THE UNIVERSITY OF QUEENSLAND
DEPARTMENT OF INFORMATION TECHNOLOGY &
ELECTRICAL ENGINEERING

BACHELOR OF ENGINEERING THESIS

GAIT GENERATION & CONTROL ALGORITHMS FOR A HUMANOID ROBOT

ADAM DRURY
2ND OCTOBER 2002

Head of School
School of Engineering
The University of Queensland
St. Lucia, QLD, 4072

Professor Kaplan,

In accordance with the requirements of the degree of Bachelor of Engineering (Honors) in the division of Electrical Engineering (Mechatronics Minor), I present the following thesis entitled *'Gait Generation & Control Algorithms for a Humanoid Robot'*. The work was performed under the supervision of Dr Gordon Wyeth.

I declare that the work presented in this thesis has not been previously submitted for a degree at the University of Queensland or any other institution. To the best of my knowledge, this thesis contains no material published by any other person, except where reference is made in text.

Yours faithfully,

Adam Drury.

ABSTRACT

This thesis details the design, simulation and implementation of joint control software and gait generation algorithms for a 1.2m tall humanoid robot. The goals were oriented towards participating in the 2002 International RoboCup Competition.

The primary aim was to develop an open loop gait pattern that allows the robot to walk unaided at a speed of $0.1ms^{-1}$. Other objectives included: creation of an algorithm for the robot to stand on one leg, further development of the UQ humanoid simulation program and the creation and optimisation of joint control software utilising local feedback.

The control algorithm is responsible for actuating each of the 15 joints in the lower body and torso. A Proportional-Integral compensator is used and the gains were individually tuned to simulation results and finally, hardware results.

The project was almost entirely successful in achieving its objectives. The 'GuRoo' walks dynamically at speeds of between 0.067 and $0.1ms^{-1}$, with only minimal human assistance required for balance in the sagittal plane. The robot's capabilities were demonstrated at RoboCup and an overall position of 7^{th} in the world was attained. The control software can execute at over 1kHz, which is 20 times faster than the required 50Hz minimum. All the movements simulate successfully and have been verified with actual results.

CONTENTS

FIGURES

CHAPTER 1 – INTRODUCTION

1.1 Humanoid Robots

Humanoid robotics is a relatively new field (exemplified by the limited number of projects in the world) and offers interesting challenges particularly in the area of bipedal walking – the major focus of this thesis. There are several main reasons for building humanoid robots. Primarily, humans have altered the natural environment to suit their own physical form and dimensions. In order to best interact with humans in this environment; a robot should have approximately humanoid size and structure. Common challenges for a robot in such an environment include: navigating corridors and stairs, opening and fitting through doors and transporting objects to/from tables and cupboards. A humanoid robot can perform all these tasks in the same way a person would.

Additionally, humans are used to interacting with other humans. Therefore, interacting with a robot would be easier and more natural if it were in humanoid form. The final, somewhat philosophical reason is that: *"to build a machine with human like intelligence, it must be embodied in a human like body."* [1]

1.2 The RoboCup Competition

The RoboCup is an annual, international robotics competition based around the game of soccer. It aims to promote intelligent robotics research as well as providing a forum where groups can meet and exchange technical details whilst educating and entertaining the public.

The ultimate goal of RoboCup is: *"By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champions."* [2]

Although current technology is far from realizing this vision, 2002 sees the introduction of a humanoid league in the RoboCup. The goals of this thesis involve working as part of a team to produce a robot ready for the 2002 competition.

1.3 Walking

The first step to beating the World Cup champions would logically be to have a robot that can walk. The majority of mobile robots to date have used wheels for locomotion and legged robots have generally had four or more legs. These solutions have the advantage of maintaining 3 points of contact with the ground at all times, resulting in static stability.

Bipedal locomotion is inherently unstable and presents a more difficult task. Adding a humanoid torso, arms and head compounds the problem by raising the centre of mass and creating more joints that need controlling in order to balance the robot. Figure 1.1 illustrates the distinction between a biped and a humanoid.

An essential component of any robot is the ability to control its joints. Once the walking gait is developed, local control algorithms are required to ensure the various joints behave as desired.

1.4 Achievements

At the beginning of 2002, the University of Queensland's humanoid robotics project existed as a paper design with work just beginning on the mechanical construction. By 4:00am on the 15th June, the robot was packed and ready to travel to Japan for competition in RoboCup 2002.

Specifically, the achievements of this thesis are:

- The design of local, Proportional – Integral control algorithms for the 15 DC motor actuated joints in the lower body and torso,
- Continued development of the UQ Humanoid simulation program,
- The design of several movement algorithms, namely: crouching, standing on one leg and walking,
- Successful simulation of the control algorithms and all movements,
- Successful implementations of these designs resulting in a humanoid robot that can crouch and stand on one leg unaided and walk with minimal assistance.

1.5 Thesis Outline

CHAPTER 2 – Provides a description of UQ's humanoid robotics project "The GuRoo" as at the beginning of 2002 and briefly reviews similar robots. It also presents background theory.

CHAPTER 3 – Lists objectives for the 2002 GuRoo team and derives specifications for the control algorithms and gait pattern.

CHAPTER 4 – Details the design of the joint control algorithms and software.

CHAPTER 5 – Details the design of the walking algorithm and related movements.

CHAPTER 6 – Discusses and evaluates the results obtained from simulations and testing.

CHAPTER 7 – Provides concluding remarks and offers suggestions for future development.

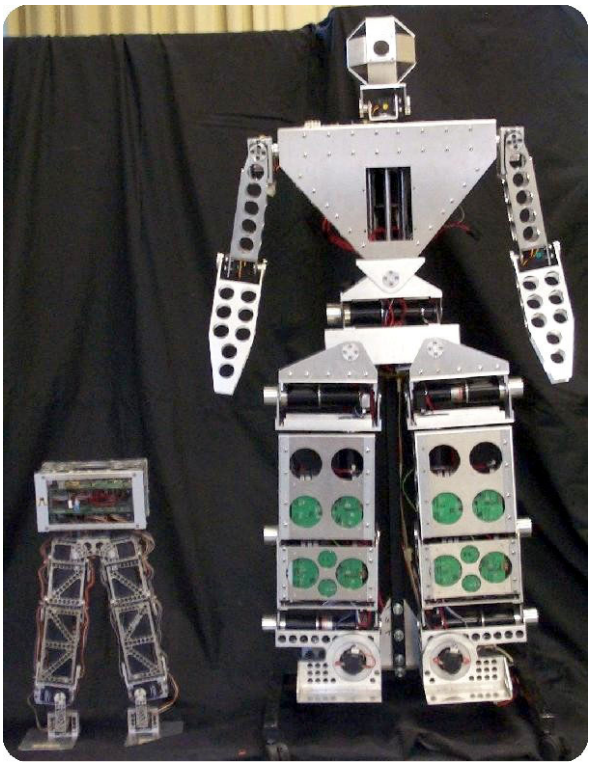


Figure 1.1: UQ's Biped (left) & Humanoid (right) Projects

CHAPTER 2 – PREVIOUS WORK

2.1 The GuRoo Project

In 2001, a team of 12 undergraduate students began the University of Queensland (UQ) humanoid project – "The GuRoo". They managed to complete most of the design work and obtain some simulation results. The design aspects relating to this thesis are as follows.

2.1.1 Hardware

Hi-Tec HS705-MG servomotors actuate the joints in the upper body with an integrated gearbox rated at 1.4Nm with a speed of 5.2rads^{-1}. There are eight of these motors: 1 in each elbow, 2 per shoulder and 2 in the neck.

The lower body requires high power Maxon RE36, 32V DC motors. Each leg has 2 motors in the ankle, 1 in the knee and 3 in the hip. There are an additional 3 motors in the spine totalling 15. These provide the roll, pitch and yaw actuation for legs, hip and torso. The hips also include a torsional spring on the roll axis that stores energy when leaning to the side and assists in returning to the neutral position. The motors are fitted with a planetary gearbox providing an output of 10Nm at 5.3rads^{-1}. Optical encoders provide position feedback and H-bridges give direction control (forward and reverse).

Each encoder has 2 channels measuring the rising and falling edges of 500 slots, giving 2000 counts per revolution. After the 1:156 gearbox, this increases to 312000counts/rev or 867 counts/degree.

TMS320F243 digital signal processors (DSP) provide the motor control. These DSP's are ideally suited to motor control featuring: eight channels each of pulse width modulated (PWM) output and 10-bit analogue to digital (AD) conversion, a 20MHz clock speed and 8k words of flash memory. Communication between boards is via a high speed Controller Area Network (CAN) bus. [1]

2.1.2 Control Algorithm

The GuRoo uses a distributed control system with five DSP's in the lower body and torso. Each DSP locally controls three DC motors, whilst a sixth TMS processor operates the servomotors. A serial link to a laptop or palmtop (IPAQ) provides the central, high-level control such as gait generation. Figure 2.1 is a block diagram of the system used for the 2002 design.

Zelniker [3] constructed a simplified model of the DC motors and added a proportional integral (PI) compensator with feed-forward (F) cancellation. He performed simulation tests in 'Simulink' to determine values for the PIF gains.

The GuRoo simulator did not use the feed-forward path and had the same standard gain values (P=25.5 and I=2700) for all motors. The small servomotors have a built in feedback loop and require only an appropriate (PWM) voltage to set their position.

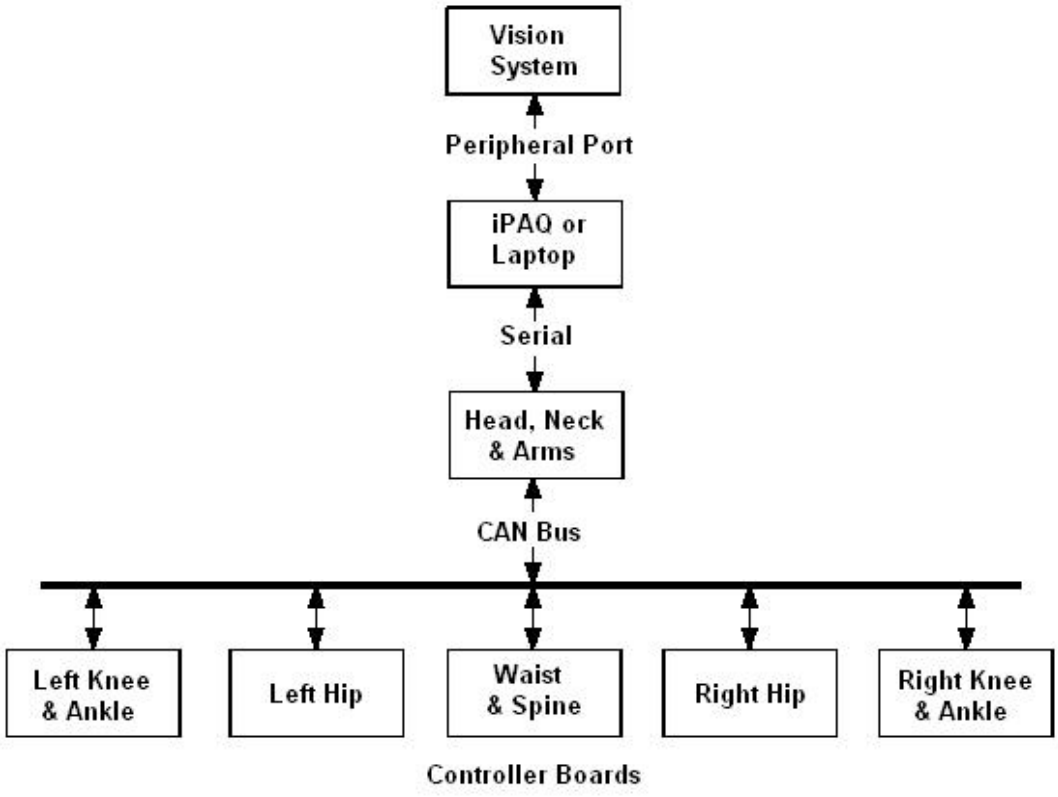


Figure 2.1: The Distributed Control System

2.1.3 Software

Smith [4] adapted the DynaMechs simulator created by McMillan [5] to suit the GuRoo. He also developed a static walking algorithm and simulated it successfully.

Stirzaker [6] wrote several pieces of low-level code for reading the encoder, setting interrupts and generating a PWM signal.

Generally, the mechanics and hardware were ready for construction at the end of 2001. However, much software still needed writing; particularly the embedded code for the TMS processors and improvements to the simulator.

2.2 Other Humanoids

The most advanced and publicly well-known humanoid projects in the world are Honda's ASIMO [7] and Sony's SDR-4X [8] (see figure 2.2).

Honda has spent 15 years on various humanoid projects, culminating in the development of ASIMO. This robot can travel at 1.6km/h, walk sideways, turn corners and grasp objects. It has the ability to change walking patterns smoothly, predict the next movement and adjust its centre of gravity accordingly. DC servomotors coupled to a harmonic speed reducer and drive unit control these actions.

ASIMO is of interest because it is similar in scale to UQ's humanoid. Both are 1.2m tall and have a mass of approximately 40kg. There are however, significant differences in the respective budgets.

Sony's robot has limited available data and is of less interest due the small size (56cm). Its most notable feature is the ability to navigate uneven terrain (10mm variations) and to walk up a 10° incline. This is achieved through a combination of *"Real-time Integrated Adaptive Motion Control"* [8] that allows the robot to maintain its posture when subject to external forces and real-time gait pattern creation. The joint positions are also flexible, providing a more natural gait and minimising damage from falls. Relatively small servomotors drive the joints.

The SDR-4X also features advanced audio and visual technology such as voice and face recognition designed for entertainment purposes.

Figure 2.2: Honda's ASIMO [7] and Sony's SDR-4X [8]

2.3 Control Theory

There are two types of classical control techniques: open loop and closed loop. In open loop systems, the output has no effect on the controller and hence the system is unable to correct for disturbances. Open loop control is useful only where the input-output relationship is well defined.

Closed loop systems measure the output and generate an error signal based on the difference between the input and output. This makes the control more robust, providing less sensitivity to noise and disturbances, the ability to correct for disturbances and greater accuracy. [9] It also allows for the use of compensation techniques.

The indicators of 'good' control are the system's ability to:

- *"Generate a response quickly and without oscillation (good transient response),*
- *Have low error once settled (good steady-state response),*
- *Not oscillate wildly or damage that system (stability)."* [10]

The following sections describe the Proportional, Integral and Derivative cascade compensation techniques. These can be combined to make PI, PD and PID compensators. Figure 2.3 is a block diagram of a system showing proportional, integral and derivative paths.

2.3.1 Proportional Control

Proportional control is simply gain adjustment. The output is a scaled version of the error signal. Increasing the gain yields a faster transient response at the expense of increased steady state error and greater overshoot.

2.3.2 Integral Compensation

The addition of an integrator to the system helps reduce the steady state error. Any small error signal will accumulate over time and drive the output toward its desired value. The integrator takes longer to act compared to the rest of the system and the overall transient response is usually slightly slower.

2.3.3 Derivative Compensation

Derivative compensation aims to improve the transient response (faster rise and settling times) whilst maintaining the same overshoot. This technique involves amplifying the derivative of the error signal. Sudden changes in the error therefore produce a stronger signal to the plant. This has the disadvantage of amplifying any noise present in the system.

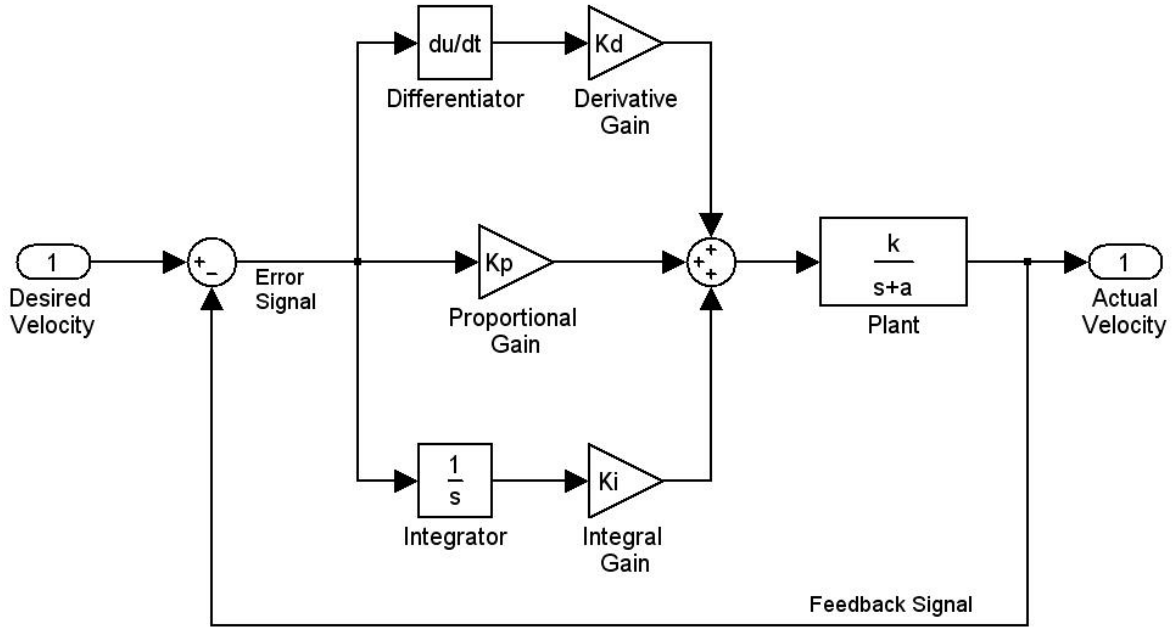


Figure 2.3: A PID Compensated Control System

2.4 Bipedal Walking

The major distinction between walking algorithms is whether they are static or dynamic. A static algorithm ensures that the robot is statically stable at all times. The robot can be paused at any point in the gait and remain balanced. This is because the centre of mass stays within a polygon drawn around the supporting foot/feet. Smith [4] used this principle to develop his gait pattern for the GuRoo.

Dynamic walking is difficult to implement but more closely reflects human walking. At certain points in the gait, such as when the swing leg extends and has the weight transferred onto it, the robot is unstable and actually falls into the next step. Dynamic algorithms generally result in faster walking that looks more natural.

CHAPTER 3 – SPECIFICATIONS

The ultimate goal of the 2002 GuRoo team is to demonstrate the robot's capabilities in the solo section of the 2002 RoboCup Humanoid League. This will require a 1.2m robot to:

a) Stand on one leg and hold a steady position for 1 minute.

b) Walk from a starting position in a straight line; navigate around a pole and return.

c) Kick a ball from 3.6m out into a 2.4m wide goal; starting from a position 1.8m behind the ball.

This thesis focuses on making the robot stand on one leg and walk; leading to the achievement of objectives a) and b). Objective c) and the second half of b) rely heavily on the development of a vision system.

Some general specifications are that all software should be directly portable between the simulator and the actual robot. Specifically, the same control code should compile and run in the simulator and on all five DSP control boards. The gait generation algorithm must interface with both the simulator and the serial to CAN communications. Smith [4] made an attempt at this objective but did not achieve it.

Additionally, all work needed to be completed before the RoboCup competition on the 19th of June 2002.

3.1 The Control System

The existing hardware design dictates the use of a distributed control system. Each controller can use only the information available locally, without 'knowledge' of the global state of the robot or of any other joints. Currently, the only available feedback information is the actual position of the motors.

Calculating the load on an individual motor or solving the inverse dynamics for 24 links with 23 revolute joints would be extremely complex. Marshall [11] demonstrates this fact in his centre of mass calculations for the GuRoo. Even if these calculations were performed and the resulting software could fit onto the TMS320F243 DSP, 'real-time' control would not be possible given the 20MHz clock speed. These restrictions limit the choice of control algorithm to local, classical feedback control.

Figure 3.1 defines the system boundaries and interfaces for the control algorithm. Desired joint velocities arrive on the CAN bus and the can_receive() function places them in an array for use by the control loop. The velocities are updated at a rate of 50Hz. The other input is the current encoder reading as returned by the read_enc() function. The control software must use this information to calculate a PWM value between ±100% to apply to the motors.

The most critical aspect of the control algorithm is its efficiency. The control loop needs to execute fast enough to justify the approximation of an analogue controller using digital hardware. The inputs must be sampled and an output produced rapidly enough to approximate a continuous function. The minimum operating frequency is 50Hz; being the rate at which the desired joint velocity array is updated. The maximum speed is limited by the 100kHz switching of the PWM channels. Additionally, each DSP controls three DC motors meaning that the algorithm must execute three times every cycle. It is desirable to have the speed as high as possible for tighter control.

The control system must first be able to hold the GuRoo steady when in an upright standing position. This ability to 'lock' the joints will be examined by applying a disturbance torque to one of the joints and plotting the system response. A successful implementation will yield a robot that can stand under its own weight and correct for any displacement error produced by the applied torque.

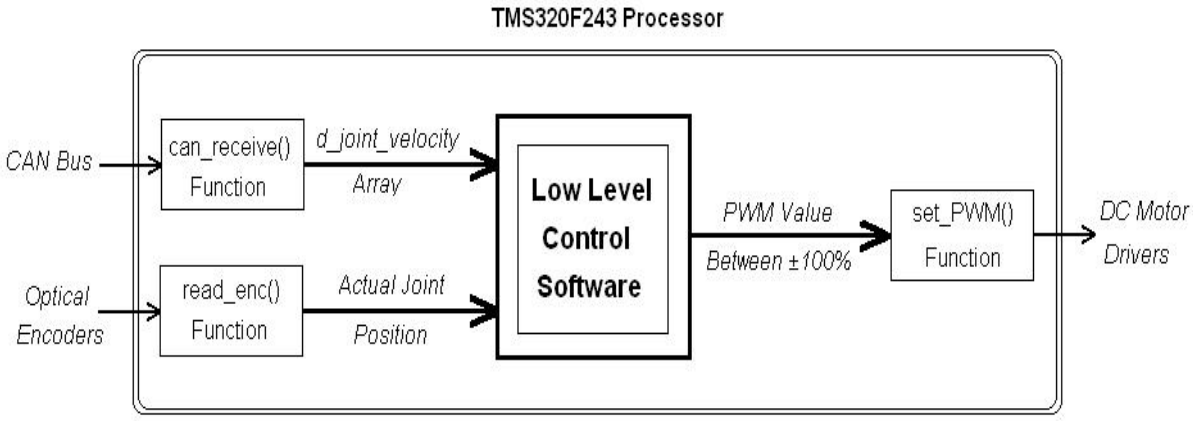


Figure 3.1: Control System Boundaries & Software Interfaces

The second test of the control system is its ability to track changes in the desired joint velocity. There are going to be significant variations in the load on each motor and obviously, the control algorithm cannot perform optimally in all situations. The movement algorithms emphasise arriving at a desired joint position at the specified time. Accurately matching the velocity profile is of secondary importance as long as the position error is small. Hence, the control loop design aims to minimise the RMS value of the error between the desired and actual velocity and position, with emphasis on reducing the magnitude of the maximum position error.

3.2 The Walking Algorithm

Smith [4] specified that his walking algorithm should allow the GuRoo to walk at $0.1ms^{-1}$ whilst observing power restrictions. These specifications have been adopted for this thesis under the assumption that *'power restrictions'* refers to the 2A current limit of the DC motors.

Smith also makes a vague statement that the gait should resemble human walking. He did not achieve this because humans do not walk statically. Therefore, it is the aim of this thesis to develop a dynamic walking algorithm.

The walking algorithm will operate as an open loop, meaning that there will be no data fed back to the central controller. This makes balancing extremely difficult because the gait generator has no knowledge of where the GuRoo's joints actually are, only where they should be. Consequently, there is a heavy reliance on the local control loops to correct for disturbances and ensure the actual joint position closely matches the desired.

The ideal algorithm would allow the GuRoo to walk in a straight line on a flat surface without human assistance. Realistically, this task is very difficult without active balance control and requires feedback at least from the encoders and ideally, force sensors on the bottom of the feet and gyroscopes.

CHAPTER 4 – CONTROL ALGORITHMS

This section describes the design of the joint control algorithms based on the specifications derived in chapter 3. The simulator was used to verify the design and assist with tuning of the control parameters. Initial tests were carried out on the robot using simple movements and the results are presented here because they were required before the gait was developed.

4.1 Design

As mentioned in the specifications, feedback control is required to accurately track the desired velocities. The optical encoders provide a position reading that needs differentiating to produce an actual velocity. The differentiation process has the effect of amplifying any noise present in the system. Hence, it was decided not to use a derivative compensator because this would further compound the noise problem.

Position accuracy has been identified as highly important when controlling the GuRoo's joints. Any steady state error in velocity will be a constantly increasing error function in position. Therefore, integral compensation was required. A block diagram of the control system for one motor is shown in figure 4.1. The integrator and zero-order hold in the feedback path model the effect of the optical encoders.

4.2 Implementation

The control code needed to run on the TMS320F243 DSP, which is a 20MHz, 16-bit fixed-point processor. All data types are either 16 or 32-bit. Floating-point numbers are represented as IEEE single precision format, with no difference between single and double precision (both are 32-bit). Floating-point arithmetic can be performed using a run-time-support library that contains a set of math functions. 'The compiler pushes the arguments onto the run-time stack and generates a call to a floating-point function. The function pops the arguments, performs the operation and pushes the result onto the stack." [12] The implication is that performing a

simple multiplication, even with small numbers such as 1.2 x 2.3, requires several stack operations and function calls with 32-bit arguments. This process is very time consuming and has a significant impact on the efficiency of the code. Consequently, all floating-point numbers were eliminated from the control algorithm. Additionally, all multiplications and division were removed and the number of conditional branches was minimised.

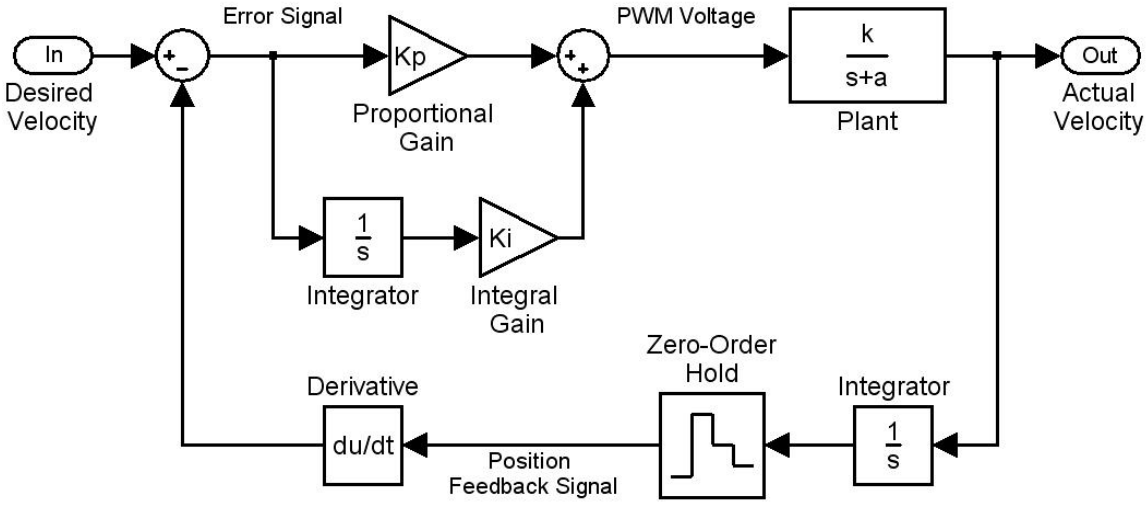


Figure 4.1: Control System for a Single Motor

These optimisations meant that integers had to be used for every operation without losing accuracy and measurements could no longer be converted to SI units. Consequently, displacement was measured in raw encoder counts and the time base was changed to one control loop period. As an example: velocity was changed from radians per second to encoder counts per control loop. Differentiation then became a straight subtraction and integration an addition.

The velocities generated by the central controller are specified in radians/second and now need converting before being broadcast on the CAN bus. The conversion algorithm is simply:

$$SRAD2ENC(Velocity) = (\text{int})\left(\frac{Velocity \times Control\_Period}{2\pi \times Encoder\_Counts\_Per\_\text{Re}volution}\right)$$

The problem of how to specify the proportional and integral gains was solved by defining them as an amount of bit shifting. The resolution of the gains is now limited to powers of 2, but this was considered an acceptable trade-off for increased execution speed.

Optimisation of the code was constrained by several factors. Firstly, there was a need to maintain simulator compatibility. This prevented the use of assembler code (which is generally faster than optimised C) and forced the inclusion of some unnecessary arrays. Secondly, the GuRoo is to be an ongoing project and emphasis was placed on code readability. For this reason, the code for the 3 motors on each board was placed in a for-loop with more arrays indexed by the iteration number of the loop, which is slower than writing the code sequentially and using extra variables.

4.2.1 Board1.c

Figure 4.2 is a flowchart showing how the control software was coded. At the start of each loop, a current encoder reading is taken. The previous position is subtracted to perform the differentiation to actual velocity. If the magnitude of this velocity is very large (greater than 0x800), it indicates an encoder overflow. This occurs because the encoders have a 16-bit output, (maximum of 65535 counts) and there are 312000 counts per revolution. Once detected, it is easily corrected by subtracting or adding 0xFFFF depending on whether the overflow was positive or negative.

Next, a 32-bit count was kept of the absolute position of the joint by accumulating the actual velocities. None of the GuRoo's joints can rotate through a full 360º due to mechanical and wiring restrictions. Assuming the GuRoo had the same initial position every time, each joint was limited to a predetermined number of encoder-counts either direction. Software position limiting helped prevent cables being sheared and motors driving against mechanical limits causing large currents that damage the armature winding. This code has since been removed for speed reasons and because the new controller hardware designed by Hood [13] includes optical limit sensors.

After the old position variable has been updated, the proportional error is calculated as the difference between the desired and actual velocities. This error is then added to the

accumulated integral error. Next, the errors are bit shifted by their respective gains and added to produce a PWM value.

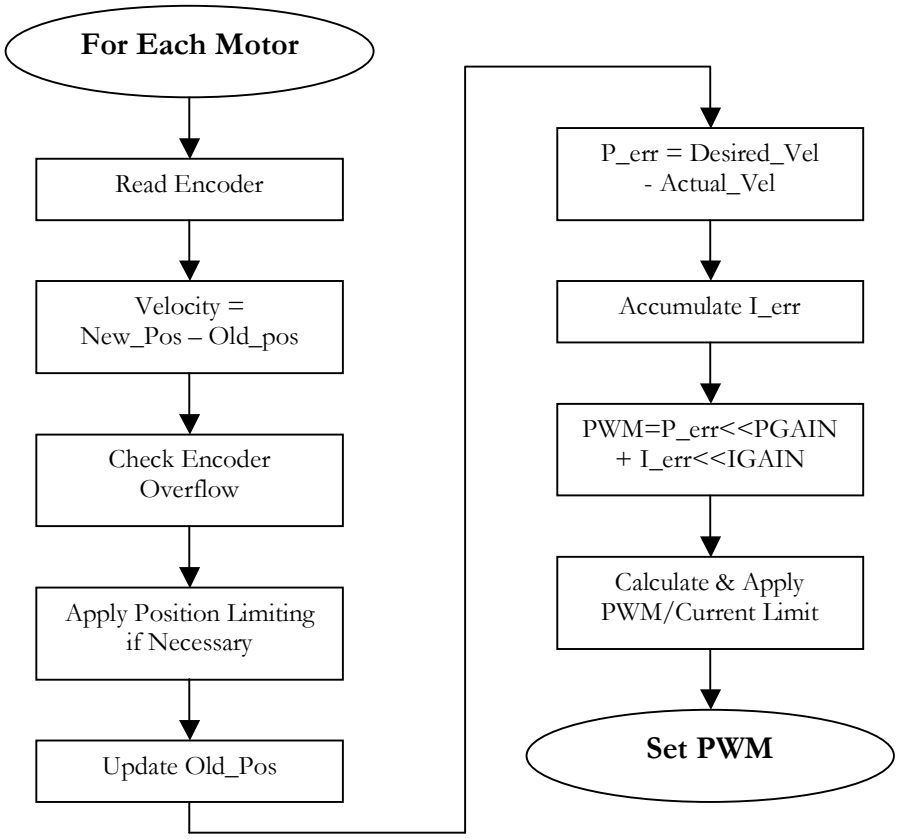


Figure 4.2: Control Code Flowchart

As a further safety precaution, the PWM value is limited to ensure the motor will not exceed its current limit. This limit is calculated dynamically based on the current velocity according to the following equation:

$$Back\_EMF = K_v \times velocity$$
$$PWM\_Limit = IR - Back\_EMF$$

where: K_v = Velocity Constant (Vs/rad),

I = Current Limit (A),

R = Armature Resistance (Ω).

Finally, the PWM value is applied to the motor via a call to the set_PWM() function. For details on setting the PWM and the read_enc() function, see Hood [13]. A complete listing of the board1.c code is presented as appendix A.1.

4.2.2 Startup.c

This file contains the main loop and initialisation routines. In keeping with the specification that the same code should run on all 5 controller-boards, an identification number was required to select the correct gain values. Currently, the ID number is simply #defined and needs changing for each board that is programmed (note that this is the only software difference between boards). The new hardware design however, will include DIP-switches for identification and allow swapping of boards without re-programming.

Once the ID has retrieved, several functions are called to set up the board including the: serial communications interface, analogue to digital Converter, quad-decoders, PWM and CAN. Hood [13] provides details of these functions. Next, the init_gains() and init_pos() functions in board1.c are called. These load the correct gains (based on BOARD_ID) into an array and read the encoders to initialise the old_j_pos array.

Execution then returns to the main loop and waits for periodic interrupts to trigger the control loop. The TMS320F243 contains only 2 timers, both of which are used by the PWM, which interrupts at 100kHz. The control algorithm is called after 400 PWM interrupts are counted, giving an operation speed of 250Hz. This speed is slower than desired, but was initially chosen to allow time for serial debugging.

4.3 Preliminary Testing

It was necessary to perform these tests in both the simulator and on the robot to verify the design and implementation of the control algorithm before the walking gait could be developed. Simulations provided a rough estimate for the magnitude of the proportional and integral gains and hardware results were used for final tuning. These hardware results are summarised below.

4.3.1 Current Limiting

The current limiting was tested by connecting a multimeter in series with the motor and manually applying a torque. The control loop held the motor in a constant position until exactly 2A was reached, at which point it could be moved.

4.3.2 Control Algorithm Speed

The overall speed of the control algorithm was measured by turning on a LED just before calling b1_control() and turning it off after execution returned to the main loop. Two additional LED's were used in a similar manner to test the time taken by the read_enc() and set_PWM() functions. A CRO was used to measure the voltage applied to the LED's and the resulting waveforms are shown in figure 4.3.

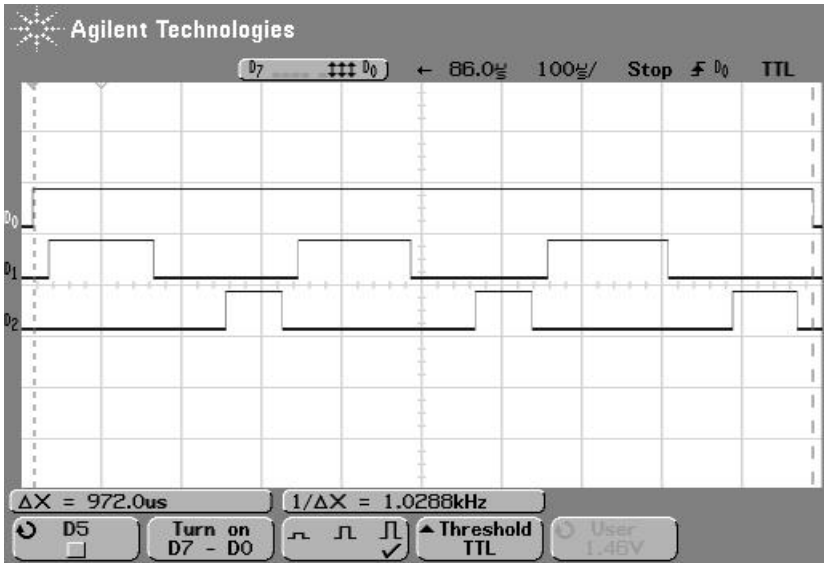


Figure 4.3: Waveforms from Control Loop Speed Test

Channel D_0 shows the total time taken to execute the control loop for three motors. The cursors measure this as 972µs or a frequency of 1.0288kHz. Channel D_1 shows that the read_enc() function takes 142µs each time it is called whilst the set_PWM() function on D_2 takes 72µs. The total time spent reading the inputs and setting the outputs is thus: 3 x (142 + 72) = 642µs or 66%. Therefore, the maximum speed attainable with this system (see figure 3.1) is 1/642µs = 1.5576kHz.

4.3.3 Joint Control

The results presented here show the response of the left knee for a crouching motion. The knee is a critical joint for this movement and can be used to show performance of the control loop over the full range of loads.

The minimum loading on the knee is when it only has to raise the weight of the foot and lower leg. This occurs during walking when one leg acts as the single support and the other lifts off the ground before being swung forward. The average load occurs during the double support phase, when both feet are firmly on the ground; each knee supports half the weight of the robot. The third experiment involved the GuRoo standing on one leg and attempting to perform a crouching motion. The knee was required to lower then raise the entire weight of the robot (with the exception of the lower half of the standing leg). This full loading situation does not normally occur and was only tested for completeness of results. The results for minimal, half and full loading are shown in figures 4.4(a), (b) and (c), respectively. Figure 4.5 is a photo of the crouching (average load) experiment.

The gains were tuned to minimise the RMS value of the error between the desired and actual trajectories for the average load and this is evident in the results. During the minimal loading test, there are slightly more oscillations because the gains are higher than optimal for this load, giving a larger RMS value. In all experiments, the oscillations are greatest around the peak of the velocity curve. This is because the acceleration is changing direction, corresponding to the maximum value of the 'jerk' function (see section 5.1 for an explanation of 'jerk' in relation to the velocity profile). Although there is some position error with the average load on the way back up, it is driven to zero by the integral compensator. 'Backlash' in the gearbox is evident as a longer than desired horizontal line at the point where velocity changes direction.

The results of the fully loaded experiment demonstrate primarily that one leg is capable of accurately lowering the total mass of the GuRoo in a controlled manner. However, it is not powerful enough to raise robot and human assistance was required towards the end of this test. The horizontal line between t = 10.5 to 12.5s demonstrates the PWM being limited to maintain a maximum current of 2A. The 'jerk' caused by the peak of the velocity profile was

enough to cause the knee to collapse at t=12.5s. The sudden increase in velocity corresponds to outside assistance. The control loop then continues to track the desired velocity despite a huge position error, indicating that the integral error must have either overflowed or saturated.

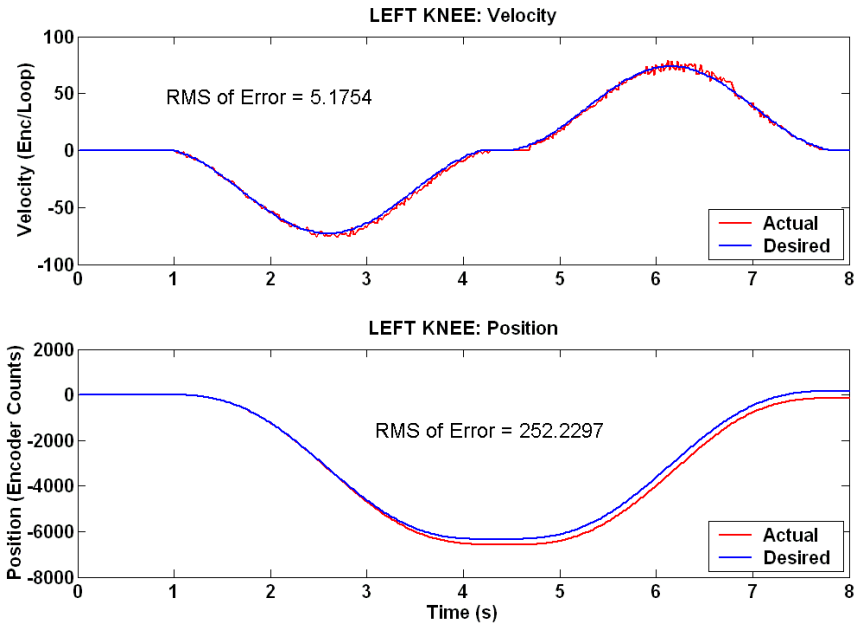


Figure 4.4(a): Left Knee with Minimum Load

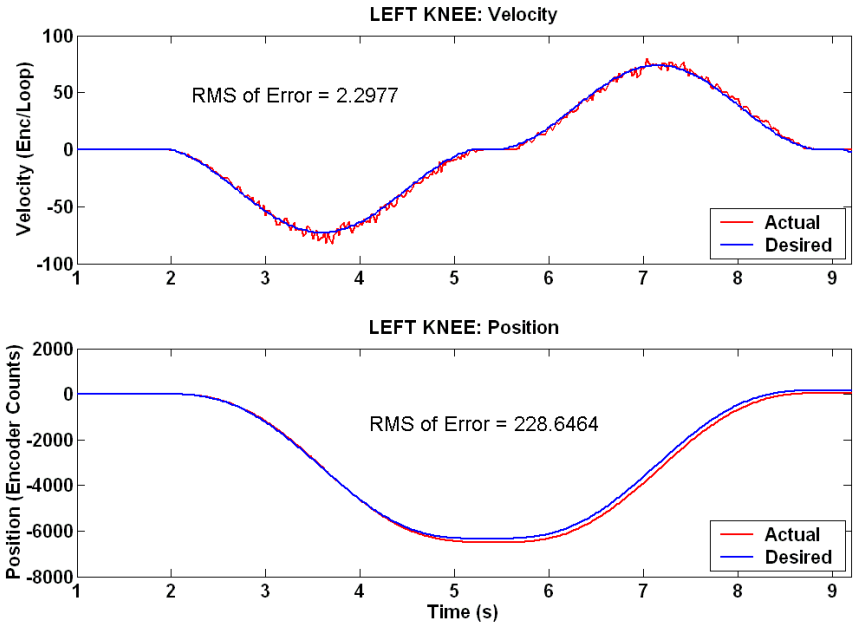


Figure 4.4(b): Left Knee with Half Load

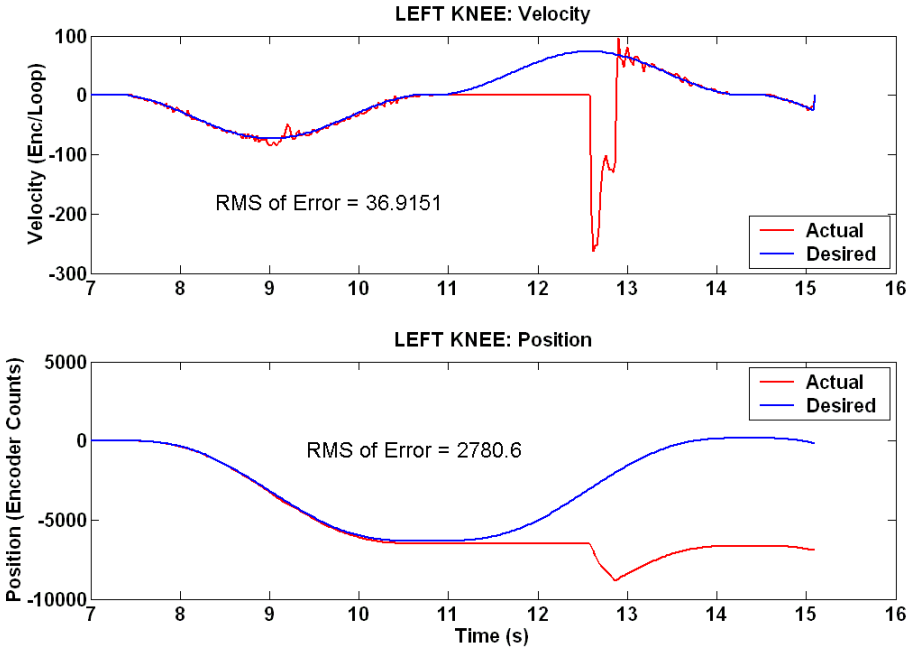


Figure 4.4(c): Left Knee with Full Load

Figure 4.5: The GuRoo Crouching

CHAPTER 5 – GAIT GENERATION

This section details the steps involved in engineering the walking algorithm. Some simple movements were designed and tested first before being integrated into the gait pattern.

5.1 The Velocity Profile

For each of the GuRoo's movements, a sequence of desired joint positions is generated. The velocity profile is responsible for defining how to move to these positions. Outlined below are the design considerations for the joint trajectories.

The GuRoo's joints do not have brakes, which means the velocity profile must begin from and taper to zero. For momentum reasons, it is desirable to have the maximum velocity close to the midpoint of the movement.

At the beginning and end of the movement, there is often backlash in the gearbox that needs overcoming, particularly if a change in direction is involved. Low acceleration is desirable at these points so the gears do not crash into each other at higher than necessary speeds. This will help reduce wear on the gearboxes.

The final and most important factor is the need to minimise disturbances caused to other joints. Defined as the time derivative of acceleration, the 'jerkiness' of a trajectory is what causes these disturbances. Step changes in 'jerkiness' on one joint transmit throughout the entire robot. Hence, a smooth continuous 'jerk' function is required.

A sinusoid of the form $\omega = \frac{1}{T}\left(1 - \cos\left(\frac{2\pi t}{T}\right)\right) \times \theta$ is a solution that meets all these criteria. Figure 5.1 shows a plot of this velocity profile along with the acceleration and jerk functions. Note the small magnitude of the 'jerk' curve – of the order of 10^{-5}. In addition, the area under the velocity function is equal to θ for any period, T. Therefore, multiplying the velocity profile

by a desired angle and specifying the move period will generate an accurate trajectory for that joint. Alternatively, the maximum acceleration occurs at $t = \frac{T}{4}$ and is given by:

$$\alpha = \frac{d\omega}{dt} = \frac{2\pi t}{T^2}\sin\left(\frac{2\pi t}{T}\right)$$

$$\alpha_{max} = \frac{\pi}{2T}$$

$$\therefore T = \frac{\pi}{2 \times \alpha_{max}}$$

It is now possible to specify movements as angle of rotation, θ, with maximum acceleration α_{max}. Defining a maximum acceleration gives control of the applied torque, which is directly proportional to the armature current of the motor. This could possibly allow for joint stiffness control and over-current protection.

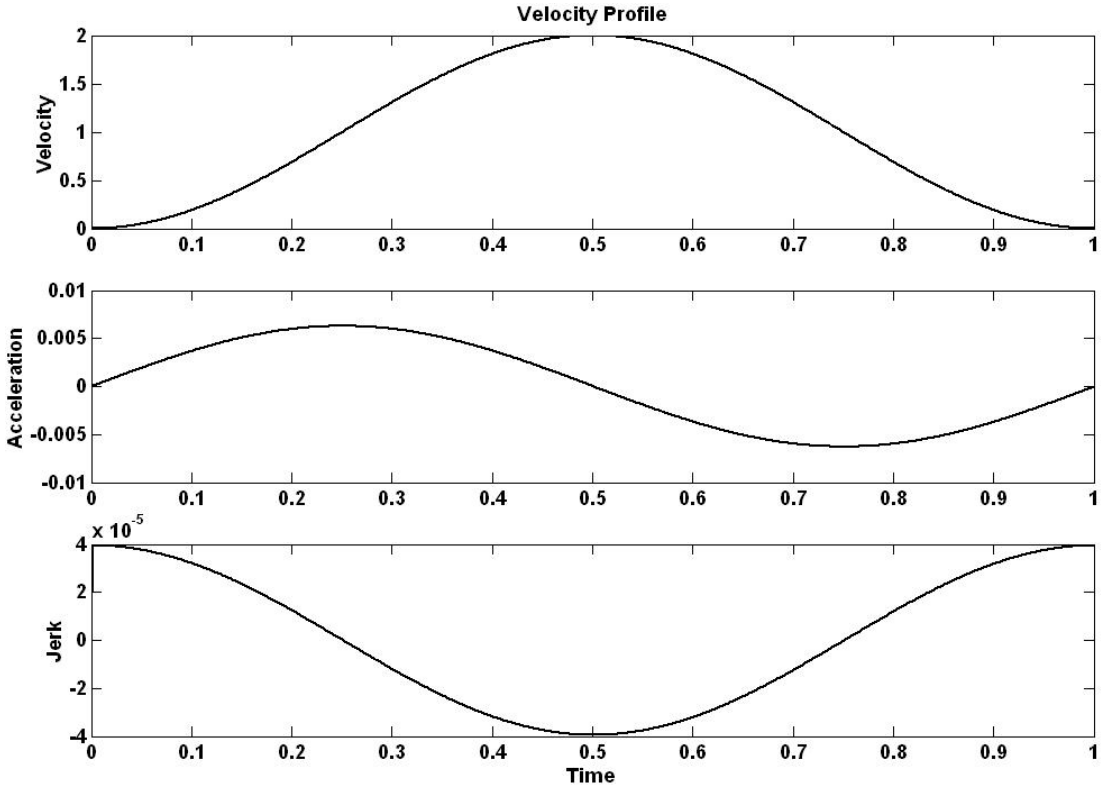
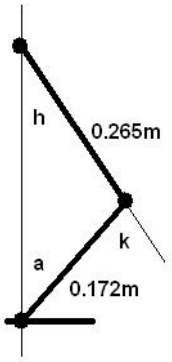


Figure 5.1: Joint Velocity, Acceleration and Jerk

5.2 Crouching

The first movement designed for the GuRoo was a repetitive crouching motion. This involved compressing the legs by moving the pitch motors in the ankle, knee and hip. For stability, it was necessary to keep the hips directly above the centre of each foot. The calculations in figure 5.2 show derivation of the joint angles.



a = ankle angle

k = knee angle

h = hip angle

$$\frac{\sin(a)}{0.265} = \frac{\sin(h)}{0.172}$$

$$\therefore h = \sin^{-1}\left(\frac{0.172\sin(a)}{0.265}\right)$$

$$k = a + h$$

Figure 5.2: Joint Angle Derivation for Crouching

Therefore, specifying an ankle angle will control the degree of the crouching motion. The hip and knee angles will adjust to ensure stability. Figure 5.3 shows this movement being simulated.

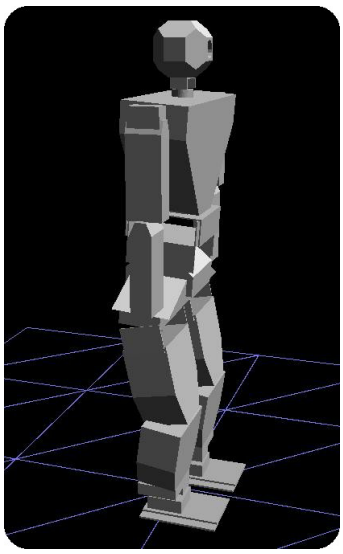


Figure 5.3: Simulated Crouching

5.3 Standing on One Leg

Standing on one leg is the first competition at RoboCup and is not a trivial task. The first part of this motion involves shifting the GuRoo's Centre of Mass (CoM) over the support foot as shown in figure 5.4.

Moving the ankle and hip roll motors by an equal amount keeps the legs parallel and allows transfer of weight in the lateral plane. Smith [4] used an angle of 0.1786 radians (10.233º) to transfer the weight in his static walking algorithm. Experimentation showed that the optimum angle was 12.5º. A correction angle of 3º was also required to drive the hip joints apart when the outer leg looses contact with the ground. Section 6.2 provides additional information on actual results.

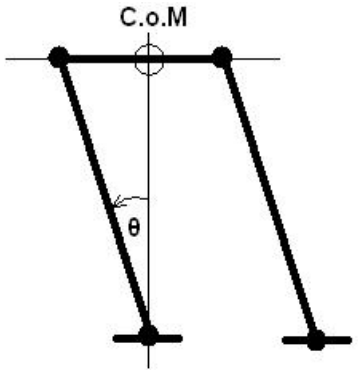


Figure 5.4: Moving the CoM over the Support Foot

The second stage utilises the crouching motion described in section 6.2 to raise the outer leg. A 30º-ankle angle is used to demonstrate sufficient clearance between the robot's foot and the ground. The local control loops are then responsible for holding this position steady for the required duration (60sec).

Returning to the initial position simply involves performing the same movements in the reverse order with opposite directions.

5.4 Walking

Dynamic walking presented a significant challenge and its development relied heavily on experimentation. There was no feedback from the joint controllers and no global sensors such as gyroscopes to work with. This meant that an open-loop algorithm was required. Furthermore, the movement algorithms do not have a reliable fixed reference point to start from on the actual robot. Although the optical encoders have an index mark, Stirzaker [6] did not allow for this in his hardware. The GuRoo, therefore, has no measure of absolute position. Hood's hardware design [13] corrects this flaw. However, a temporary initialisation routine was required. The ankle joints are the most critical because a slight inaccuracy here will be greatly magnified along the length of the robot. They are aligned by holding both the roll and pitch motors at their mechanical limits when powering the GuRoo. The align function then moves the roll motor 24º and the pitch motor 44º so the feet are parallel to the floor. The remaining joints are aligned by sight using keyboard commands transmitted serially – a far from ideal solution. As an additional complication, at this stage of the project, time pressure was becoming critical with the RoboCup competition drawing near.

The design of the walking algorithm utilises an inverted pendulum-type rocking motion where the robot continually sways laterally, independent of the forward motion. This swaying is identical to the first stage of the standing on 1 leg algorithm except the amplitude is gradually increased over the first three cycles. Initially, the GuRoo sways 2º to the left and the hips are driven an additional 2º apart. This causes the feet to tilt slightly outwards which assists with the rocking motion and corrects for any 'sag' in the roll motors. The robot then sways 6º to the right followed by 8º left. This 8º movement is now continually repeated in alternating directions with a period of 2s, resulting in a 4º deviation from the centre every second. These values of amplitude and period were experimentally determined to closely correspond to the natural frequency of the inverted pendulum model of the robot.

The 2s period of the algorithm is divided into 8 stages of 250ms each. During stages 1-4, the roll motors move the CoM to the right, while the left sway occurs during stages 5-8. At the extreme points of the lateral motion, the momentum and inertia of the robot cause the outer

(swing) leg to lose contact with the ground. During stage 4 (0.75s<t≤ 1.0s), the dynamic nature of the algorithm is evident. The pitch motors in the ankle, knee and hip use the crouching algorithm to raise the outer leg to an ankle angle of 18º. Simultaneously, the leg yaw motors move in the same direction to twist the hips to a position 8º forward of neutral. This causes the left leg to swing forward while the right leg acts as a single support. Additionally, the torso yaw motor twists by half the hip angle to ensure the GuRoo's chest faces the direction of travel. Note that for static stability when standing on 1 leg, a 12.5º deviation from the centre was required. Here, a 4º deviation is sufficient for dynamic stability.

During stage 5 (1.0s<t≤ 1.25s), the swing leg is extended and the CoM begins moving back to the centre position. The robot then falls on to the extended leg and enters the double support phase, further illustrating the dynamic aspects of the gait. By the end of stage 6, the CoM is back in the centre of the support polygon created by the feet on the ground. Stages 7 and 8 continue the sway to the extreme left. During stage 8, the right leg becomes the swing leg. It is raised and twisted forward as before, to be extended in stage 1 of the next cycle. The entire process then repeats. Figure 5.5 summarises the design of the walking algorithm.

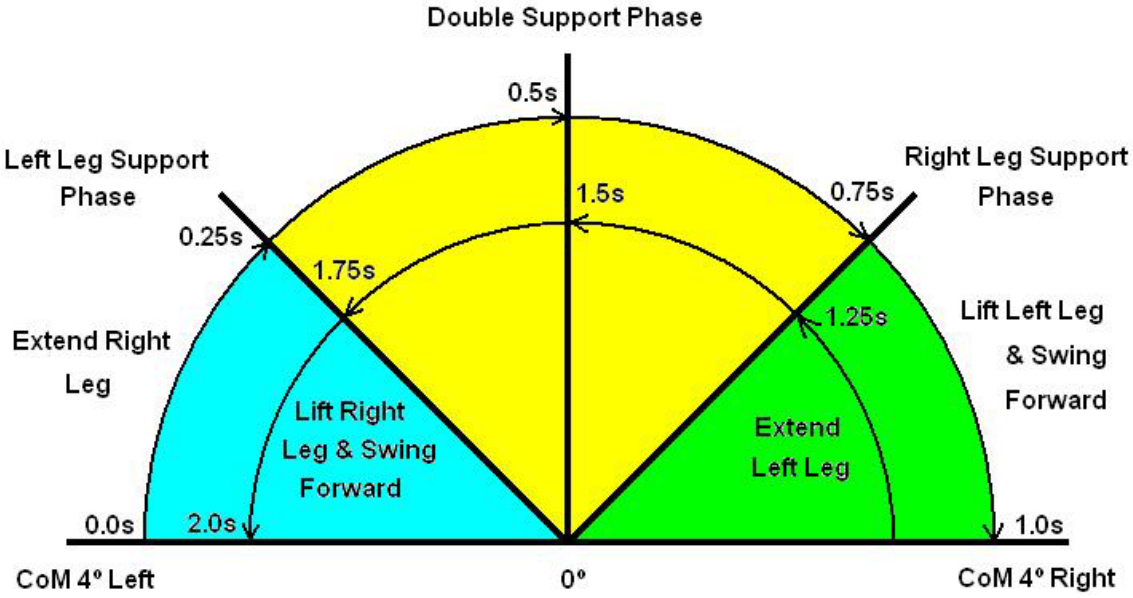


Figure 5.5: Summary of Dynamic Walking Algorithm Design

CHAPTER 6 – RESULTS

Presented here are the results of the major deliverables of this thesis; namely, the algorithms for: joint control, standing on one leg and walking. Simulation results are presented alongside the corresponding data from real hardware experiments. Screenshots and photos are used to demonstrate that the GuRoo is capable of performing the stated movements in both the simulator and during actual tests. Finally, the outcomes of the 2002 RoboCup competition are summarised.

6.1 Control

In addition to the preliminary results discussed in chapter 4, the performance of the control system is evaluated in the context of the walking gait for all relevant joints (see section 6.3). Generally, though, as work progressed and more movements were developed, it was discovered that the control was too 'sloppy' and joints were collapsing under the weight of the robot. In particular, the ankle roll motor caused major problems when attempting to stand on one leg. It was proven in section 4.3.3 that a single motor is not capable of raising the total mass of the robot while staying within the 2A continuous current limit. In 2001, this problem was solved for the hip roll motor by altering the mechanical design. A torsional spring was added to assist the joint when rising. Obviously, a similar solution was not possible this year and the only viable option was to increase the control gains as necessary. The final gain values that allow for dynamic walking are given in appendix A.1.

The second factor contributing to the 'sloppy' action of the join control was backlash in the gearboxes. This is most noticeable on the pitch axes when a small displacement on the ankle is compounded through the knee, hip and torso motors resulting in significant head movement. A method for reducing the effect of backlash was discovered by noticing that other humanoid robots walk with their knees slightly bent at all times. This maintains a loading torque on the motors and gearboxes – as opposed to keeping the legs straight under the robot and transmitting most of the load through the frame. Consequently, a small (6º) 'squat' function

was developed utilising the crouch algorithm and is performed as part of the GuRoo's initialisation routine.

6.2 Standing on One Leg

Figure 6.1 shows the GuRoo statically stable while standing on one leg. The most significant difference between the actual and simulated results is that the legs are closer together in the photo on the right. This is caused primarily by the 'sag' in the left hip roll motor. The algorithm was designed with this effect in mind as mentioned in section 5.3. Notice that the feet and top of the torso are parallel to the ground in the right hand image.

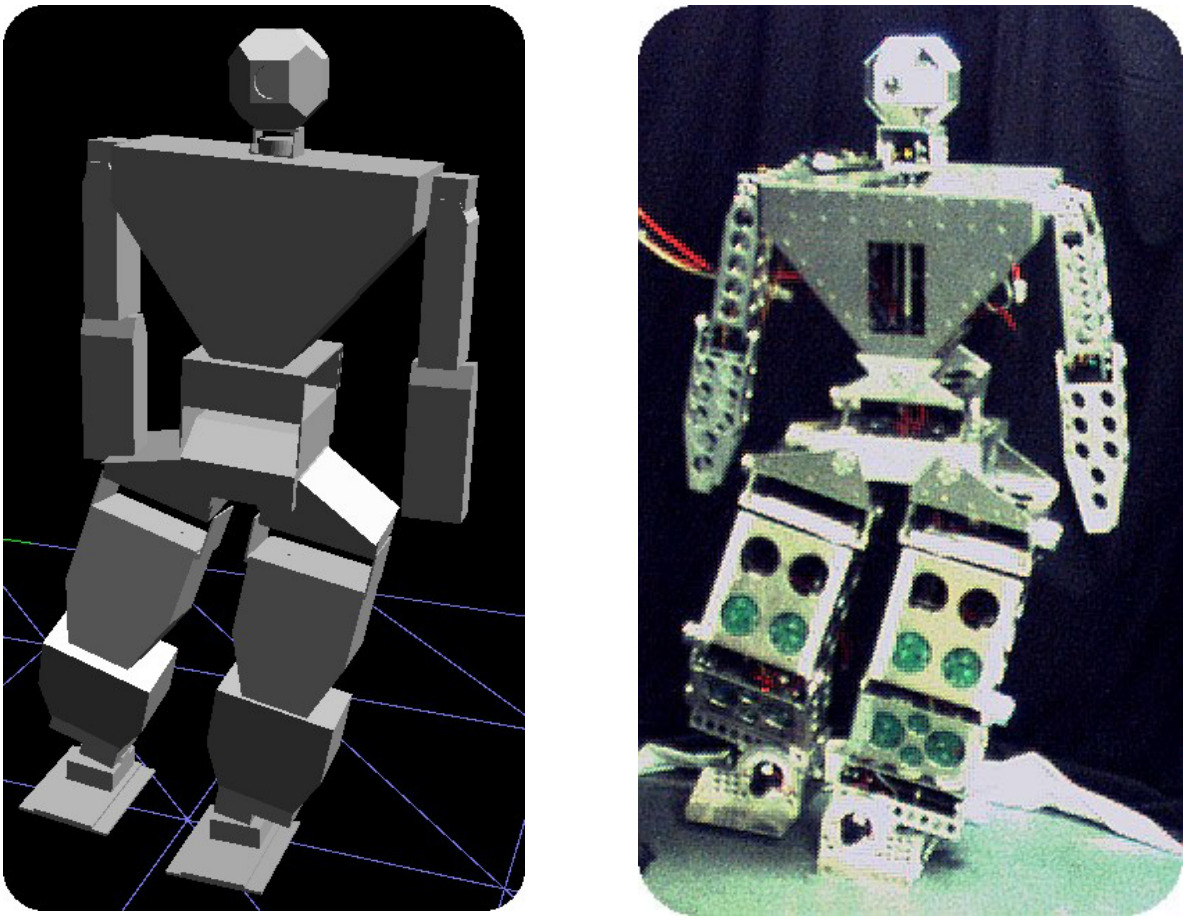


Figure 6.1: Standing on One Leg – Simulated (Left) & Actual (Right)

6.3 Walking

The sequences of images shown in figure 6.2 were taken 200ms apart. The front view incorporates a complete cycle of the walking algorithm with the lateral swaying motion clearly visible. Image 5 is the middle of the double support phase. Note how the hips twist while keeping the torso facing the direction of motion.

The side view illustrates the motion of the swing leg. It can be used to judge the speed of the walking gait. At the end of each step, the robot appears to have moved forward by approximately 1/3 to 1/2 of the foot length. Knowing that the period of the motion is 2s, the robot takes 1 step every second and each foot is 20cm long. This yields a walking speed of between 0.067 and 0.1ms^{-1}, which just meets the specifications. Image 6 in the side view is a problem because it shows instability in the sagittal plane. At this point, the robot leans so far backwards that human assistance is often required. It appears from the images that the problem could be solved by driving the roll motors in the hips or torso forward at this point.

The performance of the control system when walking is detailed in figures 6.3 and 6.4. The graphs on the left are simulated results, while those on the right show data recorded from the actual robot using serial feedback. The most notable difference between the two columns is significantly greater oscillations in the simulation results. The effect is particularly evident on the ankle roll, which exhibits a very unstable response in the simulator. This indicates that the simulation model does not include enough damping in the joints. It also explains why the control was so 'sloppy' when the gains were tuned to the simulated results. It is likely that the control would benefit from a further increase in gains.

Also shown on the ankle roll simulation is the effect of the current limiting. The amplitude of oscillation increases until it reaches the limit at just below 200 encoder counts per control loop. The evidence of backlash is easily seen in the results for the roll motors and not is modelled in the simulator. It is less evident on the pitch joints because the 'squat' performed during initialisation helps reduce the effect as described in section 5.4. Despite these differences, the general form of the simulated results correctly matches the actual data.

The correlation between desired and actual data is much less accurate and reflects the experimental nature of the walking algorithm design. The immediately noticeable feature is that none of the pitch motors reaches their desired velocity. This is caused by a low proportional gain. The response is dominated by the integral compensator, which attempts to minimise the steady-state error by increasing the movement time to maintain the same area beneath both curves. This ensures that the joint reaches its desired position. The pitch motors only move when the leg is not supporting the weight of the robot. Consequently, the ankle tracks the desired velocity better than the knee, which performs better than the hip pitch. This is because each successive joint has to move more of the mass of the leg.

The roll motors are in continual motion and have the best response even though the ankle does not reach its maximum desired velocity. The integral compensator does not get a chance to 'catch-up' the position error before it is eliminated by the error on the negative side. The consequence is that the GuRoo does not sway as far to either side as specified in the algorithm. The hip roll motor performs significantly better than the ankle because it is assisted by the spring and has a smaller load. The ankle has the entire mass of the robot pivoting around it whereas the hip only has the torso.

The yaw joint produces an interesting graph where the actual velocity appears to pre-empt the desired. This can be explained by considering that when the GuRoo takes a step, the forward momentum causes the yaw motor on the support leg to overshoot when twisting the hips. This is shown in the second half of the graph where the area under the actual velocity curve appears larger than the desired.

The robot enters the double support phase before the control loop can correct this error. Friction between the foot and the ground prevents any movement until the weight has been transferred far enough on to the other leg. This corresponds to the point on the graph where the actual velocity first rises suddenly. As the joint resolves the error, it is required to twist further to actuate the swing leg. As before, the extra width on the trailing side is caused by not reaching the peak velocity.

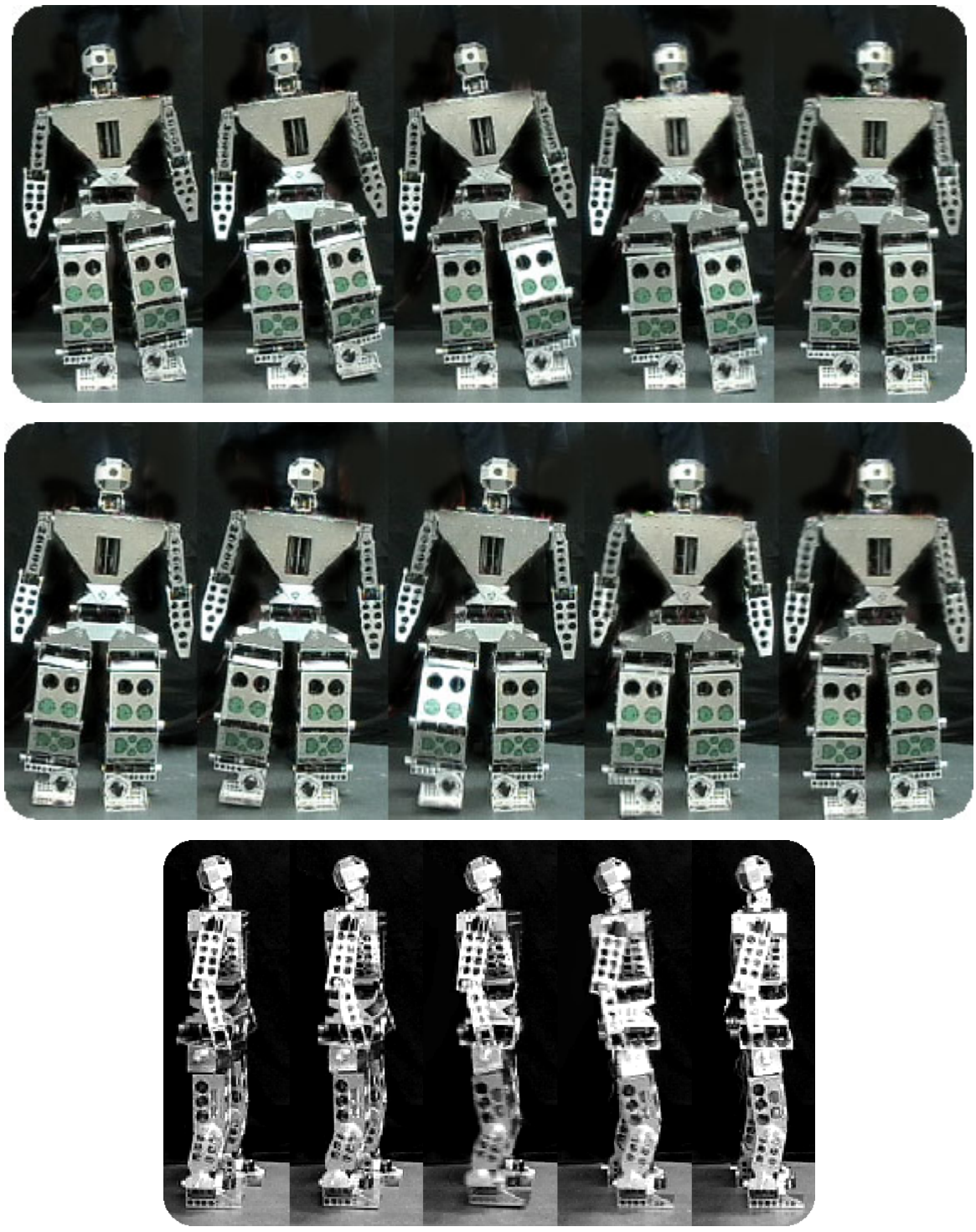


Figure 6.2: Front & Side Views of the GuRoo Walking

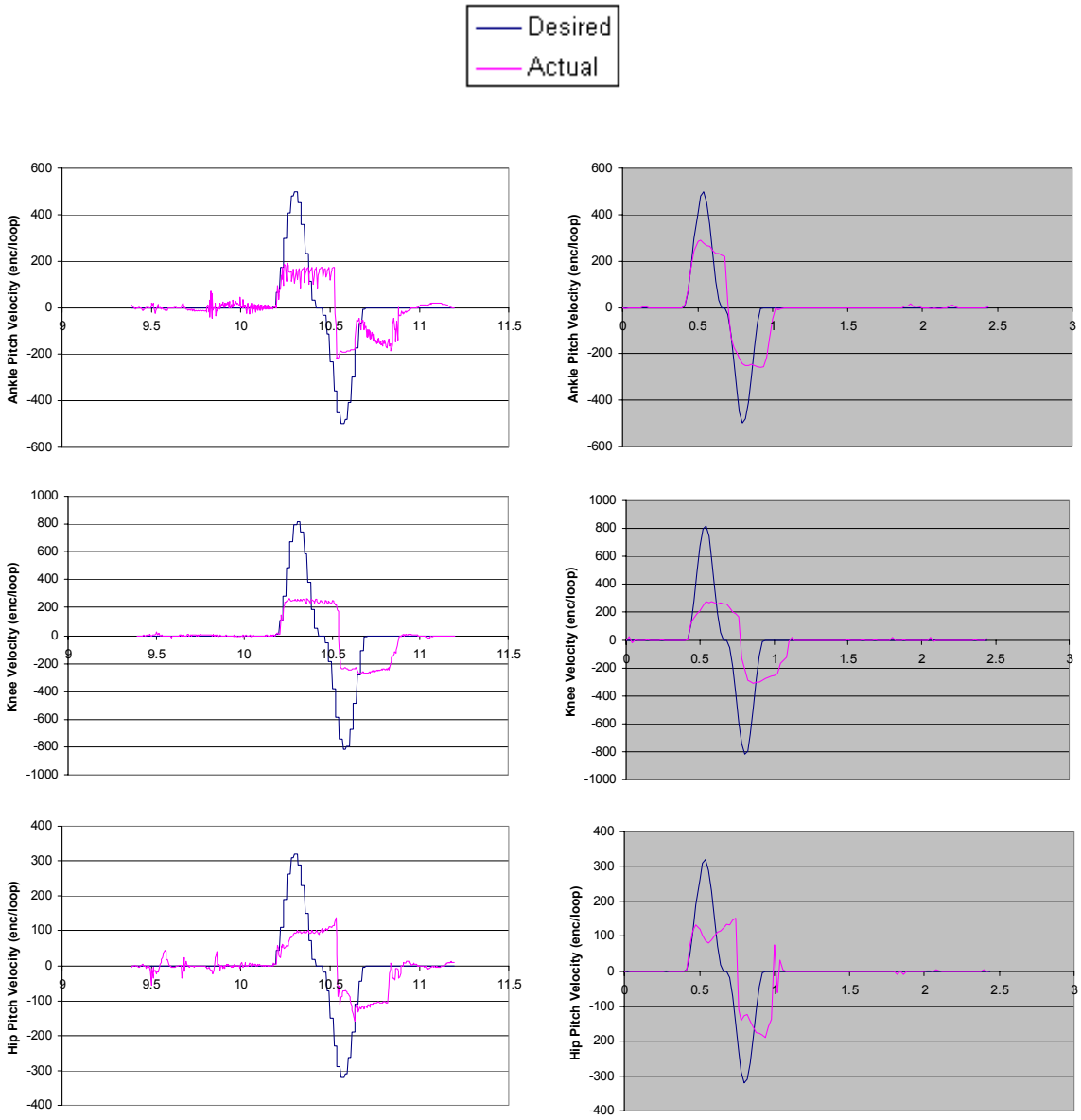


Figure: 6.3: Pitch Trajectories for Walking Algorithm

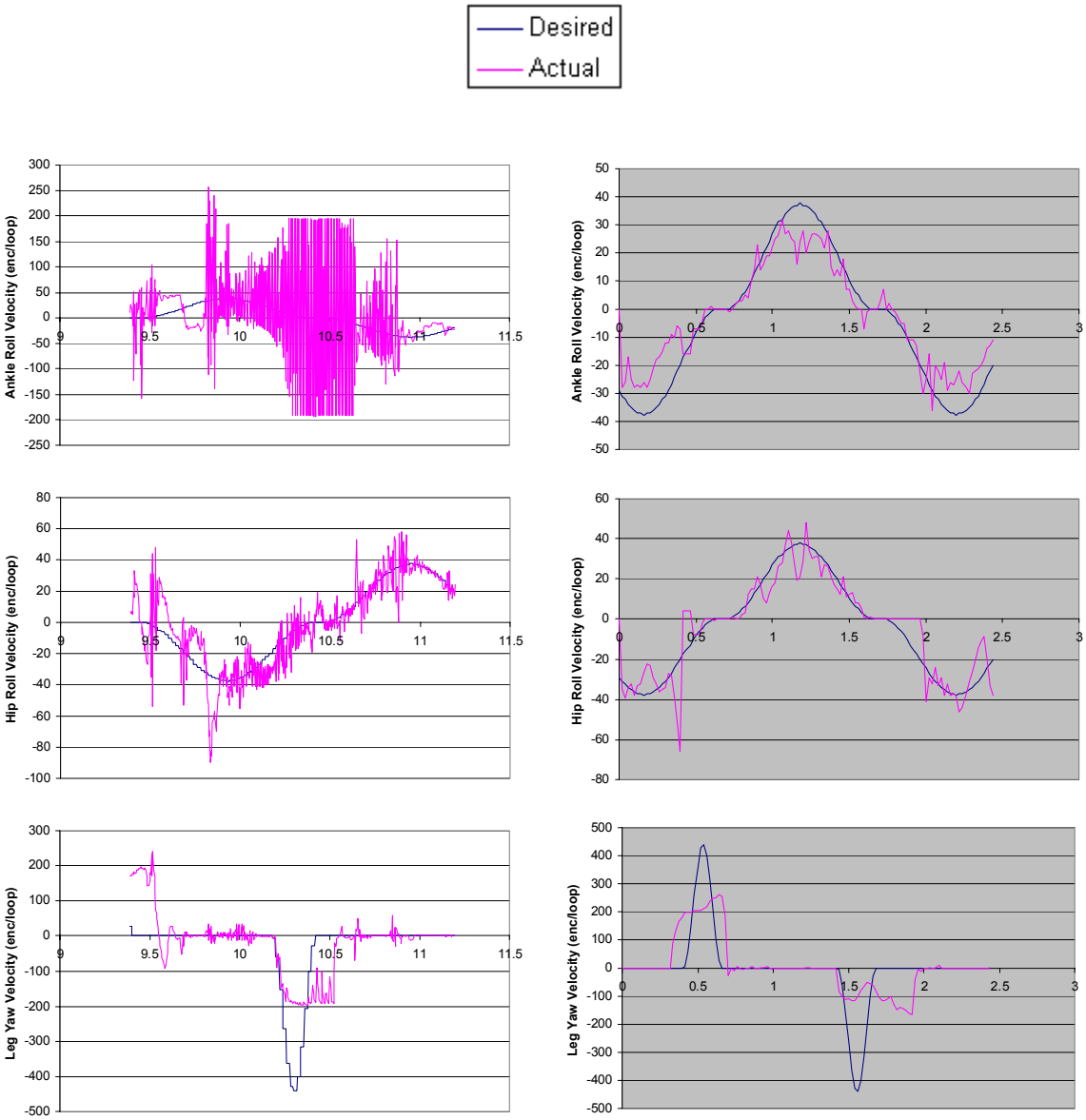


Figure 6.4: Roll & Yaw Trajectories for Walking Algorithm

6.4 RoboCup 2002

The humanoid league is divided into categories based on height because larger robots are much more challenging. GuRoo was the sole competitor in the 1.2m class and there was only one larger robot (180cm – from Sweden). This year however, was the first time the humanoid competition had been held and all robots ended up competing together.

GuRoo demonstrated the best performance of any in the standing on one leg competition by being the only competitor to start from, and return to, a double support position. All other teams manually positioned their robot on one leg and had it stay there for the required 60sec.

The walking competition was slightly less successful. GuRoo was able to walk with some human assistance as shown in figure 6.5. The official distance travelled was 354cm for a placing of 7th overall. The winner was 'Nagara,' an 83cm robot from Japan.

Additionally, there was a 'freestyle' competition where teams demonstrated their robots' capabilities to a panel of judges. UQ placed 7th in this section also, scoring a total of 13.83 points. The highest score was 24.2, achieved by a 25cm robot from southern Denmark. The prize for best humanoid overall was won by 'Nagara.'

Figure 6.5: The GuRoo Walking during RoboCup

CHAPTER 7 – CONCLUSIONS

This chapter summarises the goals of the thesis and the extent to which they were achieved. It also provides suggestions for future development.

7.1 Conclusion

The primary objective was to develop joint control and open loop gait algorithms that allow a humanoid robot to walk unaided at $0.1ms^{-1}$. The project was almost totally successful in achieving this. Some human assistance is required to balance the robot in the sagittal plane, but otherwise, the GuRoo walks dynamically at a speed of between 0.067 and $0.1ms^{-1}$. The GuRoo competed in the 2002 international RoboCup competition against smaller robots and placed 7th in the world for walking ability. A second event at RoboCup required the development of an algorithm to make the robot stand on one leg. GuRoo was the best at performing this action.

Minor goals achieved include:

- Continued development of the UQ humanoid simulator and successful simulation of the above movements,
- Creation of joint control software that is directly portable between the simulator and all controller boards with the exception of an identification number,
- Optimisation of the control software, allowing it to execute at over 1kHz, which is 20 times faster than the specified 50Hz minimum,

7.2 Future Work

The open loop walking gait has been successfully developed and will provide an excellent platform for future research. The obvious potential is to develop a closed loop algorithm using feedback from gyroscopes and force sensors placed on the soles of the feet. Additionally, the serial feedback of joint positions that is currently used for data logging could be incorporated into the gait generation. This would allow calculation of the actual centre of mass for active balance control.

Algorithms for turning corners and kicking a soccer ball were begun this year and could be completed and implemented. This would be particularly effective if combined with further development and integration of the vision system.

The communications between the central controller and the CAN bus needs a complete redesign. Standard serial transmission is slow and unreliable. A high-speed USB to CAN interface is recommended. Central control also needs to be performed on the robot to allow the GuRoo to be completely autonomous.

In addition, a more advanced joint control method is required. This could be aided by utilising the current sensors incorporated in the new hardware design and some of the mathematics from the balance code. Torque or joint stiffness control would be an ideal starting point.

Finally, the simulator model needs updating to increase the accuracy of the simulated results when compared to the actual data - including the incorporation of backlash in the gears.

REFERENCES

[1] Wyeth Kee Wagstaff et al, "Design of an Autonomous Humanoid Robot," presented at Australian Conference on Robotics and Automation, 2001.

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

[3] E. Zelniker, "Joint Control for an Autonomous Humanoid Robot," University of Queensland, 2001.

[4] A. Smith, "Simulator Adaption and Gait Pattern Creation for a Humanoid Robot," University of Queensland, 2001.

[5] S. McMillan, "Computational Dynamics for Robotic Systems on Land and Under Water," Ohio State University, 1995.

[6] J. Stirzaker, "Design of DC Motor Controllers for a Humanoid Robot," University of Queensland, 2001.

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

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

[9] N. S. Nise, *Control Systems Engineering*: John Wiley & Sons, Inc., 2000.

[10] G. F. Wyeth, "Introduction to Control Systems," in *ELEC3500 Lecture Notes, Module 1.*: University of Queensland, 2001.

[11] I. Marshall, "Active Balance Control for a Humanoid Robot," University of Queensland, 2002.

[12] Texas Instruments, "TMS320C2x/C2xx/C5x Optimising C Compiler User's Guide," 1999.

[13] A. Hood, "Distributed Motion Controllers for a Humanoid Robot," University of Queensland, 2002.

APPENDIX A – CONTROL SOFTWARE

Listed below is the low level code for the TMS320F243 DSP's that is directly relevant to the control algorithms.

A.1 Control.h

```
/*************************************************************************
 *  Summary: Gains are defined as amount of bit shifting.
 *************************************************************************/


#ifndef _CONTROL_H_
#define _CONTROL_H_

#define STANDARD_P 0
#define STANDARD_I 0
#define STANDARD_F 1

#define TORSO_TWIST_P 2
#define TORSO_TWIST_I 1
#define TORSO_TWIST_F STANDARD_F

#define TORSO_SIDE_P 2
#define TORSO_SIDE_I 1
#define TORSO_SIDE_F STANDARD_F

#define TORSO_FWD_P 2
#define TORSO_FWD_I 1
#define TORSO_FWD_F STANDARD_F

#define HIP_SIDE_P 2
#define HIP_SIDE_I 2
#define HIP_SIDE_F STANDARD_F

#define HIP_FWD_P 0
#define HIP_FWD_I 2
#define HIP_FWD_F STANDARD_F

#define LEG_TWIST_P 1
#define LEG_TWIST_I 1
#define LEG_TWIST_F STANDARD_F

#define KNEE_P 1
#define KNEE_I 1
#define KNEE_F STANDARD_F

#define ANKLE_FWD_P 3
#define ANKLE_FWD_I 2
#define ANKLE_FWD_F STANDARD_F

#define ANKLE_SIDE_P 3
#define ANKLE_SIDE_I 2
#define ANKLE_SIDE_F STANDARD_F

#endif
```

A.2 Board1.c

```
/*****************************************************************************
 *  Summary: Code to be placed on TMS boards
 *****************************************************************************/

#include "../Common/board1.h"
#include "../Common/humanoid.h"
#include "../Common/jointlim.h"
#include "../Common/control.h"

extern int BOARD;                               /* board ID number                          */

int d_joint_velocity[MOTORS_PER_BOARD];         /* variable collected from the mailbox      */

int old_j_pos[MOTORS_PER_BOARD];                /* data that needs to be stored between     */
long I_err[MOTORS_PER_BOARD];                   /* control loop iterations                  */

int data[4];                                    /* array for sending data back through CAN  */

int P_GAIN[MOTORS_PER_BOARD], I_GAIN[MOTORS_PER_BOARD], F_GAIN[MOTORS_PER_BOARD];

/*
 * b1_control.  Control algorithm for each of the 5 DC controlled TMS boards
 */
void b1_control()
{
    int k, joint_pos, joint_vel, P_err, pwm;
    int back_emf, pwm_lim;

    for (k=0; k < MOTORS_PER_BOARD; k++) {

        joint_pos = (int)(read_enc(k));
        joint_vel = joint_pos - old_j_pos[k];

        /*check for encoder overflow*/
        if (joint_vel > V_OVERFLOW)
            joint_vel -= 0xFFFF;
        else if (joint_vel < -V_OVERFLOW)
            joint_vel += 0xFFFF;

        /*add difference in position*/
        old_j_pos[k] = joint_pos;

        P_err = d_joint_velocity[k] - joint_vel;
        I_err[k] += P_err;

        pwm = (P_err<<P_GAIN[k]) + (I_err[k]<<I_GAIN[k]);

        /* Apply current limiting if neccessary.
        /* Magic numbers assume 38V supply, 2000mA current limit and 250Hz control loop */
        #define KV 0
        #define IR 380

        back_emf = -KV * joint_vel;
        pwm_lim = back_emf + IR;
        if (pwm > pwm_lim)
            pwm = pwm_lim;

        pwm_lim = back_emf - IR;
        if (pwm < pwm_lim)
            pwm = pwm_lim;

        /*data logging*/
        data[k] = joint_vel;
```

```
        set_PWM(k, pwm);
    }
}

/*startup positions and explicit initialisers*/
void init_pos() {
    int k;
    for(k=0; k < MOTORS_PER_BOARD; k++) {
        old_j_pos[k] = read_enc(k);
        d_joint_velocity[k] = 0;
        I_err[k] = 0;
    }
}

/*
 * init_gains. Initialises motor gains - proportional, integral and feedfoward.
 */
void init_gains()
{
    switch (BOARD)
    {
    case 1:
    case 2:
        P_GAIN[0] = ANKLE_FWD_P;
        P_GAIN[1] = ANKLE_SIDE_P;
        P_GAIN[2] = KNEE_P;
        I_GAIN[0] = ANKLE_FWD_I;
        I_GAIN[1] = ANKLE_SIDE_I;
        I_GAIN[2] = KNEE_I;
        F_GAIN[0] = ANKLE_FWD_F;
        F_GAIN[1] = ANKLE_SIDE_F;
        F_GAIN[2] = KNEE_F;
        break;

    case 3:
    case 4:
        P_GAIN[0] = HIP_FWD_P;
        P_GAIN[1] = HIP_SIDE_P;
        P_GAIN[2] = LEG_TWIST_P;
        I_GAIN[0] = HIP_FWD_I;
        I_GAIN[1] = HIP_SIDE_I;
        I_GAIN[2] = LEG_TWIST_I;
        F_GAIN[0] = HIP_FWD_F;
        F_GAIN[1] = HIP_SIDE_F;
        F_GAIN[2] = LEG_TWIST_F;
        break;

    case 5:
        P_GAIN[0] = TORSO_FWD_P;
        P_GAIN[1] = TORSO_SIDE_P;
        P_GAIN[2] = TORSO_TWIST_P;
        I_GAIN[0] = TORSO_FWD_I;
        I_GAIN[1] = TORSO_SIDE_I;
        I_GAIN[2] = TORSO_TWIST_I;
        F_GAIN[0] = TORSO_FWD_F;
        F_GAIN[1] = TORSO_SIDE_F;
        F_GAIN[2] = TORSO_TWIST_F;
        break;
    }
}
```

APPENDIX B – MOVEMENT SOFTWARE

A complete listing of the high-level central control code is presented below. This software generates all the GuRoo's movements including the walking gait.

B.1 Central.h

```
#ifndef _CENTRAL_H_
#define _CENTRAL_H_

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

typedef struct _joint {
     float          start_time;
     float          max_angle;
     float          max_acceleration;
     float          period;
     int            desired_joint_vel;
     bool      finished;
} Joint;

typedef struct _leg {
     Joint          ankle_roll;
     Joint          ankle_pitch;
     Joint          knee;
     Joint          hip_yaw;
     Joint          hip_pitch;
     Joint          hip_roll;
} Leg;

typedef struct _arm {
     Joint          elbow;
     Joint          shoulder_pitch;
     Joint          shoulder_roll;
} Arm;

typedef struct _spine {
     Joint          pitch;
     Joint          roll;
     Joint          yaw;
} Spine;

typedef struct _head {
     Joint          pitch;
     Joint          yaw;
} Head;

typedef struct _movestruct {
     void      (*mctrl_fn)(void);
     float          time;
     float          accel;
     float          duration;
     bool      finished;
     int            direction;
     float          sway_angle;
     bool      sway_flag;
     bool      turn;
     int            sway_no;
     int            step_no;
     Leg            left_leg;
     Leg            right_leg;
     Spine          spine;
     Arm            left_arm;
     Arm            right_arm;
     Head      head;
} Move;

void central_control(void);

void central_init(void);

void Crouch(void);
void crouch_func(void);

void Squat(void);
```

```
void squat_func(void);

void Align(void);
void align_func(void);

void Kill(void);
void kill_func(void);
void kill_servos(void);



void Walk(void);
void walking_alg2(void);

void Torso(void);
void torso_test(void);

void Init_Sway(void);
void init_sway_func(void);
void sway_func(void);
void Sway(void);
void Stop_Walk();
void stop_func();

void Init_Turn(void);
void init_turn_func(void);
void Turn(void);
void turn_func(void);
void Stop_Turn(void);
void stop_turn_func(void);


void Stand_1_Leg();
void stand_func1();

void Bow(void);
void bow_func(void) ;

void WaveToCrowd();
void wave_to_crowd();


void Kick_Ball(void) ;
void kick_func(void) ;

void Demo_Balance(void);
void demo_bal_func(void);


float Velocity(float start_time, float acceleration);
float Velocity2(float per, float stage_time);

// These are the desired joint velocities set by the central controller
extern int*         desired_joint_vel[TOTAL_MOTORS];

// This is the move structure that is currently being executed by the central controller
extern Move         move;

#endif
```

B.2 Central.cpp

```
/*****************************************************************************
 * Copyright 2001, Gordon Wyeth
 *****************************************************************************
 *     File: central.c
 *   Author: Gordon Wyeth
 *  Project: Humanoid 0.1
 *  Created: 26 March 2001
 *  Summary: Generates gait pattern by applying sinusoidal trajectory based on
 *              time taken and angle covered for each joint.  Joint Velocities
 *              are placed in desired_joint_vel which is accessable by the CAN
 *              network every CAN_DELAY seconds
 * Modified: by Andrew Smith 2001
 * Modified: Adam Drury 2002
/*****************************************************************************/

#include "central.h"
#include "can.h"
#include "../Common/board1.h"
#include "jointnum.h"
#include "../Common/humanoid.h"
#include "balance.h"

#include <math.h>
#include <stdio.h>
#include <memory.h>

//#define s_period 0.15
#define s_period 0.25f
#define stage1  (s_period)
#define stage2  (stage1 + s_period)
#define stage3  (stage2 + s_period)
#define stage4  (stage3 + s_period)
#define stage5  (stage4 + s_period)
#define stage6  (stage5 + s_period)
#define stage7  (stage6 + s_period)
#define stage8  (stage7 + s_period)
#define stage9  (stage8 + s_period)
#define stage10 (stage9 + s_period)
#define stage11     (stage10 + s_period)
#define stage12     (stage11 + s_period)

Move move;
int* desired_joint_vel[TOTAL_MOTORS];//the servo data is positions not vels
int direction = 1;
int number = 0 ;
void central_control (void)
{
     move.time += float(CENTRAL_SPEED);
     move.mctrl_fn();
#ifdef BALANCE
          balance_control();
#endif

}

void Crouch(void)
{
     int i;
     Joint* j[3];

     kill_func();

     j[0] = &move.left_leg.ankle_pitch;
     j[1] = &move.left_leg.knee;
     j[2] = &move.left_leg.hip_pitch;

     //Specify maximum angle to move
     (*j[0]).max_angle = float(16 * PI / 180.0);
     (*j[1]).max_angle = float(38 * PI / 180.0);
     (*j[2]).max_angle = float(31 * PI / 180.0);

     //Specify maximum acceleration for each joint
     (*j[0]).max_acceleration = float(0.5);
     (*j[1]).max_acceleration = float(0.5);
     (*j[2]).max_acceleration = float(0.5);

     for(i=0; i<3; i++) {
          (*j[i]).start_time = move.time;
          (*j[i]).finished = false;
          (*j[i]).period = (float)(sqrt((2*PI)/(*j[i]).max_acceleration));
     }

     move.mctrl_fn = crouch_func;
     move.finished = false;
```

```
}

void crouch_func(void) {

     Joint* j[3];
     float vel[3];
     static int dir = 1;
     int i;

     j[0] = &move.left_leg.ankle_pitch;
     j[1] = &move.left_leg.knee;
     j[2] = &move.left_leg.hip_pitch;

     for(i=0; i<3; i++) {
          //Calculate desired velocity
          vel[i] = Velocity((*j[i]).start_time, (*j[i]).max_acceleration) * dir;
          (*j[i]).desired_joint_vel = SRAD2ENC((float)((*j[i]).max_angle * vel[i]));
          // Determine when each motor has reached destination
          if((move.time - (*j[i]).start_time) > (*j[i]).period) {
               //stop and wait for other motors to finish
               (*j[i]).desired_joint_vel = 0;
               (*j[i]).finished = true;
          }
     }

     //Determine if entire movement has finished
     move.finished = (*j[0]).finished && (*j[1]).finished && (*j[2]).finished;

     //Copy the left leg to the right leg
     memcpy (&(move.right_leg), &(move.left_leg), sizeof(Leg));

     //If finished, reverse direction and
     if((move.finished) && (number <= 8)){
          dir = -dir;
          number++ ;
          Crouch();
     } else {
//        Kill();
//        kill_servos();
     }

}

void Squat(void) {
     float ankle_angle = float(6.0 * PI/180.0);
     float hip_angle, knee_angle;
     Joint* j[3];
     int i;

     kill_func();
     hip_angle = (float)(asin(0.172 * sin(ankle_angle) / 0.265));
     knee_angle = ankle_angle + hip_angle;
     hip_angle += (float)(4.0 * PI / 180);

     j[0] = &move.left_leg.ankle_pitch;
     j[1] = &move.left_leg.knee;
     j[2] = &move.left_leg.hip_pitch;

     //Specify maximum angle to move
     j[0]->max_angle = ankle_angle;
     j[1]->max_angle = knee_angle;
     j[2]->max_angle = hip_angle;

     //Specify maximum acceleration for each joint
     j[0]->max_acceleration = 1.0;
     j[1]->max_acceleration = 1.0;
     j[2]->max_acceleration = 1.0;

     for(i=0; i<3; i++) {
          j[i]->start_time = move.time;
          j[i]->finished = false;
          j[i]->period = (float)(sqrt((2*PI)/j[i]->max_acceleration));
     }
     move.mctrl_fn = squat_func;
     move.finished = false;
}

void squat_func(void) {
     Joint* j[3];
     float vel[3];
     int i;

     j[0] = &move.left_leg.ankle_pitch;
     j[1] = &move.left_leg.knee;
     j[2] = &move.left_leg.hip_pitch;

     for(i=0; i<3; i++) {
          //Calculate desired velocity
          vel[i] = Velocity(j[i]->start_time, j[i]->max_acceleration);
          j[i]->desired_joint_vel = SRAD2ENC((float)(j[i]->max_angle * vel[i]));
```

```
          // Determine when each motor has reached destination
          if((move.time - j[i]->start_time) > j[i]->period) {
               //stop and wait for other motors to finish
               j[i]->desired_joint_vel = 0;
               j[i]->finished = true;
          }
     }

     //Determine if entire movement has finished
     move.finished = j[0]->finished && j[1]->finished && j[2]->finished;

     //Copy the left leg to the right leg
     memcpy (&(move.right_leg), &(move.left_leg), sizeof(Leg));
}

void Align(void) {
     float ankle_roll_angle = float(24 * PI/180.0);
     float ankle_pitch_angle = float(-44.0 * PI/180.0);
     Joint* j[2];
     int i;

     kill_func();

     j[0] = &move.left_leg.ankle_roll;
     j[1] = &move.left_leg.ankle_pitch;

     //Specify maximum angle to move
     j[0]->max_angle = ankle_roll_angle;
     j[1]->max_angle = ankle_pitch_angle;

     //Specify maximum acceleration for each joint
     j[0]->max_acceleration = 5.0;
     j[1]->max_acceleration = 5.0;

     for(i=0; i<2; i++) {
          j[i]->start_time = move.time;
          j[i]->finished = false;
          j[i]->period = (float)(sqrt((2*PI)/j[i]->max_acceleration));
     }

     move.mctrl_fn = align_func;
     move.finished = false;
}

void align_func(void) {
     Joint* j[2];
     float vel[2];
     int i;

     j[0] = &move.left_leg.ankle_roll;
     j[1] = &move.left_leg.ankle_pitch;

     for(i=0; i<2; i++) {
          //Calculate desired velocity
          vel[i] = Velocity(j[i]->start_time, j[i]->max_acceleration);
          j[i]->desired_joint_vel = SRAD2ENC((float)(j[i]->max_angle * vel[i]));
          // Determine when each motor has reached destination
          if((move.time - j[i]->start_time) > j[i]->period) {
               //stop and wait for other motors to finish
               j[i]->desired_joint_vel = 0;
               j[i]->finished = true;
          }
     }

     //Determine if entire movement has finished
     move.finished = j[0]->finished && j[1]->finished;

     //Copy the left leg to the right leg
     memcpy (&(move.right_leg), &(move.left_leg), sizeof(Leg));

     //adjust direction of right ankle roll
     move.right_leg.ankle_roll.desired_joint_vel *= -1;
}
//Velocity profile. Generates cosine wave from max_acceleration and start_time
float Velocity(float start_time, float acceleration) {
     float v, per;
     per = (float)(sqrt(2*PI/acceleration));
     v = (float)(BOARD_DELAY*(1/per)*(1 - cos(2*PI*(move.time - start_time)/per)));
     return v;
}

//Velocity profile. Generates cosine wave from period and start time
float Velocity2(float per, float stage_time) {
     return (float)(BOARD_DELAY*(1/per)*(1 - cos(2*PI*(move.time-stage_time)/per)));
}

void Kill(void) {
     kill_servos();
     move.mctrl_fn = kill_func;
}
```

```
void kill_func(void)
{
     for(int i=0; i<TOTAL_MOTORS; i++) {
          *desired_joint_vel[i] = 0;
     }

}

void kill_servos(void) {
     send_servo(0);
}

void Torso(void) {
     kill_func();
     move.time = 0;
     move.finished = false;
     move.mctrl_fn = torso_test;
}

void torso_test(void) {
     float velocity ;
    velocity = Velocity2(s_period * 4.0, 0.0);
/*   velocity = float(0.5 * 2.0 * PI/(2.0 * 4) * sin(2.0 * PI * move.time/(2.0 * 4))) * (float) BOARD_DELAY; */
     *desired_joint_vel[LEFT_ANKLE_FWD]= SRAD2ENC((float) 30 * velocity);

}


void Init_Sway(void) {
     kill_func();
     move.finished = false;
     move.direction = -1 ;
     move.mctrl_fn = init_sway_func;
     move.time = 0;
     move.sway_angle = (float)(2.0 * PI / 180);
     move.sway_flag = false ;
}


void init_sway_func(void) {


     float sway_angle = move.sway_angle ;
     int dir = move.direction ;
     float velocity;

     if (move.time <= stage4) {
        velocity = Velocity2(s_period * 4.0, 0.0);
        *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir *(sway_angle)));
        *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle -2.0 * PI / 180.0)));
        *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity *  dir * (sway_angle )));
        *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity *  dir * (sway_angle + 2.0 * PI / 180.0)));
//        *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * dir * (torso_angle/2)));
     }
     if ((move.time > stage4) && (move.time <= stage8)){
          sway_angle = (float)(6 * PI / 180.0);
          velocity = Velocity2(s_period * 4.0, stage4);
        *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC(   (float)(velocity * -dir * (sway_angle )));
        *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * -dir * (sway_angle )));
        *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * -dir * (sway_angle )));
        *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * -dir * (sway_angle )));
//        *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(-velocity * dir * torso_angle));
     }

     if ((move.time > stage8) && (move.time <= stage12)){
          sway_angle = (float)(8 * PI / 180.0);
          velocity = Velocity2(s_period * 4.0, stage8);
        *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle)));
        *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle)));
        *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle )));
        *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle )));
//        *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * dir * (5.0 * PI / 180));
     }

     if (move.time > stage12) {
          dir = -dir ;
          move.direction = dir ;
          move.sway_angle = sway_angle ;

          move.time = stage9;
          if (move.sway_flag){
               kill_func();
               Sway();
          }
          move.sway_flag = true;
     }

}
```

```
void Sway(void) {
     kill_func();
     move.mctrl_fn = sway_func;
     move.finished = false;
     move.direction = 1 ;
     move.sway_angle = (float)(5.5 * PI/180.0) ;
     move.sway_no = 0;
     move.time = 0;
     if (number == 0) {
          move.step_no = 121;
     }
     if (number == 1){
          move.step_no = 41;
     }
     if (number == 2){
          move.step_no = 121;
     }
}

void sway_func(void){
     #define init_hip_sway         4.5 * PI/180
     #define init_twist_angle 5.0 * PI/180
     #define twist_angle           8.0 * PI/180
     #define s_lift_angle          18.0 * PI/180
     #define torso_angle           4.0 * PI/180

     float velocity;
     int dir = move.direction ;

     float sway_angle = move.sway_angle ;
//   static int sway_number = 0 ;

     float hip_angle = (float)(asin(0.172 * sin(s_lift_angle) / 0.265));
     float knee_angle = (float)(s_lift_angle + hip_angle);


     if ( move.time <= stage4 ) {
          velocity = Velocity2(s_period*4.0, 0);
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle)));
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle)));

          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle )));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle )));
          *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(-velocity * dir * (torso_angle*1.5 )));

          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(-velocity * dir * (torso_angle*0.5  )));
     }

     if ((move.time > stage3) && (move.time <= stage4)) {
          velocity = Velocity2(s_period, stage3);
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * s_lift_angle));
          *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
          *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));


          // Twist component
          *desired_joint_vel[RIGHT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir * -twist_angle));
          *desired_joint_vel[LEFT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir * -twist_angle));
          *desired_joint_vel[TORSO_TWIST] = SRAD2ENC((float)(-velocity * dir * twist_angle));
     }


     if ( (move.time > stage4) && (move.time <= stage8) ) {
          velocity = Velocity2(s_period*4.0, stage4);
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle  )));
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle ) ));

          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * dir * torso_angle));

     }
     if ((move.time > stage4) && (move.time <= stage5)) {             // Lower leg
        velocity = Velocity2(s_period , stage4);
          if (dir == -1) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -s_lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(velocity * 2 * PI / 180);
          }else {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -s_lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(-velocity * 2 * PI / 180);
          }
     }

     if ((move.time > stage5) && (move.time <= stage7)) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC(0);
```

```
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC(0);
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC(0);
     }

     if ((move.time > stage7) && (move.time <= stage8)) {             // Lift leg
        velocity = Velocity2(s_period, stage7);
          if (dir == -1) {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * s_lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));
          } else {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * s_lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));
          }

               // Twist component
          *desired_joint_vel[RIGHT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir  * twist_angle * 2));
          *desired_joint_vel[LEFT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir  * twist_angle * 2));
          *desired_joint_vel[TORSO_TWIST] = SRAD2ENC((float)(-velocity * dir  * -twist_angle * 2));

     }

     if (move.time > stage8) {
          dir = -dir;
          move.direction = dir ;
          move.sway_angle = sway_angle ;
          move.sway_no++ ;

          printf("Step Number : %d  \n" , move.sway_no );
          move.time = stage4;
          if((move.finished) || (move.sway_no >= move.step_no)) {
               Stop_Walk();
          }

     }
}

void Walk(void){
     kill_func();
     Init_Sway();
}

void Stop_Walk() {
     if(!move.finished) {
          move.finished = true;
     } else {
          kill_func();
          move.mctrl_fn = stop_func;
          move.finished = false;
          move.time = 0.0;
     }
}


void stop_func() {
     #define init_hip_sway         4.5 * PI/180
     #define init_twist_angle 5.0 * PI/180
     #define lift_angle            18.0 * PI/180
     #define fudge                 3.0 * PI/180                  // extra 2 degrees from centre

     float velocity;
     int dir = move.direction ;
     float sway_angle = move.sway_angle ;

     float hip_angle = (float)(asin(0.172 * sin(lift_angle) / 0.265));
     float knee_angle = (float)(lift_angle + hip_angle);

     if (move.time <= stage4) {
          velocity = Velocity2(s_period*4.0, 0);
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * sway_angle));
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * dir * torso_angle));
     }

     if (move.time <= stage1) {              // Lower leg
          velocity = Velocity2(s_period , 0);
          if (dir == -1) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(velocity * 1 * PI / 180);
          }else {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
```

```
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(-velocity * 1 * PI / 180);
          }
     }

     if ((move.time > stage1) && (move.time <= stage3)) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC(0);
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC(0);
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC(0);
     }

     if ((move.time > stage3) && (move.time <= stage4)) {             // Lift leg + half twist
        velocity = Velocity2(s_period, stage3);
          if (dir == -1) {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));
          } else {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));
          }

               // Twist component
          *desired_joint_vel[RIGHT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir *  twist_angle));
          *desired_joint_vel[LEFT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir * twist_angle));
          *desired_joint_vel[TORSO_TWIST] = SRAD2ENC((float)(-velocity * dir *  -twist_angle));

     }

     if ((move.time > stage4) && (move.time <= stage5)) {             // Lower leg
        velocity = Velocity2(s_period , stage4);
          if (dir == 1) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(velocity * 1 * PI / 180);
          }else {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(-velocity * 1 * PI / 180);
          }
     }


     if ((move.time > stage4) && (move.time <= stage6)) {
          velocity = Velocity2(s_period*4.0, stage4);
          *desired_joint_vel[LEFT_HIP_SIDE] =          SRAD2ENC((float)( velocity * dir * (sway_angle + 2.1 * PI/180)));
          *desired_joint_vel[RIGHT_HIP_SIDE]  =   SRAD2ENC((float)(-velocity * dir * (sway_angle - 2.1 * PI/180)));

          *desired_joint_vel[LEFT_ANKLE_SIDE] =   SRAD2ENC((float)(-velocity * dir * (sway_angle)));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] =  SRAD2ENC((float)(-velocity * dir * (sway_angle)));
          *desired_joint_vel[TORSO_FWD] =              SRAD2ENC((float)(velocity  *  (torso_angle*2.8 )));
          *desired_joint_vel[TORSO_SIDE] =        SRAD2ENC((float)(-velocity  * (torso_angle*1.1 )));
     }

     if (move.time > stage6) {
          Kill();
          kill_servos();
          move.sway_angle = (float)(2.0 * PI / 180.0) ;
          move.sway_flag = false ;
          kill_func();
          move.turn = true;

          Init_Turn();


     }
}
void Stand_1_Leg() {
     kill_func();
     kill_servos();
     move.mctrl_fn = stand_func1;
     move.finished = false;
     move.duration = 20.0;
     move.time = 0;
}

void stand_func1()
{
     #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;
     static int side = -1; //-1 = left, +1 = right

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

     //shift weight over left foot
     if ((move.time > STAGE1) && (move.time <= STAGE2)) {
        velocity = Velocity2(s_per, STAGE1);
          *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)) ));
          *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))));
     }

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

     }

     //hold for specified time
     if ((move.time > STAGE3) && (move.time <= STAGE4)) {
          kill_func();
//        send_servo(2);
     }

     //lower left leg
     if ((move.time > STAGE4) && (move.time <= STAGE5))
     {
          velocity = Velocity2(s_per, STAGE4);
          *desired_joint_vel[RIGHT_HIP_FWD]  = SRAD2ENC((float)(-velocity * HIP_ANGLE));
          *desired_joint_vel[RIGHT_KNEE]  = SRAD2ENC((float)(-velocity * KNEE_ANGLE));
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(-velocity * ANKLE_ANGLE));
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(-velocity * (5.0 * PI / 180)));
          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * (3.0 * PI / 180)));
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * (3.0 * PI / 180)));
          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * torso_correction));
          *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(velocity * 5.0 * PI / 180));
     }

     //shift back to centre
     if ((move.time > STAGE5) && (move.time <= STAGE6)) {
        velocity = Velocity2(s_per, STAGE5);
          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * (stand_lean_angle - 1*PI/180)));
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * (stand_lean_angle - 5*PI/180)));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * stand_lean_angle));
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * (stand_lean_angle + (7.5*PI/180))));
     }

     if ((move.time > STAGE6) && (move.time <= STAGE7)) {
          velocity = Velocity2(s_per, STAGE6);
          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(-velocity * torso_correction));
     }

     //finish
     if (move.time > STAGE7) {
          Kill();
          //kill_servos();
          Crouch();
     }
}

void central_init (void)
{
     desired_joint_vel[LEFT_ANKLE_SIDE] = &(move.left_leg.ankle_roll.desired_joint_vel);
     desired_joint_vel[LEFT_ANKLE_FWD]  = &(move.left_leg.ankle_pitch.desired_joint_vel);
     desired_joint_vel[LEFT_KNEE]       = &(move.left_leg.knee.desired_joint_vel);

     desired_joint_vel[RIGHT_ANKLE_SIDE]     = &(move.right_leg.ankle_roll.desired_joint_vel);
     desired_joint_vel[RIGHT_ANKLE_FWD] = &(move.right_leg.ankle_pitch.desired_joint_vel);
     desired_joint_vel[RIGHT_KNEE]      = &(move.right_leg.knee.desired_joint_vel);

     desired_joint_vel[LEFT_HIP_FWD]         = &(move.left_leg.hip_pitch.desired_joint_vel);
     desired_joint_vel[LEFT_HIP_SIDE]   = &(move.left_leg.hip_roll.desired_joint_vel);
```

```
     desired_joint_vel[LEFT_LEG_TWIST]  = &(move.left_leg.hip_yaw.desired_joint_vel);

     desired_joint_vel[RIGHT_HIP_FWD]   = &(move.right_leg.hip_pitch.desired_joint_vel);
     desired_joint_vel[RIGHT_HIP_SIDE]  = &(move.right_leg.hip_roll.desired_joint_vel);
     desired_joint_vel[RIGHT_LEG_TWIST] = &(move.right_leg.hip_yaw.desired_joint_vel);

     desired_joint_vel[TORSO_FWD]       = &(move.spine.pitch.desired_joint_vel);
     desired_joint_vel[TORSO_SIDE]      = &(move.spine.roll.desired_joint_vel);
     desired_joint_vel[TORSO_TWIST]          = &(move.spine.yaw.desired_joint_vel);

     desired_joint_vel[LEFT_SHOULDER]   = &(move.left_arm.shoulder_pitch.desired_joint_vel);
     desired_joint_vel[LEFT_UPPER_ARM]  = &(move.left_arm.shoulder_roll.desired_joint_vel);
     desired_joint_vel[LEFT_LOWER_ARM]  = &(move.left_arm.elbow.desired_joint_vel);
     desired_joint_vel[RIGHT_SHOULDER]  = &(move.right_arm.shoulder_pitch.desired_joint_vel);
     desired_joint_vel[RIGHT_UPPER_ARM] = &(move.right_arm.shoulder_roll.desired_joint_vel);
     desired_joint_vel[RIGHT_LOWER_ARM] = &(move.right_arm.elbow.desired_joint_vel);

     desired_joint_vel[NECK]                 = &(move.head.yaw.desired_joint_vel);
     desired_joint_vel[HEAD]                 = &(move.head.pitch.desired_joint_vel);

     Kill();
     move.finished = true;
}



void Kick_Ball() {
     kill_func();
     move.mctrl_fn = kick_func;
     move.finished = false;
     move.duration = 60.0;
     move.time = 0;
}

void kick_func()
{
     #define per 3.0f
     #define kick_per 1.0f
     #define lean_angle 12.5 * PI/180

     #define KSTAGE1 1.0f
     #define   KSTAGE2 (KSTAGE1 + per)
     #define KSTAGE3 (KSTAGE2 + per)
     #define KSTAGE4 (KSTAGE3 + per)
     #define KSTAGE5 (KSTAGE4 + kick_per)
     #define KSTAGE6 (KSTAGE5 + per)
     #define KSTAGE7 (KSTAGE6 + per)
     #define KSTAGE8 (KSTAGE7 + per)
     #define KSTAGE9 (KSTAGE8 + per)
     #define KSTAGE10 (KSTAGE9 + per)



     float ANKLE_ANGLE = (float)(30.0 * PI/180);
     float HIP_ANGLE = (float)(asin(0.172 * sin(lift_angle) / 0.265));
     float KNEE_ANGLE = (float)(ANKLE_ANGLE + HIP_ANGLE);
     float TORSO_ANGLE = (float)(5.0 * PI / 180);
     float torso_correction = 15 * PI / 180 ;


     float velocity;
     static int side = -1; //-1 = left, +1 = right

     //hold position
     if(move.time <= KSTAGE1) {
     }

     //shift weight over left foot
     if ((move.time > KSTAGE1) && (move.time <= KSTAGE2)) {
        velocity = Velocity2(per, KSTAGE1);
          *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)) ));
          *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))));
     }

     //raise right leg
     if ((move.time > KSTAGE2) && (move.time <= KSTAGE3))
     {    velocity = Velocity2(per, KSTAGE2);
          *desired_joint_vel[RIGHT_HIP_FWD]  = SRAD2ENC((float)(velocity * HIP_ANGLE));
          *desired_joint_vel[RIGHT_KNEE]  = SRAD2ENC((float)(velocity * KNEE_ANGLE));
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * ANKLE_ANGLE));
          *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(-velocity * 3.0 * PI / 180));
     }

     //swing back right leg
     if ((move.time > KSTAGE3) && (move.time <= KSTAGE4)) {
          velocity = Velocity2(per, KSTAGE3);
          *desired_joint_vel[RIGHT_HIP_FWD]  = SRAD2ENC((float)(-velocity * 5*PI/180));
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * 5*PI/180));
```

```
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(-velocity * (3.0 * PI / 180)));
          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * (2.0 * PI / 180)));

          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(-velocity * torso_correction));
          *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(velocity * TORSO_ANGLE));
     }


     //swing through right leg
     if ((move.time > KSTAGE4) && (move.time <= KSTAGE5)) {
          velocity = Velocity2(kick_per, KSTAGE4);
          *desired_joint_vel[RIGHT_HIP_FWD]  = SRAD2ENC((float)(velocity * 20*PI/180));
//        *desired_joint_vel[RIGHT_KNEE]  = SRAD2ENC((float)(-velocity * KNEE_ANGLE));
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(-velocity * 20*PI/180));
          *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(-velocity * TORSO_ANGLE));
     }

     //bring leg back under
     if ((move.time > KSTAGE5) && (move.time <= KSTAGE6)) {
          velocity = Velocity2(per, KSTAGE5);
          *desired_joint_vel[RIGHT_HIP_FWD]  = SRAD2ENC((float)(-velocity * 15*PI/180));
//        *desired_joint_vel[RIGHT_KNEE]  = SRAD2ENC((float)(velocity * KNEE_ANGLE));
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * 15*PI/180));
     }


     //lower left leg
     if ((move.time > KSTAGE6) && (move.time <= KSTAGE7))
     {
          velocity = Velocity2(per, KSTAGE6);
          *desired_joint_vel[RIGHT_HIP_FWD]  = SRAD2ENC((float)(-velocity * HIP_ANGLE));
          *desired_joint_vel[RIGHT_KNEE]  = SRAD2ENC((float)(-velocity * KNEE_ANGLE));
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(-velocity * ANKLE_ANGLE));
          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * torso_correction * 2));
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * (3.0 * PI / 180)));
     }

     //shift back to centre
     if ((move.time > KSTAGE7) && (move.time <= KSTAGE8)) {
        velocity = Velocity2(per, KSTAGE7);
          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * (stand_lean_angle - 0*PI/180)));
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * (stand_lean_angle - 2.5*PI/180)));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * stand_lean_angle));
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * (stand_lean_angle + (7.5*PI/180))));
     }


     if ((move.time > KSTAGE8) && (move.time <= KSTAGE9)) {
          velocity = Velocity2(per, KSTAGE6);
          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(-velocity * torso_correction));
          *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(-velocity * 3*PI/180));
     }

     //finish
     if (move.time > KSTAGE9) {
          Kill();
          kill_servos();
     }
}


void Bow() {
     kill_func();
     move.mctrl_fn = bow_func;
     move.finished = false;
     move.duration = 60.0;
     move.time = 0;
}

void bow_func() {

     #define per 3.0f

     #define BSTAGE1 1.0f
     #define   BSTAGE2 (BSTAGE1 + per)
     #define BSTAGE3 (BSTAGE2 + per)
     #define BSTAGE4 (BSTAGE3 + per)
     #define BSTAGE5 (BSTAGE4 + 3*per)

     float velocity;
     float bow_angle  = (float)(5.0 * PI / 180.0) ;


     if(move.time <= BSTAGE1) {
     }

     //bow fwd
     if ((move.time > BSTAGE1) && (move.time <= BSTAGE2)) {
        velocity = Velocity2(per, BSTAGE1);
          *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(-velocity * bow_angle/2));
```

```
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(-velocity * bow_angle/2));
          *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * bow_angle));
          *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * bow_angle));
          *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(velocity * bow_angle));
     }

     //bow back
     if ((move.time > BSTAGE2) && (move.time <= BSTAGE3)) {
        velocity = Velocity2(per, BSTAGE2);
          *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * bow_angle/2));
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * bow_angle/2));
          *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(-velocity * bow_angle));
          *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(-velocity * bow_angle));
          *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(-velocity * bow_angle));
     }

     if (move.time > BSTAGE3) {
          Kill();
          kill_servos();
          send_servo(1);
          send_servo(0);
     }

}


void WaveToCrowd() {
     kill_func();
     kill_servos();
     send_servo(2);
     move.mctrl_fn = wave_to_crowd ;
     move.time = 0;
}

void wave_to_crowd() {

     #define wave_per 5.0f
     #define WSTAGE1 0.0f
     #define WSTAGE2 (WSTAGE1 + wave_per)
     #define WSTAGE3 (WSTAGE2 + wave_per)
     #define WSTAGE4 (WSTAGE3 + wave_per)

     float turn_angle = (float)(25.0 * PI / 180.0) ;
     float velocity;

     kill_servos();

     //Turn right and wave
     if ((move.time > WSTAGE1) && (move.time <= WSTAGE2)) {
        velocity = Velocity2(wave_per, WSTAGE1);
          *desired_joint_vel[TORSO_TWIST] = SRAD2ENC((float)(-velocity * turn_angle));
     }

     //bow back
     if ((move.time > WSTAGE2) && (move.time <= WSTAGE3)) {
        velocity = Velocity2(wave_per, WSTAGE2);
          *desired_joint_vel[TORSO_TWIST] = SRAD2ENC((float)(velocity * turn_angle * 2));
     }

     if ((move.time > WSTAGE3) && (move.time <= WSTAGE4)) {
        velocity = Velocity2(wave_per, WSTAGE3);
          *desired_joint_vel[TORSO_TWIST] = SRAD2ENC((float)(-velocity * turn_angle));
     }


     if (move.time > WSTAGE4) {
//        Kill();
          Stand_1_Leg();
     }
}


void Init_Turn(void) {
     kill_func();
     move.finished = false;
     move.direction = -1 ;
     move.mctrl_fn = init_turn_func;
     move.time = 0;
     move.sway_angle = (float)(2.0 * PI / 180);
     move.sway_flag = false ;
     move.turn = true;
}



void init_turn_func(void) {

     float sway_angle = move.sway_angle ;
     int dir = move.direction ;
     float velocity;
```

```
     if (move.time <= stage4) {
        velocity = Velocity2(s_period * 4.0, 0.0);
        *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir *(sway_angle)));
        *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle -2.0 * PI / 180.0)));
        *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity *  dir * (sway_angle )));
        *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity *  dir * (sway_angle + 2.0 * PI / 180.0)));
//        *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * dir * (torso_angle/2)));
     }
     if ((move.time > stage4) && (move.time <= 8)){
          sway_angle = (float)(6 * PI / 180.0);
          velocity = Velocity2(s_period * 4.0, stage4);
        *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC(   (float)(velocity * -dir * (sway_angle )));
        *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * -dir * (sway_angle )));
        *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * -dir * (sway_angle )));
        *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * -dir * (sway_angle )));
//        *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(-velocity * dir * torso_angle));
     }

     if ((move.time > stage8) && (move.time <= stage12)){
          sway_angle = (float)(8 * PI / 180.0);
          velocity = Velocity2(s_period * 4.0, stage8);
        *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle)));
        *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle)));
        *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle )));
        *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle )));
//        *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * dir * (5.0 * PI / 180));
     }

     if (move.time > stage12) {
          dir = -dir ;
          move.direction = dir ;
          move.sway_angle = sway_angle ;

          move.time = stage9;
          if (move.sway_flag){
               kill_func();
               Turn();
          }
          move.sway_flag = true;
     }

}

void Turn(void) {
     kill_func();
     move.mctrl_fn = turn_func;
     move.finished = false;
     move.direction = 1 ;
     move.sway_angle = (float)(5.5 * PI/180.0) ;
     move.sway_no = 0;
     move.time = 0;
     send_servo(3);
}

void turn_func() {
     #define init_hip_sway         4.5 * PI/180
     #define init_twist_angle 5.0 * PI/180
     #define twist_angle           8.0 * PI/180
     #define s_lift_angle          18.0 * PI/180
     #define torso_angle           4.0 * PI/180

     float velocity;
     int dir = move.direction ;

     float sway_angle = move.sway_angle ;
//   static int sway_number = 0 ;

     float hip_angle = (float)(asin(0.172 * sin(s_lift_angle) / 0.265));
     float knee_angle = (float)(s_lift_angle + hip_angle);


     if ( move.time <= stage4 ) {
          velocity = Velocity2(s_period*4.0, 0);
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle)));
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle)));

          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle )));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle )));
//        *desired_joint_vel[TORSO_FWD] = SRAD2ENC((float)(-velocity * dir * (torso_angle*1.5 )));

          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(-velocity * dir * (torso_angle*0.5  )));
     }

     if ((move.time > stage3) && (move.time <= stage4)) {
          velocity = Velocity2(s_period, stage3);
          *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * s_lift_angle));
          *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
          *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));
```

```
          // Twist component
          *desired_joint_vel[LEFT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir * -twist_angle));
     }


     if ( (move.time > stage4) && (move.time <= stage8) ) {
          velocity = Velocity2(s_period*4.0, stage4);
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * (sway_angle  )));
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * dir * (sway_angle ) ));

          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * dir * torso_angle));

     }


     if ((move.time > stage4) && (move.time <= stage5)) {             // Lower leg
        velocity = Velocity2(s_period , stage4);
          if (dir == -1) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -s_lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(velocity * 2 * PI / 180);
          }else {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -s_lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(-velocity * 2 * PI / 180);
          }
     }

     if ((move.time > stage5) && (move.time <= stage7)) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC(0);
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC(0);
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC(0);
     }

     if ((move.time > stage7) && (move.time <= stage8)) {             // Lift leg
        velocity = Velocity2(s_period, stage7);
          if (dir == -1) {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * s_lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));
          } else {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * s_lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));
          }

               // Twist component
//        *desired_joint_vel[RIGHT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir  * twist_angle * 2));
          *desired_joint_vel[LEFT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir  * twist_angle * 2));
//        *desired_joint_vel[TORSO_TWIST] = SRAD2ENC((float)(-velocity * dir  * -twist_angle * 2));

     }

     if (move.time > stage8) {
          dir = -dir;
          move.direction = dir ;
          move.sway_angle = sway_angle ;
          move.sway_no++ ;

          printf("Turn Number : %d  \n" , move.sway_no );
          move.time = stage4;
          if((move.finished) || (move.sway_no >= 11)) {
               direction = dir;
               Stop_Turn();
          }
     }
}



void Stop_Turn() {
     if(!move.finished) {
          move.finished = true;
     } else {
          kill_func();
          move.mctrl_fn = stop_turn_func;
          move.finished = false;
          move.time = 0.0;
     }
}
```

```
void stop_turn_func() {
     #define init_hip_sway         4.5 * PI/180
     #define init_twist_angle 5.0 * PI/180
     #define lift_angle            18.0 * PI/180
     #define fudge                 3.0 * PI/180                  // extra 2 degrees from centre

     float velocity;
     int dir = move.direction ;
     float sway_angle = move.sway_angle ;

     float hip_angle = (float)(asin(0.172 * sin(lift_angle) / 0.265));
     float knee_angle = (float)(lift_angle + hip_angle);

     if (move.time <= stage4) {
          velocity = Velocity2(s_period*4.0, 0);
          *desired_joint_vel[LEFT_HIP_SIDE] = SRAD2ENC((float)(-velocity * dir * sway_angle));
          *desired_joint_vel[RIGHT_HIP_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[LEFT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] = SRAD2ENC((float)(velocity * dir * sway_angle));
          *desired_joint_vel[TORSO_SIDE] = SRAD2ENC((float)(velocity * dir * torso_angle));
     }

     if (move.time <= stage1) {              // Lower leg
          velocity = Velocity2(s_period , 0);
          if (dir == -1) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(velocity * 1 * PI / 180);
          }else {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(-velocity * 1 * PI / 180);
          }
     }

     if ((move.time > stage1) && (move.time <= stage3)) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC(0);
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC(0);
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC(0);
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC(0);
     }

     if ((move.time > stage3) && (move.time <= stage4)) {             // Lift leg + half twist
        velocity = Velocity2(s_period, stage3);
          if (dir == -1) {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));
          } else {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * hip_angle));
          }

               // Twist component
//        *desired_joint_vel[RIGHT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir *  twist_angle));
          *desired_joint_vel[LEFT_LEG_TWIST] = SRAD2ENC((float)(-velocity * dir * twist_angle));
//        *desired_joint_vel[TORSO_TWIST] = SRAD2ENC((float)(-velocity * dir *  -twist_angle));

     }

     if ((move.time > stage4) && (move.time <= stage5)) {             // Lower leg
        velocity = Velocity2(s_period , stage4);
          if (dir == 1) {
               *desired_joint_vel[LEFT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -lift_angle));
               *desired_joint_vel[LEFT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[LEFT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(velocity * 1 * PI / 180);
          }else {
               *desired_joint_vel[RIGHT_ANKLE_FWD] = SRAD2ENC((float)(velocity * -lift_angle));
               *desired_joint_vel[RIGHT_KNEE] = SRAD2ENC((float)(velocity *  -knee_angle));
               *desired_joint_vel[RIGHT_HIP_FWD] = SRAD2ENC((float)(velocity * -hip_angle));
//             *desired_joint_vel[TORSO_FWD] = SRAD2ENC(-velocity * 1 * PI / 180);
          }
     }


     if ((move.time > stage4) && (move.time <= stage6)) {
          velocity = Velocity2(s_period*4.0, stage4);
          *desired_joint_vel[LEFT_HIP_SIDE] =          SRAD2ENC((float)( velocity * dir * (sway_angle + 5.1 * PI/180)));
          *desired_joint_vel[RIGHT_HIP_SIDE]  =   SRAD2ENC((float)(-velocity * dir * (sway_angle - 2.1 * PI/180)));

          *desired_joint_vel[LEFT_ANKLE_SIDE] =   SRAD2ENC((float)(-velocity * dir * (sway_angle)));
          *desired_joint_vel[RIGHT_ANKLE_SIDE] =  SRAD2ENC((float)(-velocity * dir * (sway_angle)));
//        *desired_joint_vel[TORSO_FWD] =              SRAD2ENC((float)(velocity  *  (torso_angle*2.8 )));
```

```
          *desired_joint_vel[TORSO_SIDE] =        SRAD2ENC((float)(-velocity  * (torso_angle*1.1 )));
     }

     if (move.time > stage6) {
          Kill();
          kill_servos();
          move.sway_angle = (float)(2.0 * PI / 180.0) ;
          move.sway_flag = false ;
          number++ ;
          kill_func();
          WaveToCrowd();
     }
}
```

APPENDIX C – MATLAB CODE

The following MATLAB function was used to evaluate the effectiveness of the control algorithms.

A.1 Graph.m

```
%
%   Graph.m
%   Plots data for specified motor numbers and calculates
%   RMS of error between desired and actual trajectories.
%   Input argument is a vector containing motor numbers (0-14)
%

function graph(motors)
    close all;
    M = CSVREAD('log_data_crouch0.csv', 1, 0);
    time = M(:,1);

    for n = 1:length(motors)
        k = motors(n);
        i = 3*k + 2;

        d_vel = M(:,i);
        vel = M(:,i+1);
        j_input = M(:,i+2);

        d_pos(1) = d_vel(1);
        for j = 2:length(d_vel)
            d_pos(j) = d_pos(j-1) + d_vel(j);
        end

        pos(1) = vel(1);
        for j = 2:length(vel)
            pos(j) = pos(j-1) + vel(j);
        end

        figure;
        subplot(2,1,1);
        plot(time, vel, 'r');
        hold on;
        plot(time, d_vel, 'b');
        title(strcat(joint(k), ': Velocity'));
        xlabel('Time (s)');
        ylabel('Velocity (Enc/Loop)');
        legend('Actual', 'Desired');

        subplot(2,1,2);
        plot(time, pos, 'r');
        hold on;
        plot(time, d_pos, 'b');
        title(strcat(joint(k), ': Position'));
        xlabel('Time (s)');
        ylabel('Position (Encoder Counts)');
        legend('Actual', 'Desired');
        hold off;
```

```
        vel_err = d_vel - vel;
        pos_err = d_pos - pos;

        SUMSQ_vel = sum(vel_err.^2);
        SUMSQ_pos = sum(pos_err.^2);
        RMS_vel = sqrt(SUMSQ_vel/length(vel_err))
        RMS_pos = sqrt(SUMSQ_pos/length(pos_err))

    end

return;

function name = joint(i)
    switch i;
        case 0;
            name = 'LEFT ANKLE FWD';
        case 1;
            name = 'LEFT ANKLE SIDE';
        case 2;
            name = 'LEFT KNEE';
        case 3;
            name = 'RIGHT ANKLE FWD';
        case 4;
            name = 'RIGHT ANKLE SIDE';
        case 5;
            name = 'RIGHT KNEE';
        case 6;
            name = 'LEFT HIP FWD';
        case 7;
            name = 'LEFT HIP SIDE';
        case 8;
            name = 'LEFT LEG TWIST';
        case 9;
            name = 'RIGHT HIP FWD';
        case 10;
            name = 'RIGHT HIP SIDE';
        case 11;
            name = 'RIGHT LEG TWIST';
        case 12;
            name = 'TORSO FWD';
        case 13;
            name = 'TORSO SIDE';
        case 14;
            name = 'TORSO TWIST';
    end;
return;
```