] + ptr_tmp1[3][3]*ptr_d[3][3];

        return;
}



/*
 * vec_mult -
 *
 * This function evaluates the equation:
 *
 * output_vector = matrix x input_vector
 *
 */
void vec_mult (double *ptr_out, double **ptr_mat, double *ptr_vec) {

        ptr_out[0] = ptr_mat[0][0]*ptr_vec[0] + ptr_mat[0][1]*ptr_vec[1] +
ptr_mat[0][2]*ptr_vec[2] + ptr_mat[0][3]*ptr_vec[3];
        ptr_out[1] = ptr_mat[1][0]*ptr_vec[0] + ptr_mat[1][1]*ptr_vec[1] +
ptr_mat[1][2]*ptr_vec[2] + ptr_mat[1][3]*ptr_vec[3];
        ptr_out[2] = ptr_mat[2][0]*ptr_vec[0] + ptr_mat[2][1]*ptr_vec[1] +
ptr_mat[2][2]*ptr_vec[2] + ptr_mat[2][3]*ptr_vec[3];
        ptr_out[3] = ptr_mat[3][0]*ptr_vec[0] + ptr_mat[3][1]*ptr_vec[1] +
ptr_mat[3][2]*ptr_vec[2] + ptr_mat[3][3]*ptr_vec[3];

        return;
}
```

B.3 – Central.c (extract)

```
/*
 *        Author: Ian Marshall
 *
 *        This code is used to demonstrate active balance control of the GuRoo
 *
 */

void Demo_Balance() {
          kill_func();
          kill_servos();
          move.mctrl_fn = demo_bal_func;
          move.finished = false;
          move.duration = 4.0;
          move.time = 0;
}

void demo_bal_func()
{
          #define s_per 4.0f
          #define stand_lean_angle 11 * PI/180

          #define STAGE1 1.0f
          #define   STAGE2 (STAGE1 + s_per)
          #define STAGE3 (STAGE2 + s_per)
          #define STAGE4 (STAGE3 + move.duration)
          #define STAGE5 (STAGE4 + s_per)
          #define STAGE6 (STAGE5 + s_per)
          #define STAGE7 (STAGE6 + s_per)

          float ANKLE_ANGLE = (float)(30.0 * PI/180);
          float HIP_ANGLE = (float)(asin(0.17 * sin(lift_angle) / 0.265));
          float KNEE_ANGLE = (float)(ANKLE_ANGLE + HIP_ANGLE);
          float torso_correction = 10 * PI / 180 ;

          float velocity;

          // hold position
          if(move.time <= STAGE1) {
          }


          // shift weight over right foot
          if ((move.time > STAGE1) && (move.time <= STAGE2)) {
        velocity = Velocity2(s_per, STAGE1);
                    *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * stand_lean_angle));
                    *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC( (float) (velocity * (stand_lean_angle -
(3.5*PI/180)) ));
                    *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * stand_lean_angle));
                    *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * (stand_lean_angle +
(3.5*PI/180))));

          }


          // raise left leg
          if ((move.time > STAGE2) && (move.time <= STAGE3))
          {         velocity = Velocity2(s_per, STAGE2);
                    *desired_joint_vel[LEFT_HIP_FWD]  = SRAD2ENC((float)(velocity * HIP_ANGLE));
                    *desired_joint_vel[LEFT_KNEE]  = SRAD2ENC((float)(velocity * KNEE_ANGLE));
                    *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * ANKLE_ANGLE));
                    *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(-velocity * 5.0 * PI / 180));

          }


          // raise left leg further
          if ((move.time > STAGE3) && (move.time <= STAGE4)) {
                    velocity = Velocity2(s_per, STAGE4);

                    *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * 30 * PI/180));
                    *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * 40 * PI/180));
          }



          // bend right knee
          if ((move.time > STAGE4) && (move.time <= STAGE5))
          {
                    velocity = Velocity2(s_per, STAGE6);
                    *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity * 15 * PI/180));

          }


          // tilt torso
          if ((move.time > STAGE5) && (move.time <= STAGE6)) {
                    velocity = Velocity2(s_per, STAGE7);
                    *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * -20 * PI/180));

          }


          // twist torso
          if ((move.time > STAGE6) && (move.time <= STAGE7)) {
                    velocity = Velocity2(s_per, STAGE7);
                    *desired_joint_vel[TORSO_TWIST] = SRAD2ENC((float)(velocity * -20 * PI/180));
                    printf("final stage\n");

          }


          // finish
          if ((move.time > STAGE7)) {
                    printf("done movement\n");
                    Kill();

          }

}
```

Appendix C: SIMULATOR CENTRE OF MASS CODE

```
/*
  Method that works out the centre of gravity of the robot
  Added by CSH 5/9/02
*/
Float mass;
CartesianTensor inertia;
CartesianVector cg_pos;

void getCentreOfGravity(CartesianVector cog) {

        Float force = 0;

        Float cogX = 0;
        Float cogY = 0;
        Float cogZ = 0;

        for (int i = 0; i < num_links; i++) {
                dmLink* link = guroo->getLink(i);
                cerr << link->getName() << endl;
                dmRigidBody *rigidBody = dynamic_cast<dmRigidBody*>(link);

                if (rigidBody != NULL) {

                        const dmABForKinStruct* forwardKinematics = guroo-
>getForKinStruct(i);
                        rigidBody->getInertiaParameters(mass, inertia, cg_pos);

                        CartesianVector linkCOG = {
                                (Float)forwardKinematics->p_ICS[0],
                                (Float)forwardKinematics->p_ICS[1],
                                (Float)forwardKinematics->p_ICS[2]
                        };

                        cogX = ((mass * (linkCOG[0] - cogX)) / (force + mass)) + cogX;
                        cogY = ((mass * (linkCOG[1] - cogY)) / (force + mass)) + cogY;
                        cogZ = ((mass * (linkCOG[2] - cogZ)) / (force + mass)) + cogZ;

                        force = force + mass;

                } else {
                        cerr << "pointer was null" << endl;
                }


        }

        cog[0] = cogX;
        cog[1] = cogY;
        cog[2] = cogZ;


}
```

Appendix D: CD

D.1 – MATLAB Code

This CD includes all MATLAB Source Code used to generate the graphs displayed in this thesis.

The CD also includes:

- C-Source Code for the Balance System
- PDF Copy of this report