THE UNIVERSITY OF QUEENSLAND

Gait Generation for a Humanoid Robot

By

Tim Pike

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

Submitted for the degree of Bachelor of Engineering (Honours) in the division of Software Engineering.

October 2003

Tim Pike
Brisbane, QLD 4067

29/10/2003

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

Dear Professor Kaplan,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Software Engineering, I present the following thesis entitled "*Gait Generation for a Humanoid Robot*". The work was performed under the supervision of Dr Gordon Wyeth.
I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Tim Pike

Acknowledgements

This thesis would not have been possible without the help of the following people:

- Gordon Wyeth, for allowing me the chance to do this thesis and to work on the GuRoo project.

- Damien Kee, for all his help and advice while working on this thesis. Also for all his hard work continually keeping GuRoo operational.

- Jonathon Roberts, from the CSIRO, for showing me the GAlib library and for all his help in not just getting my stuff to work, but for all the work he did on the GuRoo project.

- All the talented people who were either working on the GuRoo project this year or have previously worked on it.

- Anyone else I forgot to mention here.

Abstract

This thesis details the continued work done on the GuRoo humanoid robot developed at the University of Queensland. It examines the work done towards giving the robot the ability to generate a gait capable of moving the robot in a forward direction in a straight line.

The aim of this thesis is therefore to develop the gait generation algorithms, including the development of straight line walking inside that algorithm, testing inside a simulated environment and testing on actual hardware.

The primary achievement of this thesis was the development and implementation of a gait generation algorithm that allowed GuRoo to walk in a straight line. The gait generation algorithm developed was a static algorithm that is parameterised to allow the easy changing of aspects of the walk, i.e. step size etc. This allows a much more dynamic and flexible system. Also the algorithm was developed with a number of changeable variables in mind, that currently are fine tuned using a genetic algorithm, but can easily be modified to be changed dynamically using feedback from the actual robot.

Also among the achievements of this thesis, was the design and implementation of basic turning for the robot, as well as the continued development of the simulated GuRoo environment, including the design and implementation of a new behaviour control system that allows simultaneous tasks (i.e. crouching and torso movement, or walking and torso movement) that are independent of each other to run concurrently on the robot (both in simulation and real life). This is an important achievement since the future work on the GuRoo software will include moving to a more sophisticated behaviour management system, which will hopefully be built on top of this newly implemented control system.

Table of Contents

List of Figures

1 Introduction

The aim of this thesis is to continue the development of the GuRoo humanoid robot. The goal of this thesis is therefore to develop a usable gait generation algorithm that will allow GuRoo to walk stability in a straight line.
To prove the gait generation is valid, it will be developed and tested inside a simulated environment and then tested on the actual robot.
Also it is a secondary goal of this thesis to extend the gait generation algorithm to allow the robot to turn and move at the same time.

One of the outcomes of this thesis, besides developing the gait generation algorithm, was the development of the MasterSlave behaviour control system. This system was developed as part of this thesis for two reasons. Firstly, the current system was considered to be unsuitable for the gait generation algorithm being developed. Secondly with plans next year to move to a more sophisticated behaviour management system this new system would provide a good base for that.

1.1 GuRoo

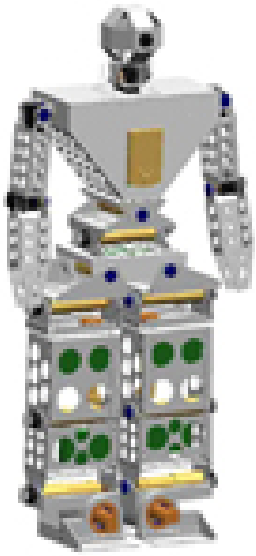

Figure 1.1: CAD Diagram of GuRoo

The goal of the GuRoo project is to eventually compete in the RoboCup humanoid soccer competition. The idea is that eventually a team of humanoid robots will be able to compete (and win) against a human team.

The mechanical GuRoo humanoid robot is the product of two years of work done by undergraduates at the University of Queensland. The robot has twenty-three degrees of freedom (actuated joints), which were deemed as the best to imitate the human form. GuRoo stands at 1.2 metres tall and weighs approximately 38 kilograms.

The GuRoo robot uses eight RC servo motors to control the upper joints (head, neck and arms) and fifteen 70W DC geared motors. The lower DC motors are controlled by five identical controller boards, with each board controlling three motors each. The motors use velocities rather than angles for movement and the control boards perform low-level control loops to maintain the desired velocities. The desired joint velocities for each joint are calculated by a software program, either running on a standard PC or an IPAQ handheld computer that allows the robot complete freedom. The information is passed via a serial cable to the robot and is then converted to a Controller Area Network (CAN) that then passes the information to the correct board.

Important to the success of the GuRoo project, is the simulation program developed concurrently with the GuRoo hardware. The simulator program uses the open-source Dynamechs library developed by Scott McMillan [McMillian, 1995]. This allows work to be done on the software for the robot without needing constant access to the actual hardware. Also it reduces the risk of damage to the hardware from untested software. The simulator is designed so that the majority of code written for GuRoo can be used both on the simulator and on actual hardware without requiring any modifications.

The simulator program simulates all the aspects of the robot to a close degree, including simulating the CAN bus etc.

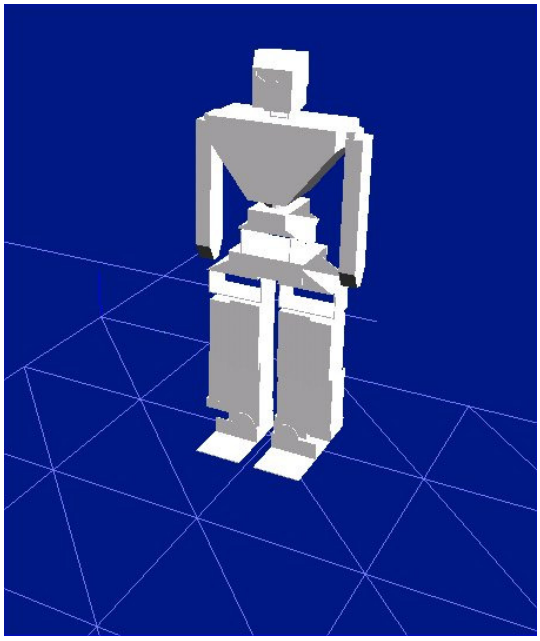

Figure 1.2: Simulated GuRoo

1.2 Reasoning behind need for Gait Generation

Since the goal of GuRoo is to eventually compete in the humanoid robot soccer competition at RoboCup, giving the robot the ability to work is an important step towards realising that goal.

Clearly, the walk must be easily configurable and dynamic in the sense that the walk can be altered in real-time to be able to give the robot the maximum amount of freedom to move.

2 Research/Background

In this chapter, we will outline some of the techniques that are used in this thesis.

2.1 Zero Moment Point

During any given movement of any section, the robot has two main forces acting on it. One of these forces is gravity and the other is inertia. The point where the moments caused by these two forces cancel is the zero moment point (ZMP).

This concept was first put forward by Vukobratovic in the nineteen seventies and is widely used today as a technique for assisting in robot stability.

2.2 Genetic Algorithms (GAlib)

The idea behind genetic algorithms is to mimic the basic ideas of natural evolution and use that to effectively solve complicated problems. They use basic principles of natural selection and evolution in order to produce many solutions to any given problem.

Genetic algorithms are very general algorithms and usually work well in any search space. This happens since in order to use a genetic algorithm to solve a problem, all you need to know is what you need the solution to be able to do well, i.e. you need to have a objective function that compares what the possible solutions can do to what the final 'perfect' solution should be able to do.

In a general genetic algorithm, we have any possible solution (an individual), the group of all individuals (a population), all the possible solutions to the problem (the search space). Note that the search space is the collection of all the possible solutions, while a population may only contain some of those solutions (as individuals).

The genetic algorithm generally works in the following fashion. A population is created with a group of individuals (which are randomly selected from the search space). All of these individuals are then evaluated using the objective function. This function, as stated above, determines a fitness or score for the individual. The fitness is the measure of how well the individuals does in respect to what the perfect solution should do.

Now, once all the individuals have been evaluated, usually two of them are selected based on their fitness to 'reproduce'. Basically this means that traits from each individual are selected to create a new individual.
Usually the worst scoring individuals are eliminated from the population (basic survival of the fittest).
Also during this time its possible for some of the individuals, usually one or two tenths of a percent of the total population, to be mutated. This means that a trait of the given individual is either changed by a small amount or replaced. This allows for variety in the population.

This process of reproduction continues then until a suitable solution has been found, or a certain number of generations has passed, dependant on the programmers needs.

Genetic algorithms are used as part of this thesis, however we utilize an open-source, free for non-commercial use genetic algorithm library (GAlib) written by Matthew Wall [Wall, 1995]. This library allows us to use genetic algorithms without writing an algorithm from scratch (which would be a exercise outside the scope of this thesis). All we are required to write is a small program to run the genetic algorithm and a fitness function that runs inside the robot simulator program.

3 Initial Work/Observations

In this chapter, we will firstly look at the old high-level control system. This system previously controlled the running of behaviours and has now been replaced by the MasterSlave behaviour control system that was developed as part of this thesis.

Secondly, we will look at a walk that was developed for the GuRoo robot. This walk was previously the most successful walk used on the robot. However the NewWalk gait generation algorithm, developed as part of this thesis, now replaces it.

3.1 The old high-level control system

The robot currently runs two different areas of control systems. One of these areas is focused on the motors, and controlling the motors to ensure that they work towards the maximum efficiency and accuracy. This is a very low-level type of control system. The other control system is one that sits above that. This system decides exactly what motor velocities to provide to the low-level control system in order to achieve the desired robot movement (i.e. walking, crouching, etc.). This high-level system doesn't concern itself with how the motors work, but simply on deciding what velocities to supply to them at any given moment in time.

Initially, when the robot was constructed, the major focus was on the hardware design. Initially programs like the simulator program and the software to run on the actual robot were written to support the design, building and testing of the actual robotic hardware. Thusly there was a limited amount of design work done on the actual high-level control systems.

As a result, the high-level system was designed to allow a single control function to decide what motor velocities to supply to the robot. These motor velocities were

supplied directly to the robot by using a simple array to store them and those velocities could be modified by any other function.

The resulting high-level control loop appeared similar to Figure 3.1. Although different functions were written to allow different actions on the robot (i.e. one function to do walking, one to do crouching etc.), only one of these functions could run at any given time.

At the time however, this was sufficient for the needs of the robot, given its limited capabilities.

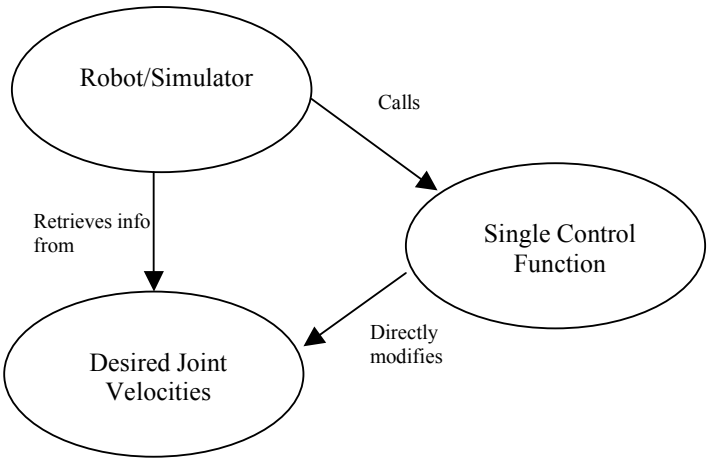

Figure 3.1: The Old High-level Control System Loop

3.2 The GAwalk

Developed at the University of NSW by Tak Fai Yik [Yik, 2002], the GAwalk algorithm uses a genetic algorithm in order to evolve a gait from a series of pre-generated key-frames. The idea behind GAwalk is that these key-frames (which are positions of the robot at given points in time) are predetermined to be zero moment point stable. A locus of motion is then evolved using a genetic algorithm that interpolates between the stable key-frames to generate a complete and stable gait.

Although this is a quite sophisticated method of generating a gait sequence, it does have a number of disadvantages.
Firstly, the stable key-frames are based on a number of known variables. Things such as the distance to step, the height to lift the legs when moving etc. all are predetermined.
Secondly, the entire sequence of generating a gait using this method is not real-time. The process is done offline and the generation can take anywhere from twenty-four to four-eight hours to complete.

Clearly, although this method of generating a gait is an impressive one and one that does provide a stable straight-line walking gait for GuRoo, it isn't completely inline with the goals of the project.
It isn't likely that GuRoo is always going to be travelling at the same speed all the time, therefore in order for the GAwalk to be able to change its speed, it must either be able to generate new gait sequences on the fly (something that just isn't possible given the computing hardware available) or to have all the different possible speeds already pre-generated (something that isn't impossible, but quite unfeasible).

4 Project Goals

4.1 Gait Generation Algorithm

The primary goal of this thesis is to develop a gait generation algorithm. This algorithm should be able to allow the GuRoo humanoid robot to walk stably in a straight line. This walking should be easily parameterised, with factors such as step size and the height to lift the legs during walking to be manipulated during run-time.

The secondary goal of this thesis is to extend the above algorithm to be able to, in a basic way, allow the robot to move omni-directionally, i.e. robot will move in a curved line. This movement should also be easily parameterised, with factors such as turn angle able to be manipulated during run-time.

The goal of walking in a straight line was achieved by the development of the NewWalk gait generation algorithm. This algorithm, which will be examined in a later chapter, achieves the primary goal of stable straight-line walking.

Efforts towards the secondary goal were achieved by the development of the NWTurn behaviour. This behaviour runs independently from NewWalk, although it does use a similar structure. It relies on the new MasterSlave behaviour control system (see below) to allow it to achieve its goal of extending the NewWalk gait to allow turning. How it achieves this will be explained later in this thesis.

4.2 MasterSlave Behaviour Control System

Although not an initial goal of this thesis, this aspect of this thesis was a necessary one. As mentioned earlier in this report, the MasterSlave system was developed because of the unsuitability of the current high-level control system for current and future work. Therefore the goals of this system were to not just replace the current system, but also to

extend it in a variety of ways (listed below) and to provide a suitable base for future work on robot behaviour management.

The major goals for the MasterSlave behaviour control system are:

- Allow more than one behaviour running at any given time
- Allow more than one instance of any behaviour to run
- Force all the behaviour code to be encapsulated inside a single class
- Provide a single well controlled conduit to allow manipulation of joints
    - Setting joint velocities
    - Keeping track of current joint positions
- Provide a cleaner mechanism for allowing behaviours to interact
- Provide a common base for developing behaviours
- Provide a suitable base for future behaviour management system
- Provide a suitable base for the NewWalk and NWTurn gait generation algorithms

5 The MasterSlave Behaviour Control System

Above we outlined just what we desired the MasterSlave behaviour control system to achieve. Now, to examine in detail how the MasterSlave system works, we will look at the individual components of the system. The system is made up of three different components, each encapsulated within their own classes. These classes are the *Joint* class, the *MasterControl* class and finally the *SlaveControl* class.

5.1 Design and Implementation

These components work together, as detailed below, to achieve our desired goals for the system.

5.1.1 The Joint Class

The idea behind this class is to encapsulate all the information needed for each particular joint inside a single class.

Each instance of this class stores the current desired velocity for the joint. This velocity is set through a helper function and is stored in radians. With the joint however the desired joint velocity cannot be added, subtracted etc. it can only be set. This keeps the class simple.

Once the MasterControl class is ready, it will call the update function (see below) that will convert the current velocity (stored in radians) to encoder counts that the low level control loops will use.

This class also updates the theoretical current position of the joint (stored in radians). This is done (as shown below) by accumulating the desired joint velocities during every update. However, these positions are not actual positions of the motors, they are,

assuming that the robot does exactly what it is told to do and behaves as theoretically expected, only theoretical values (but usually close to actual values).
The joint class also contains functions to reset this theoretical joint position.

```
////////////////////////////////////
// Updates the joint
////////////////////////////////////
int Joint::update ()
{
  // Convert to the correct format
  int temp_des_vel = 0;
  temp_des_vel = SRAD2ENC (desired_joint_vel);

  // Update the position of the joint
  current_joint_pos += desired_joint_vel;

  // Clear the desired joint velocity
  desired_joint_vel = 0.0f;

  // Return the stored desired joint velocity
  return temp_des_vel;
}
```

Figure 5.1: The update function inside the Joint class

5.1.2 The MasterControl Class

This class is designed to be the brains of the MasterSlave behaviour control system. It's designed to provide a conduit that all high level operations on the robot are conducted.

Apart from storing a list of all the robots joints, it also stores a list of all the slaves being used and the current time for the robot. This means that for any operation that is done on the robot, it must be done through an instance of this class.

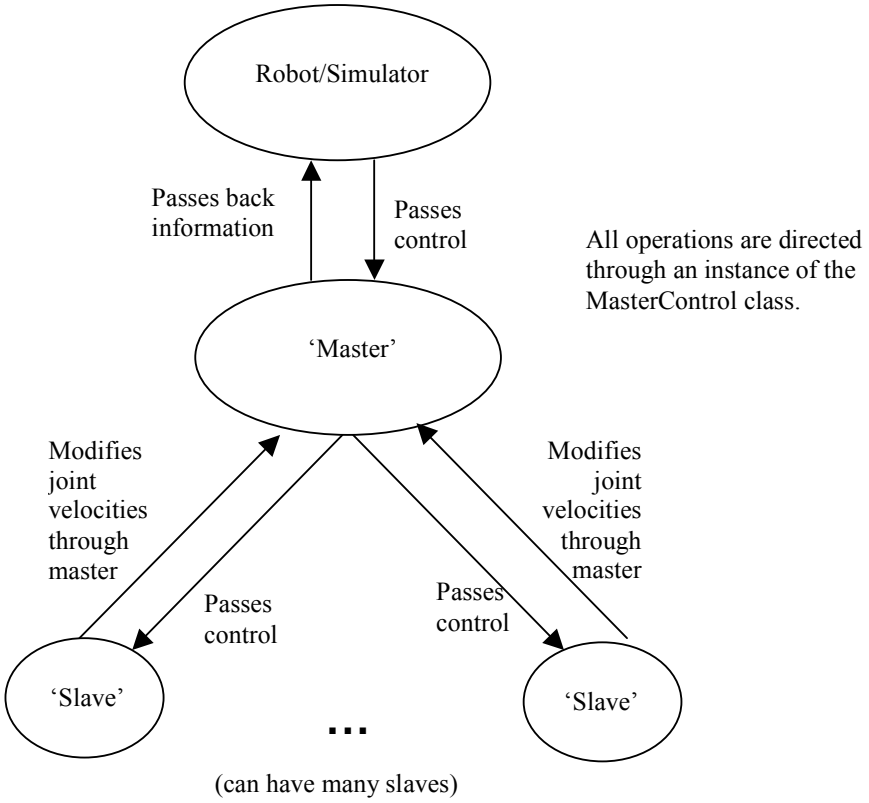

Figure 5.2: The MasterControl class acting as a conduit for Slave operations

Such a design is achieved using a number of simple techniques.
Although the question may be raised as to why C++ is used to implement the design, especially when most of the GuRoo codebase is written in the C language. One of the reasons this was done was that it actively encourages encapsulation and limits the desire to place code anywhere in the code base. Using C++ and classes this allows each behaviour to be isolated inside separate files and classes and to keep all the important functions and variables that were previously global inside the MasterControl task with functions to control access to them.

The second reason goes to the heart of the MasterSlave system and to the core design of the MasterControl class. Since each behaviour is completely isolated from all other

behaviours (except where explicitly written to interact) and all the MasterControl class has is a list of the slaves, it has no idea how to access them. We overcome this problem by making use of inheritance and virtual functions, techniques that are easily done using C++.

How we implement the SlaveControl class will be discussed after this section, but basically the idea is that that every slave is derived from the basic SlaveControl class. Inside SlaveControl is a function that is declared a virtual function. In a basic sense, having a function declared as a virtual function ensures that it will be able to be called by MasterControl for any derived instance of SlaveControl and always run the function written for that derived instance.
In terms of MasterControl, this ensures that it will always have a common way of running the slaves.

In order to run the slaves, we use two functions, registerController and unregisterController. This allows us to add a slave to the list of slaves to be run and remove a slave from the list of slaves that are running respectively. Currently this list of slaves is implemented using a simple array to store pointers to the slaves. Also, at the moment no particular importance is given to the running of slaves with them being run in the order that they are registered.

```
//////////////////////////////////////
// Register a slave controller with the master control
// Any slave registered will have SlaveControl::controlFunction called
// every MasterControl::update.
//////////////////////////////////////
bool MasterControl::registerController (SlaveControl *control)
{
  // TODO: Implement a more sophisticated system perhaps
  if (!control)
  {
    printf ("ERROR: Master Control::registerController, invalid slave to register.\n");
    return false;
  }

  if (numSlaves > MAX_SLAVES)
  {
    printf ("ERROR: Master Control::registerController, too many slaves registered.\n");
    return false;
  }

  // Check that the slave hasn't already been registered
  for (int i = 0; i < numSlaves; i++)
  {
    if (slaves[i] == control)
      return true;
  }

  // Add to slave list
  slaves[numSlaves] = control;
  // Increase count of slaves registered
  numSlaves++;

  return true;
}
```

Figure 5.3: The function to register slaves inside MasterControl.

```
//////////////////////////////////////
// Unregister a slave control
//////////////////////////////////////
bool MasterControl::unregisterController (SlaveControl *control)
{
 if (!control)
 {
  printf ("ERROR: Master Control::unregisterController, invalid slave.\n");
  return false;
 }

 int slavePos = -1;

 // Search the list for the slave specified
 for (int i = 0; i < numSlaves; i++)
 {
  if (slaves[i] == control)
  {
   slavePos = i;
   i = numSlaves;
  }
 }

 // Found slave, now remove pointer and move all slave pointers back by one
 if (slavePos != -1)
 {
  // Clear the slave pointer (but do not delete the instance)
  slaves[slavePos] = NULL;

  // Move all the pointers back by one position
  for (int i = slavePos; i < numSlaves; i++)
  {
   slaves[i] = slaves[i+1];
  }

  // Clear last pointer (otherwise there would be two pointers to the same slave)
  slaves[numSlaves] = NULL;

  // Reduce number of slaves by 1
  numSlaves--;
 }
```

```
  else
  {
    // Error, cannot find the given slave in the list, print a error message
    printf ("ERROR: Master Control::unregisterController, unable to find slave
'%s\n", control->getName ());
  }
  return true;
}
```

Figure 5.4: The function to unregister slaves inside MasterControl

Now we have mechanisms for keeping track of the slaves in the system. We can register a slave and unregister a slave from a list of slaves stored in MasterControl.
However we also need a mechanism for running the slaves.

Inside both the main control loop of both the simulator program and the gait program that controls the actual robot, a call is made to the update function of the MasterControl class. This function (shown below), for every slave that is registered, calls the slaves 'controlFunction' function. Since this is a virtual function, we do not have to worry about what type of slave it is (as long as it inherited from SlaveControl originally). Now the control is passed to the Slave which then completes its operations and passes control back, allowing the next slave to be run or, should all the slaves been run it will extract the desired joint velocities from the various instances of the Joint class (which represent the various joints of the robot) and make them available to the low level control loops outside of the MasterSlave system.

```
//////////////////////////////////////
// Updates the master controller
// Updates the time and all the slaves etc.
//////////////////////////////////////
void MasterControl::update ()
{
 // Update the time
 time += float(CENTRAL_SPEED);

 // Call all the slaves control function
 for (int i = 0; i < numSlaves; i++)
 {
  // Null pointer check
  if (slaves[i] != NULL)
  {
   // If a slave is not finished, then call controlFunction
   // otherwise if finished, then automatically unregister
   if (!slaves[i]->isFinished ())
    slaves[i]->controlFunction ();
   else
    unregisterController (slaves[i]);
  }
 }

 // Extract the desired velocities of the joints and update the joints
 for (int a = 0; a < TOTAL_MOTORS; a++)
 {
  msDesiredJointVel[a] = getJoint (a)->update ();
 }
}
```

Figure 5.5: The 'update' function inside MasterControl

Finally there is one more major function that we have inside the MasterControl class. This function is what allows the Slaves to manipulate the desired velocities of each joint.

However, the control function of the slaves are not allowed to explicitly just set the desired velocities of the joint, instead the MasterControl class provides a number of ways to 'modify' the current desired joint velocity. This includes adding the slaves desired velocity to the current desired velocity for that joint, subtracting the slaves desired

velocity to the current desired velocity for that joint and finally to simply overwrite the current desired joint velocity with one specified by the slave. The function to do this is called modifyJointVel and allows the slave to specify the joint to apply the desired velocity to, the velocity to be applied and how to apply it (additive, subtractive etc.).

```
////////////////////////////////////
// Modify a desired joint velocity by a given amount
// a given way (ie. Add, subtract, overwrite etc.)
////////////////////////////////////
void MasterControl::modifyJointVel (int joint, float velocity, int action)
{
  float currDesJointVel = 0;
  Joint *theJoint = NULL;

  // Get the joint in question
  theJoint = getJoint (joint);

  // Get the current desired joint velocity
  currDesJointVel = theJoint->getDesiredJointVel ();

  // Apply the velocity using the given action
  switch (action)
  {
    // Additive
    case JOINT_MODIFY_ADD:
    {
      currDesJointVel += velocity;
    } break;

    // Subtractive
    case JOINT_MODIFY_SUB:
    {
      currDesJointVel -= velocity;
    } break;

    // Overwriting
    case JOINT_MODIFY_OVERWRITE:
    {
      currDesJointVel = velocity;
    } break;

    // Default behaviour (ie. invalid action parameter)
    default:
    {
      // ERROR?
      printf ("ERROR: MasterControl::modifyJointVel, unknown action specified,
'%d'.\n", action);
    } break;
  }
```

```
  // Set the new desired joint velocity in the joint
  theJoint->setDesiredJointVel (currDesJointVel);

  // Clear pointer
  theJoint = NULL;
}
```

Figure 5.6: The modifyJointVel function

Therefore, by using these mechanisms, we create a central conduit that all behaviours are conducted through.

5.1.3 The SlaveControl Class

The idea here is that the class SlaveControl is a base class for all other slaves. This class contains all the necessary basic information for slaves to be able to be run by the MasterControl class' update function.
When you create a slave it is always derived from SlaveControl. This is how inheritance is used to ensure that all slaves will be able to easily interact with the MasterControl class.
Each slave also has a pointer to an instance of the MasterControl class. This is passed as a parameter to the constructor of the SlaveControl.

```
GAwalk *gawalk = new GAwalk ("GAwalk", master);
master->registerController (gawalk);
```

Figure 5.7: An instance of a slave being created (with a name and a pointer to an instance of the MasterControl class) and then being registered with an instance of the MasterControl class

Also, the SlaveControl class has a single function, called ControlFunction that provides a common interface for the MasterControl class to run the slave. This function will be called for each registered slave once during every cycle of the main control loop.

Inside this function, it is expected that all the code to calculate the desired velocities for each required joint is done. Those joint velocities are set by using the modifyJointVel function inside the MasterControl class.

An example of a simple slave is shown below. This slave generates a velocity profile from the given period and start time. It then requests to modify the desired joint velocity of the joint LEFT_KNEE by overwriting whatever value is currently set with the already generated velocity.

```
//////////////////////////////////////
// Control function, called by MasterControl
// All the gait generation work is done here
//////////////////////////////////////
void SimpleSlave::controlFunction ()
{
  // Example of changing joint velocities

  // Get the velocity profile, given a period of 1 and start time of 1
  float velocity = master->Velocity2 (1.0f, 1.0f);

  // Set the left knee joint velocity by the given velocity
  // We want to overwrite any other given value, so we specify
  //  JOINT_MODIFY_OVERWRITE
  // we could also specify JOINT_MODIFY_ADD and JOINT_MODIFY_SUB
  // to add velocities and subtract velocities from the current desired joint velocity
  // of the joint
  master->modifyJointVel (LEFT_KNEE, velocity,  JOINT_MODIFY_OVERWRITE);
}
```

Figure 5.8: controlFunction of a simple Slave

5.2 Results

For these results, we will show graphs displaying the desired and actual joint velocities. For these tests, we will use a slave called RobotBasics that allows us to manipulate a single joint with us specifying the angle to move the joint and the period over which to move it.
The full code for RobotBasics is too long to show here, but for now assume that the slave will take the supplied parameters and operate as described above.

For these we will use the leg twist joint of the left leg and the right ankle pitch joint of the right leg. Each will use a period of three, since this is a good number for both the simulator and the actual robot. The angle we will use will be twenty-two and a half degrees for the twist joint and eleven and a quarter degrees for the ankle pitch joint. This allows us to see the movement of the robot without risking damage to the actual robot (when running on the actual robot, for the simulator we disable gravity to see the effects without interference).

5.2.1 Running multiple slaves (Simulator)

All these results are from the simulator program.
We will show the results of running the leg twist movement using the RobotBasics slave, firstly running by itself and operating on the left leg.
Then we will show the graph of the same slave running twice on the same leg (i.e. there will be two slaves desiring to move the left leg twist twenty-two and a half degrees each).
Then finally we will run two RobotBasics slaves, one operating on the left leg twist and the other operating on the right ankle pitch, both registered at the same time.

```
// Add a new RobotBasics class
RobotBasics *rb = new RobotBasics ("testRB", master);
master->registerController (rb);

// Start a test action
rb->startTest (3.0f, LEFT_LEG_TWIST, 22.5f);
```

Figure 5.9: Code to run one RobotBasics slave operating on the left leg twist

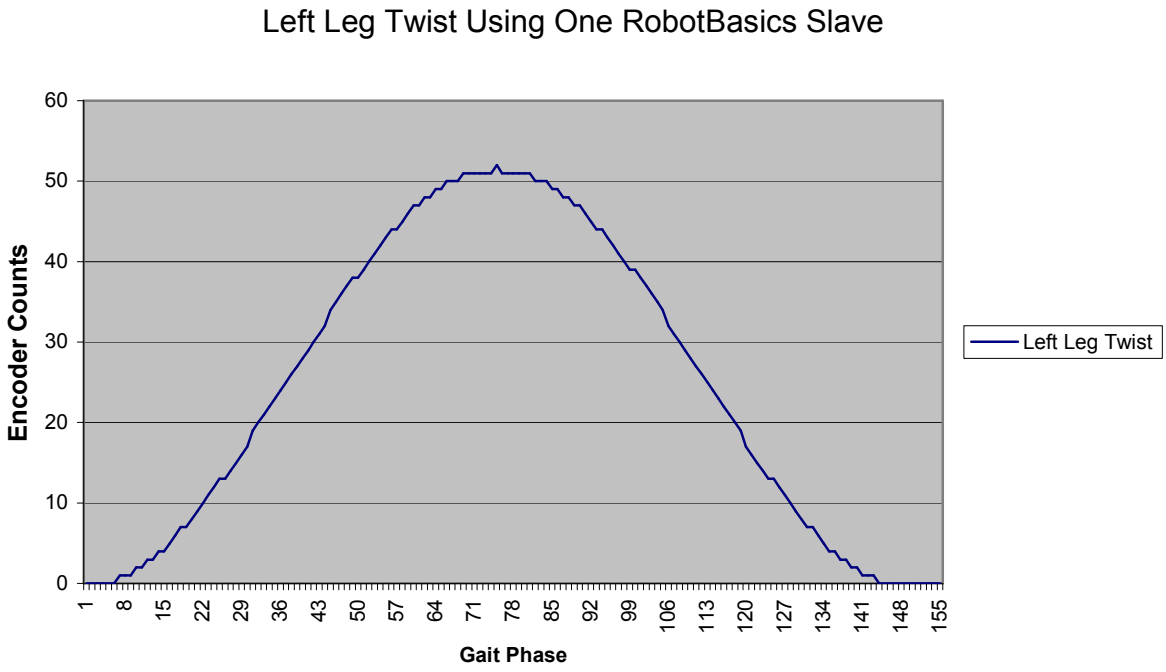

Figure 5.10: Graph of desired joint velocity (encoder counts) of Left Leg Twist joint using one RobotBasics slave (Simulator)

As you can see, the maximum number of encoder counts reached is fifty-two. Now we will show the maximum number of encoder counts using two different instances of a RobotBasics slave operating on the same joint in the same manner.

```
// Add a new RobotBasics class
RobotBasics *rb = new RobotBasics ("testRB", master);
master->registerController (rb);

// Start a test action
rb->startTest (3.0f, LEFT_LEG_TWIST, 22.5f);

// Add another new RobotBasics class
RobotBasics *rb2 = new RobotBasics ("testRB2", master);
master->registerController (rb2);

// Start a test action
rb2->startTest (3.0f, LEFT_LEG_TWIST, 22.5f);
```

Figure 5.11: Code to run two RobotBasics slave operating on the left leg twist in the same manner.

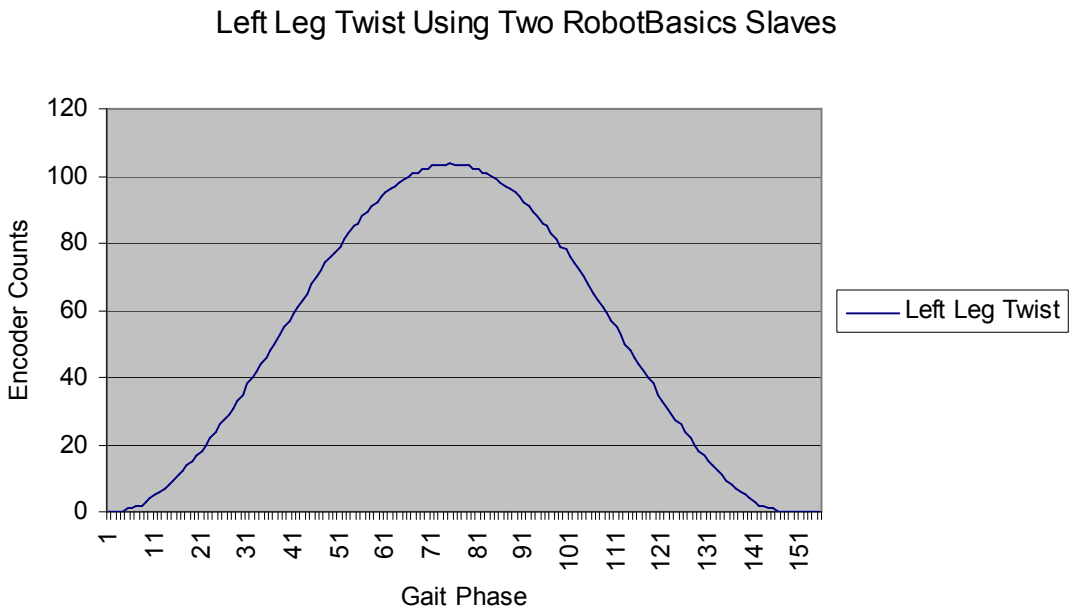

Figure 5.12: Graph of desired joint velocity (encoder counts) of Left Leg Twist joint using one RobotBasics slave (Simulator)

As you can see, the overall desired joint velocity, in encoder counts, is now twice as large as before.

This proves two things. Firstly it shows that two slaves operating on the same joint can do so without knowledge of each other without any effect at all (i.e. effects such as overwriting joint velocities, changing joint velocities to unusual values etc.).

Secondly it shows that two instances of the same slave can run along side each other without any problems at all.

Finally we will demonstrate two RobotBasics slaves running at the same time operating on two different joints (although in a similar manner).

```
// Add a new RobotBasics class
RobotBasics *rb = new RobotBasics ("testRB", master);
master->registerController (rb);
// Start a test action
rb->startTest (3.0f, LEFT_LEG_TWIST, 22.5f);

// Add another new RobotBasics class
RobotBasics *rb2 = new RobotBasics ("testRB2", master);
master->registerController (rb2);

// Start a test action
rb2->startTest (3.0f, RIGHT_ANKLE_FWD, 11.25f);
```

Figure 5.13: Code to run two RobotBasics slave operating on the left leg twist in a similar manner

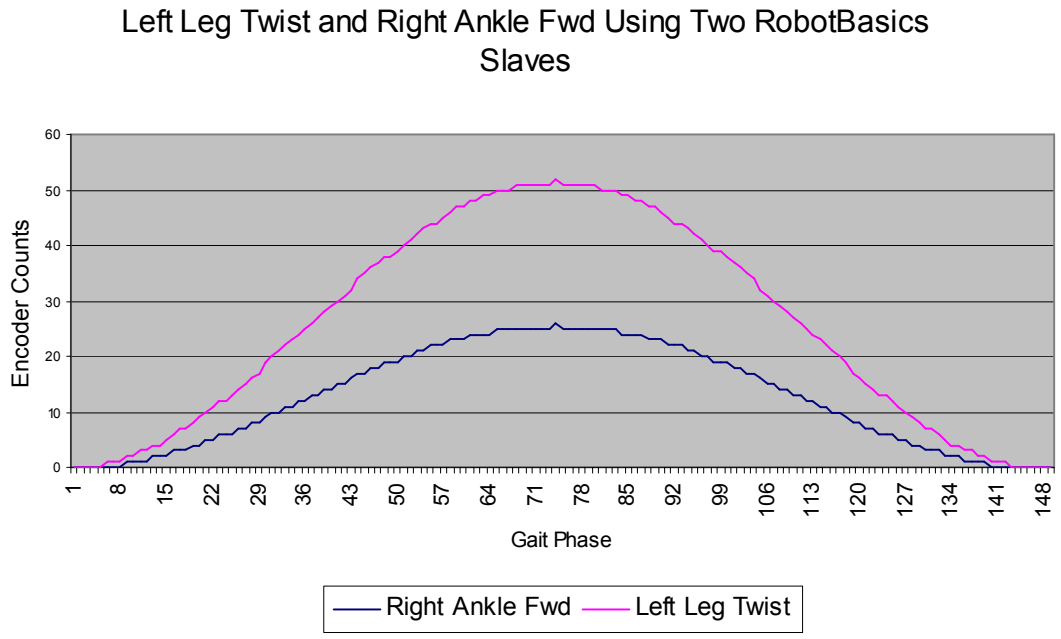

Figure 5.14: Graph of desired joint velocity (encoder counts) of Left Leg Twist and Right Ankle Fwd joints using two RobotBasics slaves (Simulator)

As you can see by the above results, we have run the two instances of the same slave operating on two different joints in a similar manner at the same time.

5.2.2 Running on real robot

In order to ensure that the MasterSlave system works the in the same manner when utilised in both the simulator program and the program that drives the actual robot (since the same MasterSlave code is used for both programs), we will run the exact same tests as above.

Firstly we will test running a single RobotBasics slave operating on the left leg twist joint of an angle of twenty-two and a half degrees.
Secondly we will test running two RobotBasics slaves both operating on the left leg twist joint of an angle of twenty-two and a half degrees each (meaning the joint should move a total of forty-five degrees).

Finally we will test running two RobotBasics slaves each operating on the left leg twist and right ankle forward joints. Each slave will move their joint twenty-two and a half degrees and eleven and a quarter degrees respectively.

Also we will graph both the desired joint velocity and the actual joint velocity in order to show that the system is actually driving the robot, not just generating velocities. It should be noted that in the graphs, the notable difference in desired versus actual is due in part to delay in the logging of the data and has nothing to do with the MasterSlave system.

Note that the code used for these tests is the same as code used above.

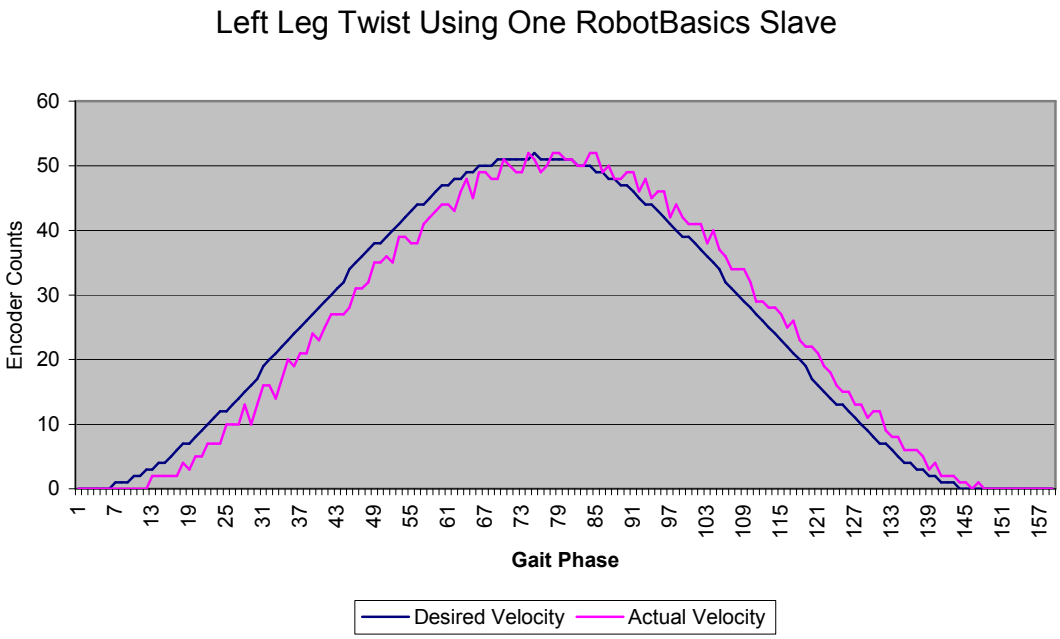

Figure 5.15: Graph of desired and actual joint velocities (encoder counts) of Left Leg Twist joint using one RobotBasics slave (Actual Robot)

As you can see the desired joint velocity profile that is generated is the same as the result shown above for the simulator program.

Now we run the second test of having two slaves operating on the same joint in the same manner (at the same time).

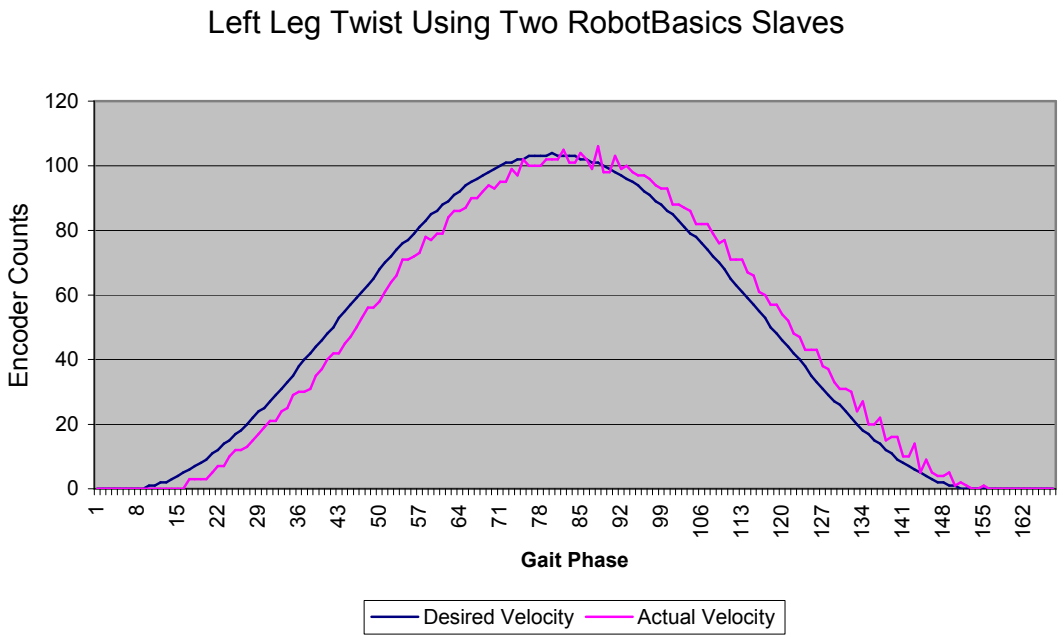

Figure 5.16: Graph of desired joint velocity (encoder counts) of Left Leg Twist joint using one RobotBasics slave (Actual Robot)

Now we will run the test of having two instances of the RobotBasics class running at the same time. One instance will be operating on the left leg twist joint and the other will be operating on the right ankle forward of the robot. The left leg twist will be moved twenty-two and a half degrees and the right ankle forward will be moved eleven and a quarter degrees.

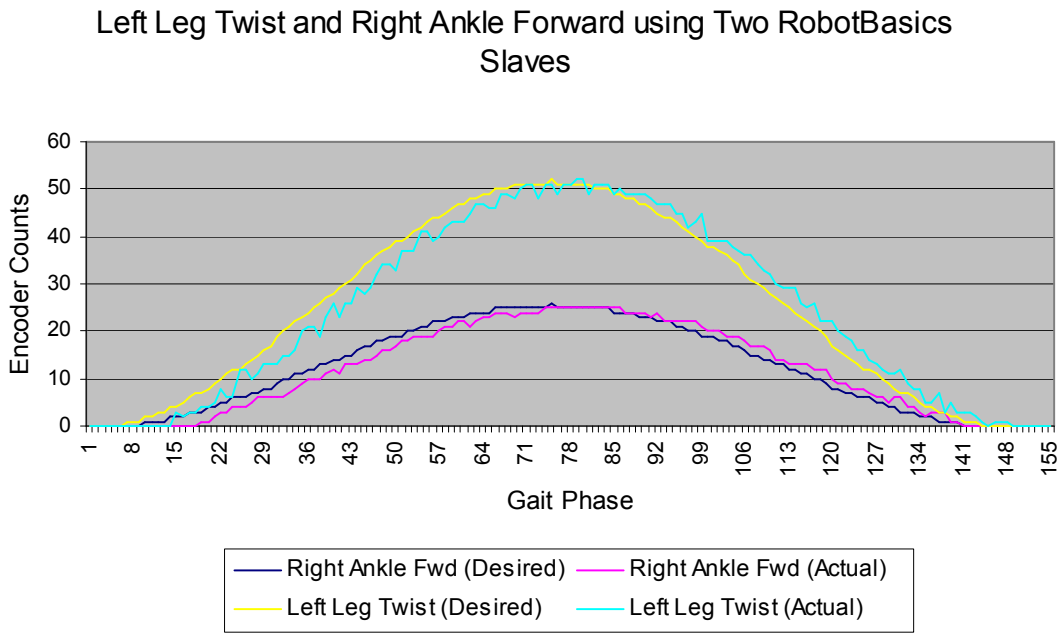

Figure 5.17: Graph of desired joint velocity (encoder counts) of Left Leg Twist and Right Ankle Fwd joints using two RobotBasics slaves (Simulator)

Now as you can see from the above results, we get exactly the same results as from the simulator.

From these results we can determine a number of important points.

- The system can run multiple slaves at the same time and those slaves can either be of the same class or different classes (assuming all slaves are derived from SlaveControl initially). The MasterControl does not concern itself with the types of slaves being run.

- All slaves can operate on joints independent of all other slaves.

- Finally we see that the MasterSlave system works for both the simulator program and for the program driving the actual robot. Given that the MasterSlave system was written to be independent of both programs, this is an important result to be shown.

6 The NewWalk Gait Generation Algorithm

Previously we have outlined what our desired goals of NewWalk were, i.e. building a gait generation algorithm that can allow the GuRoo robot to walk stably in a straight line. Now we will look at the design and implementation of the NewWalk gait generation algorithm.

6.1 The Simplicity of NewWalk

The driving factor behind the design of NewWalk was the idea of simplicity. Previous work towards gait generation involved complex software and algorithms that, although they provided useful results, were difficult to continue the work on.
As an example of this, the GAwalk (outlined in a previous chapter of this document) that uses evolutionary computing methods and some rather complicated mathematics. Although this does provide good results, it leaves a complicated and difficult (code)base to work with and coupled with a large lack of documentation within that code, makes future work with GAwalk extremely difficult.

Therefore, the idea was to develop an algorithm that was not just effective, but also simplistic.

6.2 Design of NewWalk

Previously, most of the walking algorithms that were developed, including the GAwalk and a walk developed by Andrew Smith [Smith, 2001], were methods that were offline methods. That is, the walk was generated by another program and that resulting walk was used to run the actual robot. The generation however was not done in real-time, the walk was unable to be changed while the robot was running. This meant that such conditions like on command stopping of the walk were unable to be implemented.

Although, again, though the results from such walks were encouraging, the fact that the walk was not configurable in real-time was a major issue. Since when have you ever seen a person always walk at the same speed and always travel the same distance every single time they moved?
Therefore, after looking at past work, it was determined that a purely off-line computational approach to gait generation would not be able to satisfy the needs of the GuRoo project. However, it was also clear that a purely manually calculated approach (all the kinematics done by a human) would probably not be feasible either (nor have a good chance of succeeding).
Thusly, the NewWalk algorithm was designed to be a middle ground between these two different paradigms.

The design of NewWalk is thusly a three-stage approach.

Firstly, we take a basic walk cycle, one that is able to move the robot in a forward manner with some basic stability. This is the base for the entire gait generation algorithm. We now take this basic walk and break it down into its most basic components. These components are designed to be independent of each other despite the fact that some of these components may be running simultaneously.

Now we have a number of components, we use a number of different variables to tie them together. These variables are in essence, magic numbers, numbers that take into account the user defined aspects of a walk (step size, leg lift height etc.) and generate a magic number in which to bind all the different components together.
These numbers ensure that all the components work co-operatively together to not just drive the robot forward, but ensure repeatability in the gait sequence.
Later we can use a off-line computation calculation technique, in this case genetic algorithms, to calculate the exact numbers needed to ensure the best walk possible.

Now we have a number of components and variables to bind these components together, we have to calculate at which point in the gait cycle these components will run.

The basic idea is that each component generates a velocity for a number of different joints. Once all the components have completed running we then start the sequence of operations again and continually running this cycle until a stop command is received or the behaviour stops running.

6.3 Components of NewWalk

As stated above, the sequence of operations that a gait cycle consists of is broken down into a number of various components. These components are designed to mimic certain aspects of a humanoid walk, although adapted to the humanoid GuRoo robot.

We will now examine these components in more detail. In the implementation of NewWalk, we call the components 'Stages'.

These 'stages' are ordered below in the basic order that they run in, however it should be noted again that many of these stages do run simultaneously, so the order listed below isn't a fixed one.

6.3.1 Lean to Side

This stage is a very simple stage. It is designed as part of the sequence of operations to start a gait sequence. All it is designed to do is lean the robot to one side, moving the weight of the robot from above both legs to over the top of only one leg. It does this by manipulating the ankle and hip roll joints

6.3.2 Raise Leg

This stage is also a very simple stage. As the second part of the sequence of operations to start a gait sequence, it runs after the 'Lean to Side' stage. All this stage does is lift a single leg off the ground by manipulating the knee, hip pitch and ankle pitch joints.

Since the previous stage moved the centre of gravity away from the leg being manipulated here, the robot is able to lift the leg without affecting its stability.

Once this stage is completed, the robot is now in a stage where is can start the walk cycle. Once a walk cycle is completed (i.e. all the necessary components or stages have been run) the robot will always finish in this state, ready to begin another cycle.

6.3.3 Extend Leg Forward

Now that the robot is in the walk cycle ready state (i.e. one leg is raised and the weight of the robot has been moved to the opposite leg) we can begin the walk cycle. This stage is therefore the first stage of a walk cycle.
This stage is designed to extend the leg forward by straightening the knee joint and manipulating the ankle and hip pitch joints to prepare the leg to be placed back on the ground, except that the leg will now be placed in front of the robot (the distance to move the leg forward is determined by a user-configurable variable which can be changed in real-time while the robot is operating).

6.3.4 Lower Other Leg

In operational terms, this stage is the same as the 'Raise Leg' stage, with the only difference being that it moves the leg a difference amount and also operates on the opposite leg. This stage operates on the leg that is currently holding the weight of the robot on it.
The idea behind this stage is to use this to assist in placing the foot being extended forward (in the 'Extend Leg Forward' stage).

6.3.5 Sway to other side

This stage is similar to the 'Lean to Side' stage. It manipulates the ankle and hip roll joints of both legs to move the weight of the robot. However, this stage, instead of just

moving the weight from both legs to one leg, moves the weight of the robot from one leg to the other. The robot therefore 'Sways' from one side to the other.

Because this has the effect of moving the weight of the robot from one side to the other, or in this case from the leg placed below the robot to the leg placed forward of the robot, this moves the weight of the robot not just to the other side, but also moves the weight of the robot forward as well.

6.3.6 Straighten Body

This component of the walk is broken into two stages. The two stages are different in that one operates while the weight is on the left leg and the other operates while the weight is on the right leg.
Either way, this stage is designed to manipulate the ankle and hip pitch joints in order to assist the movement of the weight of the robot forward.
This stage and the 'Sway to other side' stage are usually run simultaneously in order to smoothly move the robots weight forward.

6.3.7 Move other leg forward

This stage is the final stage in the first half of the walk cycle. This stage is designed to move the robot in a stage similar to the walk cycle start state (the same state that the 'Raise Leg' stage leaves the robot in) with the distinct exception that the leg raised is now the opposite one. It manipulates the ankle pitch, hip pitch and knee joints in order to place the robot in the desired state.

6.3.8 (Swap legs and repeat)

These stages are the same as the 'Extend Leg Forward', 'Lower Other Leg', 'Sway to Other Side', 'Straighten Body' and 'Move Other Leg Forward' stages except that opposite joints are manipulated (left and right are reversed).

Since the state that the 'Move Other Leg Forward' stage leaves the robot in is the exact state the 'Extend Leg Forward' stage needs to start (except that the sides are reversed) it means that we can repeat the same sequence and end up with the robot, at the end of the walk cycle, in the walk cycle start state, ready to begin another walk cycle.

6.3.9 Drop Leg

These stages are part of the walk exit sequence that is used to stop the robot. These stage and the 'Lean to Centre' stage are the opposite of the two stages used to start the robot. This stage manipulates the knee, ankle pitch and hip pitch joints in order to place the robot's foot on the ground below the robot.

6.3.10 Lean to Centre

This stage is the final stage of the walk exit sequence. This stage moves the weight of the robot back to two legs by manipulating the ankle and hip roll joints. Once this stage is complete, the robot will theoretically have the same joint positions that it started with (assuming that no other slaves are running that might manipulate the joints themselves).

At this point once this stage finishes, the NewWalk behaviour will stop operating.

6.3.11 Walk Cycle and Turn Signal Stages

Finally we have number of stages that we used as helper stages. These stages are designed to run along side the other stages and provide an alert when to run certain other operations.
These stages do no actually do anything, but simply run inside the existing stage control framework.

Firstly we have a stage that signals the beginning of a new walk cycle.

In order to create a stable walk, the robot must always be in continuous forward motion. However, since the motor velocities that are generated are in the form of sine curves, if we start the walk cycle at the same time the last one stopped, the robot would not move smoothly forward (and that would create instabilities).
We get around this by signalling the beginning of a new walk cycle by using a new walk cycle stage. This stage runs for the entire walk cycle and finishes before the cycle of complete. When it finishes, it signals the next walk cycle to begin.
So at any point in time, it is possible to have two walk cycles running simultaneously. Also, at the same time that this stage signals to start a new walk cycle, it also updates the configuration of the particular walk in progress (i.e. updates and sets a new step size, or finds that the walk should be stopped etc.).

Secondly we have the two stages which, when completed running, signal to another behaviour (if the behaviour is currently running) informing it that the walk is in a certain state. This other behaviour is NWTurn and provides a mechanism for allowing the robot to walk and turn at the same time (NWTurn will be described in detail in a later chapter).

6.4 Implementation of Stages

In order to implement the algorithm we have designed, we need to implement a framework in which to work inside.

This framework will consist of, as stated above, developing the various components as 'stages'. These stages have three variables associated with them. These variables are used to maintain control of the stages and are called stageAct, stageStart and stageTime. The stage variables are stored in a large array. Each stage has a number associated with it.

stageAct is used to set a task to be either running or not.
stageStart is used to set when a task is supposed to start running. It is also used to generate a velocity profile to be used for that stage.
stageTime is used to determine how long a stage will be running for. It is also used to generate a velocity profile to be used for that stage.

```
//////////////////////
/// Activation of a stage
//////////////////////
bool stageAct[NUM_STAGES];

//////////////////////
/// Start time of a stage
//////////////////////
float stageStart[NUM_STAGES];

//////////////////////
/// Running time of a stage
//////////////////////
float stageTime[NUM_STAGES];
```

Figure 6.1: Definition of the Stage control variable arrays. From inside the NewWalk class definition

Firstly we will look at how the timings for the stages are setup, then after that we will look at how the algorithm keeps the various stages running by starting and stopping the stages. Finally we will look at the actual implementation of the various stages.

We use a function to set the timings for the entire gait sequence. It uses the current time as a base and calculates the stageStart and stageTime for each stage. The function takes one parameter. This parameter determines which set of stages will have the timings setup for them.

6.4.1 Timings

The timings are broken down into four sections, TIMING_FIRST, TIMING_CYCLE, TIMING_CYCLE_TWO and TIMING_STOP.
TIMING_FIRST is used to setup the 'Lean to Side' and 'Raise Leg' stages. These stages are the start of a walk sequence.
TIMING_CYCLE is the walk cycle.
TIMING_CYCLE_TWO is the same as TIMING_CYCLE. The difference is that the stages have different number that can identify them, but they do use the same code. This is required since the same stage with two different velocity profiles may be running at the same time.
TIMING_STOP is used to setup the 'Drop Leg' and 'Lean to Centre' stages. These stages stop a walk in progress.

```
//////////////////////////////////////
// Setup the timings of the gait
//////////////////////////////////////
void NewWalk::setupTiming (int timing)
{
  float timeLine = 0.90f;
  float baseTime = master->getTime ();

  // Set the timings for the gait
  if (timing == TIMING_FIRST)
  {
    // Setup the timings for the walk sequence
    for (int i = 0; i < NUM_STAGES; i++)
```

```
  {
   stageAct[i] = false;
   stageStart[i] = -1.0f;
   stageTime[i] = 0.0f;
  }
  // Setup for first cycle
  // The first cycle is unique

  // order of stages
  // 0 & 9,5,6,7,19,10,11,12,18,15,6 & 14
  stageStart[0] = baseTime;
  stageAct[0] = true;
  stageTime[0] = 1.0f;

  stageStart[5] = baseTime + (1.0f);
  stageAct[5] = false;
  stageTime[5] = 1.0f;
}
else if (timing == TIMING_CYCLE)
{
  // Setup the timings for the walk sequence
  for (int i = 0; i < 20; i++)
  {
   stageAct[i] = false;
   stageStart[i] = -1.0f;
   stageTime[i] = 0.0f;
  }

  stageAct[20] = false;
  stageStart[20] = -1.0f;
  stageTime[20] = 0.0f;

  stageAct[22] = false;
  stageStart[22] = -1.0f;
  stageTime[22] = 0.0f;

  stageAct[23] = false;
  stageStart[23] = -1.0f;
  stageTime[23] = 0.0f;

  // This stage runs for the entire cycle
  stageStart[9] = baseTime + (0.0f * timeLine);
  stageAct[9] = false;
  stageTime[9] = timeLine * 1.2500f;

  stageStart[14] = baseTime + (1.75f * timeLine);
  stageAct[14] = false;
```

```
stageTime[14] = timeLine * 1.2500f; //3.75

// Now we start to blend the stages
// Extend leg forward
stageStart[6] = baseTime + (0.0f * timeLine);
stageAct[6] = false;
stageTime[6] = timeLine * 1.0f;

// Lower other leg
stageStart[7] = baseTime + (0.0f * timeLine);
stageAct[7] = false;
stageTime[7] = timeLine * 1.25f;

// Sway to side
stageStart[19] = baseTime + (0.0f * timeLine);
stageAct[19] = false;
stageTime[19] = timeLine * 2.0f;

// Move other leg forward
stageStart[10] = baseTime + (1.0f * timeLine);
stageAct[10] = false;
stageTime[10] = timeLine * 1.0f;

// Switched to other leg now

// Extend leg forward
stageStart[11] = baseTime + (1.75f * timeLine);
stageAct[11] = false;
stageTime[11] = timeLine * 1.0f;

// Lower other leg
stageStart[12] = baseTime + (1.75f * timeLine);
stageAct[12] = false;
stageTime[12] = timeLine * 1.25f;

// Sway to side
stageStart[18] = baseTime + (1.75f * timeLine);
stageAct[18] = false;
stageTime[18] = timeLine * 2.0f;

// Move other leg forward
stageStart[15] = baseTime + (2.75f * timeLine);
stageAct[15] = false;
stageTime[15] = timeLine * 1.0f;

// New Walk Cycle Start stage trigger
stageStart[20] = baseTime + (0.0f * timeLine);
```

```
  stageAct[20] = false;
  stageTime[20] = timeLine * 3.5f;

  // Turning Start stage trigger 1
  stageStart[22] = baseTime + (0.0f * timeLine);
  stageAct[22] = false;
  stageTime[22] = timeLine * 3.25f;

  // Turning Start stage trigger 2
  stageStart[23] = baseTime + (0.0f * timeLine);
  stageAct[23] = false;
  stageTime[23] = timeLine * 1.5f;

}
else if (timing == TIMING_CYCLE_TWO)
{
  // Setup the timings for the walk sequence
  for (int i = 30; i < NUM_STAGES; i++)
  {
    stageAct[i] = false;
    stageStart[i] = -1.0f;
    stageTime[i] = 0.0f;
  }
  stageAct[21] = false;
  stageStart[21] = -1.0f;
  stageTime[21] = 0.0f;

  stageAct[24] = false;
  stageStart[24] = -1.0f;
  stageTime[24] = 0.0f;

  stageAct[25] = false;
  stageStart[25] = -1.0f;
  stageTime[25] = 0.0f;

  // This stage runs for the entire cycle
  stageStart[38] = baseTime + (0.0f * timeLine);
  stageAct[38] = false;
  stageTime[38] = timeLine * 1.250f;

  stageStart[39] = baseTime + (1.75f * timeLine);
  stageAct[39] = false;
  stageTime[39] = timeLine * 1.2500f; //3.75

  // Now we start to blend the stages
  // Extend leg forward
  stageStart[30] = baseTime + (0.0f * timeLine);
```

```
stageAct[30] = false;
stageTime[30] = timeLine * 1.0f;

// Lower other leg
stageStart[31] = baseTime + (0.0f * timeLine);
stageAct[31] = false;
stageTime[31] = timeLine * 1.25f;

// Sway to side
stageStart[32] = baseTime + (0.0f * timeLine);
stageAct[32] = false;
stageTime[32] = timeLine * 2.0f;

// Move other leg forward
stageStart[33] = baseTime + (1.0f * timeLine);
stageAct[33] = false;
stageTime[33] = timeLine * 1.0f;

// Switched to other leg now

// Extend leg forward
stageStart[34] = baseTime + (1.75f * timeLine);
stageAct[34] = false;
stageTime[34] = timeLine * 1.0f;

// Lower other leg
stageStart[35] = baseTime + (1.75f * timeLine);
stageAct[35] = false;
stageTime[35] = timeLine * 1.25f;

// Sway to side
stageStart[36] = baseTime + (1.75f * timeLine);
stageAct[36] = false;
stageTime[36] = timeLine * 2.0f;

// Move other leg forward
stageStart[37] = baseTime + (2.75f * timeLine);
stageAct[37] = false;
stageTime[37] = timeLine * 1.0f;

// New Walk Cycle Start stage trigger
stageStart[21] = baseTime + (0.0f * timeLine);
stageAct[21] = false;
stageTime[21] = timeLine * 3.5f;

 // Turning Start stage trigger 1
```

```
    stageStart[24] = baseTime + (0.0f * timeLine);
    stageAct[24] = false;
    stageTime[24] = timeLine * 3.75f;

    // Turning Start stage trigger 2
    stageStart[25] = baseTime + (0.0f * timeLine);
    stageAct[25] = false;
    stageTime[25] = timeLine * 1.5f;

  }
  else if (timing == TIMING_STOP)
  {
    stageStart[16] = baseTime + (0.0f * timeLine);
    stageAct[16] = false;
    stageTime[16] = 1.0f;

    stageStart[1] = baseTime + (0.0f * timeLine);
    stageAct[1] = false;
    stageTime[1] = 1.0f;
  }
}
```

Figure 6.2: The setupTimings function

Now we have a function to setup the timings of the a gait sequence. What we need now is a mechanism for starting and stopping the stages. This mechanism however needs to sit inside the controlFunction of the slave.

6.4.2 Timing Control Algorithm

What we do is run a small loop at the end of the NewWalk's controlFunction function. This loop checks every stage firstly to see if that stage is finished (if the time stored in the MasterControl class is larger than the both the stageStart time and the stageTime time combined). If so, then it sets stageAct to false. Also if a stage is finished and is a stage of a certain type (i.e. a walk start cycle start or a turn signal stage) then the necessary action will be taken.

For example, if the stage number finished corresponds to the stage that signals to start a new walk cycle, then the SetupTimings function is called with either the TIMING_CYCLE or TIMING_CYCLE_TWO parameter (depending on which cycle is already running).

Also the mechanism checks to see if a stage is ready to start. It does this by checking if the stageAct is false, the stageStart time is less than the current time in MasterControl and the combination of the stageStart time and the stageTime time is less than the time in MasterControl. If so, it sets the stageAct for that stage to be true.

```
  for (int i = 0; i < NUM_STAGES; i++)
  {
    // Stage is finished
    if (stageAct[i] == true && (master->getTime () >=
(stageStart[i]+stageTime[i])))
    {
      // Stop the stage
      stageAct[i] = false;

      // Stages 20 & 21 are exit signal stages, they actually do nothing
      // But signal when the next cycle should be setup
      if (i == 20)
      {
        currentWalkFactors.stepSize = desiredWalkFactors.stepSize;
        currentWalkFactors.legLift = desiredWalkFactors.legLift;
        currentWalkFactors.stop = desiredWalkFactors.stop;

        if (currentWalkFactors.stop == false)
          setupTiming (TIMING_CYCLE_TWO);
        else
          setupTiming (TIMING_STOP);
      }
      else if (i == 21)
      {
        currentWalkFactors.stepSize = desiredWalkFactors.stepSize;
        currentWalkFactors.legLift = desiredWalkFactors.legLift;
        currentWalkFactors.stop = desiredWalkFactors.stop;

        if (currentWalkFactors.stop == false)
          setupTiming (TIMING_CYCLE);
        else
          setupTiming (TIMING_STOP);
      }
      else if (i == 16)
      {
        // End of walk
        slaveKill ();
```

```
    }
    else if (i == 5)
    {
      // Setup for walk complete, now start the cycle
      setupTiming (TIMING_CYCLE);
    }
    // turning
    else if (i == 23 || i == 25)
    {
      if (turnSlave)
        turnSlave->setupTiming (NWT_TIMING1);
    }
    else if (i == 22 || i == 24)
    {
      if (turnSlave)
        turnSlave->setupTiming (NWT_TIMING2);
    }
  }
  else
  // Stage is ready to start
  if (stageAct[i] == false && (master->getTime () >= stageStart[i]) &&
      stageStart[i] >= 0 && (master->getTime () <= (stageStart[i] + stageTime[i])))
  {
    // Start the stage
    stageAct[i] = true;
  }
}
```

Figure 6.3: The mechanism that controls the stages. Taken from NewWalk::controlFunction

6.4.3 Stage Implementations

Now we will show the implementations of each of the various stages. Since NewWalk is derived from SlaveControl, 'master' is a pointer to the global instance of MasterControl. The function Velocity2 that takes two parameters, the period and the start time respectively, generates a cosine wave that is used as a velocity profile to drive a joint.

For all these implementations, they are for the first half of a complete walk cycle. For the second half, the implementation is the same (with different stage numbers) with the exception that the left and right joints are reversed. The only exception to that is the

Sway to Other Side. For the second half of the walk cycle, the angle is the opposite (negative) of the stage in the first half (both implementations of Sway to Other Side are shown below)

6.4.3.1 Lean to Side

```
//// swayRight /////
if (stageAct[0] || stageAct[17])
{
   if (stageAct[0])
      velocity = master->Velocity2 (stageTime[0], stageStart[0]);
   else
      velocity = master->Velocity2 (stageTime[17], stageStart[17]);

   // Start a swaying motion (sway right)
   master->modifyJointVel (LEFT_ANKLE_SIDE, (swayMaxAngle * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_ANKLE_SIDE, (swayMaxAngle * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (LEFT_HIP_SIDE, (-swayMaxAngle * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_HIP_SIDE, (swayMaxAngle * velocity),
JOINT_MODIFY_ADD);
}
```

Figure 6.4: Implementation of Lean to Side component of NewWalk

6.4.3.2 Raise Leg

```
//// Lift left leg ////
if (stageAct[5])
{
   velocity = master->Velocity2 (stageTime[5], stageStart[5]);

   master->modifyJointVel (LEFT_KNEE, (legLiftAngle * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (LEFT_ANKLE_FWD, ((legLiftAngle / 2) * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (LEFT_HIP_FWD, ((legLiftAngle / 2) * velocity),
JOINT_MODIFY_ADD);
}
```

Figure 6.5: Implementation of Raise Leg component of NewWalk

6.4.3.3 Extend Leg Forward

```
//// Extend Leg Forward ////
if (stageAct[6] || stageAct[30])
{
   if (stageAct[6])
     velocity = master->Velocity2 (stageTime[6], stageStart[6]);
   else if (stageAct[30])
     velocity = master->Velocity2 (stageTime[30], stageStart[30]);

   master->modifyJointVel (LEFT_KNEE, (-legLiftAngle * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (LEFT_ANKLE_FWD,
          ((((stepSize) - (anklePlaceValue)) - (legLiftAngle / 2)) * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (LEFT_HIP_FWD,
          ((stepSize - (legLiftAngle / 2)) * velocity), JOINT_MODIFY_ADD);
}
```

Figure 6.6: Implementation of Extend Leg Forward component of NewWalk

6.4.3.4 Lower Other Leg

```
//// Lower right leg //// (step3)
if (stageAct[7] || stageAct[31])
{
   if (stageAct[7])
     velocity = master->Velocity2 (stageTime[7], stageStart[7]);
   else if (stageAct[31])
     velocity = master->Velocity2 (stageTime[31], stageStart[31]);

   master->modifyJointVel (RIGHT_ANKLE_FWD, (ankleMoveValue *
velocity), JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_KNEE, (kneeBendAngle * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_HIP_FWD, (hipMoveValue * velocity),
JOINT_MODIFY_ADD);
}
```

Figure 6.7: Implementation of Lower Other Leg component of NewWalk

6.4.3.5 Sway to other side (First Half of Walk Cycle)

```
//// swayRightCentreLeft /////
if (stageAct[19] || stageAct[32])
{
   if (stageAct[19])
     velocity = master->Velocity2 (stageTime[19], stageStart[19]);
   else if (stageAct[32])
     velocity = master->Velocity2 (stageTime[32], stageStart[32]);

   // Continue swaying motion (centre it)
   master->modifyJointVel (LEFT_ANKLE_SIDE, (-swayMaxAngle * velocity *
2), JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_ANKLE_SIDE, (-swayMaxAngle * velocity
* 2), JOINT_MODIFY_ADD);
   master->modifyJointVel (LEFT_HIP_SIDE, (swayMaxAngle * velocity * 2),
JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_HIP_SIDE, (-swayMaxAngle * velocity * 2),
JOINT_MODIFY_ADD);
}
```

Figure 6.8: Implementation Sway to Other Side component of NewWalk (First half of walk cycle)

6.4.3.6 Sway to other side (Second Half of Walk Cycle)

```
if (stageAct[18] || stageAct[36])
{
   if (stageAct[18])
     velocity = master->Velocity2 (stageTime[18], stageStart[18]);
   else if (stageAct[36])
     velocity = master->Velocity2 (stageTime[36], stageStart[36]);

   // Continue swaying motion (centre it)
   master->modifyJointVel (LEFT_ANKLE_SIDE, (swayMaxAngle * velocity *
2), JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_ANKLE_SIDE, (swayMaxAngle * velocity *
2), JOINT_MODIFY_ADD);
   master->modifyJointVel (LEFT_HIP_SIDE, (-swayMaxAngle * velocity * 2),
JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_HIP_SIDE, (swayMaxAngle * velocity * 2),
JOINT_MODIFY_ADD);
}
```

Figure 6.9: Implementation Sway to Other Side component of NewWalk (Second half of walk cycle)

6.4.3.7 Straighten Body

```
//// Straighten body and torso //// (step5)
if (stageAct[9] || stageAct[38])
{
   if (stageAct[9])
   {
     velocity = master->Velocity2 (stageTime[9], stageStart[9]);

     master->modifyJointVel (LEFT_ANKLE_FWD,
       ((anklePlaceValue - (stepSize)) * velocity), JOINT_MODIFY_ADD);
     master->modifyJointVel (LEFT_HIP_FWD, (-stepSize * velocity),
JOINT_MODIFY_ADD);
     master->modifyJointVel (RIGHT_HIP_FWD, (-hipMoveValue * velocity),
JOINT_MODIFY_ADD);
     master->modifyJointVel (RIGHT_ANKLE_FWD,
       ((ankleMoveValue / 3) * velocity), JOINT_MODIFY_ADD);  //5 * (PI / 180);
     master->modifyJointVel (TORSO_FWD, (-torsoLeanFwd * velocity),
JOINT_MODIFY_ADD);
   }

   if (stageAct[38])
   {
     velocity = master->Velocity2 (stageTime[38], stageStart[38]);

     master->modifyJointVel (LEFT_ANKLE_FWD,
       ((anklePlaceValue - (stepSize)) * velocity), JOINT_MODIFY_ADD);
     master->modifyJointVel (LEFT_HIP_FWD, (-stepSize * velocity),
JOINT_MODIFY_ADD);
     master->modifyJointVel (RIGHT_HIP_FWD, (-hipMoveValue * velocity),
JOINT_MODIFY_ADD);
     master->modifyJointVel (RIGHT_ANKLE_FWD,
       ((ankleMoveValue / 3) * velocity), JOINT_MODIFY_ADD);  //5 * (PI / 180);
     master->modifyJointVel (TORSO_FWD, (-torsoLeanFwd * velocity),
JOINT_MODIFY_ADD);
   }
}
```

Figure 6.10: Implementation of Straighten Body component of NewWalk. Since these two stages run simultaneously, they need to be separated to use two different velocity profiles, even though it's the same code

6.4.3.8 Move Other Leg Forward

```
//// Move right leg forward //// (step6)
if (stageAct[10] || stageAct[33])
{
   if (stageAct[10])
     velocity = master->Velocity2 (stageTime[10], stageStart[10]);
   else if (stageAct[33])
     velocity = master->Velocity2 (stageTime[33], stageStart[33]);

   master->modifyJointVel (RIGHT_ANKLE_FWD,
     (((legLiftAngle / 2) -  (ankleMoveValue + (ankleMoveValue / 3))) * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_HIP_FWD,
     ((legLiftAngle / 2) * velocity), JOINT_MODIFY_ADD);
   master->modifyJointVel (RIGHT_KNEE,
     ((legLiftAngle - kneeBendAngle) * velocity), JOINT_MODIFY_ADD);
}
```

Figure 6.11: Implementation Move Other Leg Forward component of NewWalk

6.4.3.9 (Swap legs and repeat)

As stated above, the implementation for the second half of a single walk cycle is simply the same implementation except that left and right are reversed (i.e. LEFT_KNEE becomes RIGHT_KNEE etc.).

6.4.3.10 Drop Leg

```
// Drop left leg
if (stageAct[16])
{
   velocity = master->Velocity2 (stageTime[16], stageStart[16]);

   master->modifyJointVel (LEFT_KNEE, (-(legLiftAngle) * velocity),
JOINT_MODIFY_ADD);
   master->modifyJointVel (LEFT_ANKLE_FWD, (-(legLiftAngle / 2) *
velocity), JOINT_MODIFY_ADD);
   master->modifyJointVel (LEFT_HIP_FWD, (-(legLiftAngle / 2) * velocity),
JOINT_MODIFY_ADD);
}
```

Figure 6.12: Implementation Drop Leg component of NewWalk

6.4.3.11 Lean to Centre

```
//// swayRightCentre /////
if (stageAct[1])
{
    velocity = master->Velocity2 (stageTime[1], stageStart[1]);

    // Continue swaying motion (centre it)
    master->modifyJointVel (LEFT_ANKLE_SIDE, (-swayMaxAngle * velocity),
JOINT_MODIFY_ADD);
    master->modifyJointVel (RIGHT_ANKLE_SIDE, (-swayMaxAngle * velocity),
JOINT_MODIFY_ADD);
    master->modifyJointVel (LEFT_HIP_SIDE, (swayMaxAngle * velocity),
JOINT_MODIFY_ADD);
    master->modifyJointVel (RIGHT_HIP_SIDE, (-swayMaxAngle * velocity),
JOINT_MODIFY_ADD);
}
```

Figure 6.13: Implementation Drop Leg component of NewWalk

6.4.3.12 Walk Cycle and Turn Signal Stages

These stages do not have an implementation like the stages above. Instead they simply run and are caught by the stage control mechanisms. In essence they are dummy stages.

6.4.4 DesiredWalkFactors and CurrentWalkFactors

Finally in the implementation of NewWalk there is a structure called WalkFactors.

```
//////////////////////////////////////////////////
/// Structure for holding the changing variables of the walk
//////////////////////////////////////////////////
typedef struct _walkfactorsstruct {
   /// Size of step
   float stepSize;

   /// Amount of lift leg
   float legLift;

   /// True if want robot to stop walking
   bool stop;
} WalkFactors;
```

Figure 6.14: WalkFactors structure. Taken from the NewWalk class definition (NewWalk.h)

We use two instances of this structure, called desiredWalkFactors and currentWalkFactors. These are used to control a walk in progress.

DesiredWalkFactors is exactly as the name suggests. It is the instance that is exposed to the world and can be modified by any other part of the entire program. The idea is that if the walk needs to be manipulated (i.e. larger step size, different leg lift height or just to stop the walk) the necessary 'factor' in desiredWalkFactors is modified.

CurrentWalkFactors is the opposite of desiredWalkFactors. It is the current configuration of the walk in progress. It cannot be modified by anything outside of NewWalk.

While a walk is running, the walk cycle is continually starting, running and finishing. When a walk cycle is finished, the factors in currentWalkFactors are updated from desiredWalkFactors. This is best done at the end of a walk cycle since modifying the walk mid-cycle can lean to the robot not finishing a cycle in the correct position (maybe

only one or two degrees out with respect to a single joint, but this error will accumulate every cycle, resulting in a serious error or a fall quite fast).

```
// Stages 20 & 21 are exit signal stages, they actually do nothing
// But signal when the next cycle should be setup
if (i == 20)
{
 currentWalkFactors.stepSize = desiredWalkFactors.stepSize;
 currentWalkFactors.legLift = desiredWalkFactors.legLift;
 currentWalkFactors.stop = desiredWalkFactors.stop;

 if (currentWalkFactors.stop == false)
   setupTiming (TIMING_CYCLE_TWO);
 else
   setupTiming (TIMING_STOP);
}
else if (i == 21)
{
 currentWalkFactors.stepSize = desiredWalkFactors.stepSize;
 currentWalkFactors.legLift = desiredWalkFactors.legLift;
 currentWalkFactors.stop = desiredWalkFactors.stop;

 if (currentWalkFactors.stop == false)
   setupTiming (TIMING_CYCLE);
 else
   setupTiming (TIMING_STOP);
}
```

Figure 6.15: Extract from the code to control the running of stages. This section updates currentWalkFactors and responds to changes in the walk factors

6.5 NewWalk's Binding Variables

As stated previously, there are a number of variables that allow the different components of the NewWalk algorithm to be binded together.
These binding variables are as follows:

- kneeBendAngle
- anklePlaceValue
- ankleMoveValue
- hipMoveValue

```
float legLiftAngle = currentWalkFactors.legLift;
float stepSize = currentWalkFactors.stepSize;

// Loaded from a file, fine-tuned using a gaNewWalk
float kneeBendAngle = (stepSize / 20.0f) * KNEE_BEND_ANGLE_VAR;
float anklePlaceValue =  (stepSize / 20.0f) * ANKLE_PLACE_VALUE_VAR;
float ankleMoveValue = (stepSize / 20.0f) * ANKLE_MOVE_VALUE_VAR;
float hipMoveValue = (stepSize / 20.0f) * HIP_MOVE_VALUE_VAR;

// These two are independant of the others
float torsoLeanFwd = 0.0f;
float swayMaxAngle = 6.5;

// Convert to radians
stepSize *= (PI / 180);
kneeBendAngle *= (PI / 180);
torsoLeanFwd *= (PI / 180);
swayMaxAngle *= (PI / 180);
legLiftAngle *= (PI / 180);
anklePlaceValue *= (PI / 180);
ankleMoveValue *= (PI / 180);
hipMoveValue *= (PI / 180);
```

Figure 6.16: Code to setup the variables that bind the stages together. From NewWalk::controlFunction

As you can see in the code above, these numbers are loaded from a file. This is since these numbers are currently loaded from a file. This means we can use an offline calculation to generate these numbers.

6.6 Results Using Hand-Tuned Variables

Initially before a program was written using genetic algorithm techniques, the four variables that bind the stages of the walk together were fine tuned by hand.
Although this didn't give optimum results, it did give satisfactory results since if parts of the walk sequence are changed (i.e. timings etc.) then these numbers will need to be recalculated, something that isn't feasible when fine-tuning using a genetic algorithm can take anywhere from twenty-four to seventy-two hours to complete.

Also by showing the results of using both manually tuned numbers and automatically generated numbers (using a genetic algorithm), it shows the potential for using automated techniques for fine tuning not just these aspects of the walk, but extending it to calculate the timings of the walk (since timings are currently calculated manually at the moment) and possibly other aspects in the future.

We measure the stability of a walk using the Zero Moment Point principle (discussed in the background chapter earlier).

Below you can see graph of the movement of the zero moment point in both the x and y axis'. Ideally, the movement of the ZMP in the x-axis should be a smooth sine wave oscillation and the movement of the ZMP in the y-axis should be a smooth linear line with no oscillations at all (since oscillations in the y-axis of the ZMP indicate a rocking motion of the robot which leads to instability in the robot).

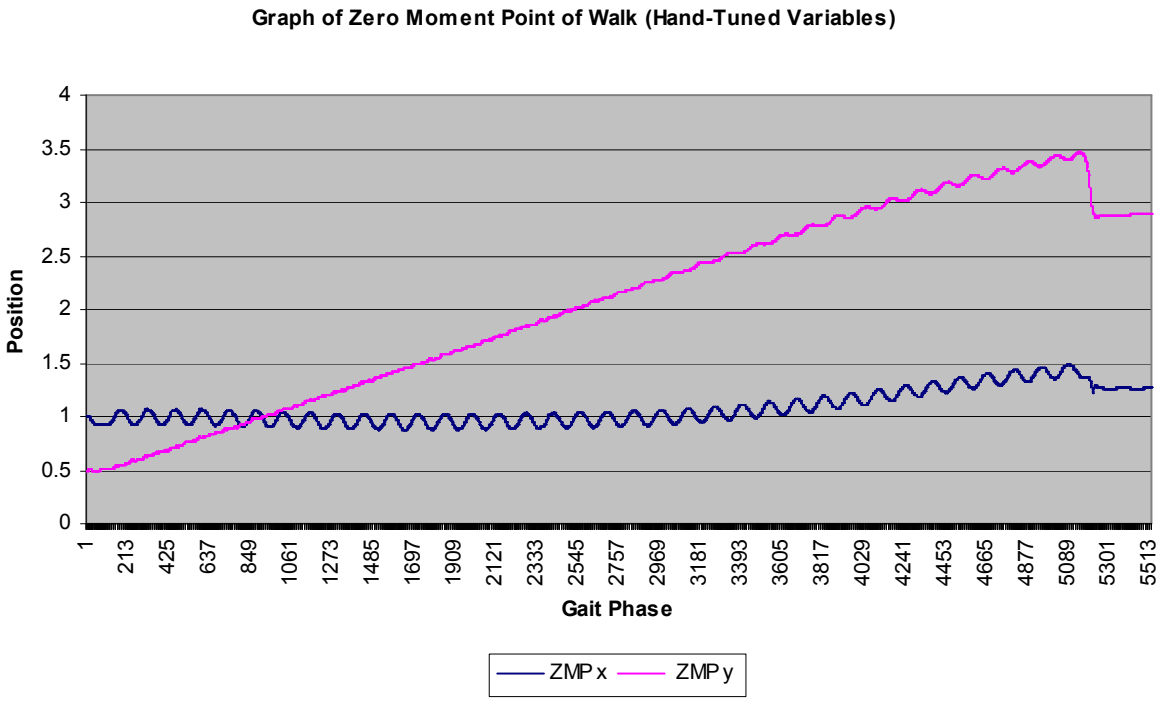

Figure 6.17: Graph of the Zero Moment Point positions during a walk using Hand-Tuned variables

6.7 Automated Fine-tuning

Since it is very difficult to determine what the best values are for the variables that bind the stages together, we use an offline calculation in order to determine the best values.

In order to do this, we need two things. Firstly we need a genetic algorithm in which to calculate the best set of values based on a given set of scores. Secondly we need to extend the simulator to provide a fitness function that will accept different values for the four variables and give a score based on how well the walk does compared to a 'perfect' walk. We call the scoring function the fitness function.

6.7.1 NewWalkGA

Because all slaves are written as classes in C++, we can use inheritance to help us write out fitness function. So we create a slave called NewWalkGA that is derived from NewWalk. We write a new controlFunction function that takes the current position of the zero moment point (ZMP) into account and compares it to a pre-calculated 'perfect' position of the ZMP (for that particular point in time). The function also makes a call to the original NewWalk controlFunction to run the actual walk. The calculated comparisons are then accumulated, averaged and weighted to give a score for the walk. Note that this slave is currently designed to run in the simulator only (due to lack of sensor information onboard the robot).

```
////////////////////////////////////
// Control function, called by MasterControl
// All the gait generation work is done here
////////////////////////////////////
void NewWalkGA::controlFunction ()
{
   //*****************************************//
   // Fitness Function
   //*****************************************//
   // Zero Moment Point
   float zmp[2];
   // Centre of Gravity
   float cg[2];
```

```
    // Deviation from known good zmp
    float zmpDeviation = 0;

    // Get the zero moment point value
    zmp_cg_calc(&zmp[0], &zmp[1], &cg[0], &cg[1]);

    // Compare against known good zmp
    float devix = fabs (zmpfitx[compareCount] - zmp[0]);
    float deviy = fabs (zmpfity[compareCount] - zmp[1]);

    if (compareCount > 102)
    {
       float dx = fabs (devix - preZmpX);
       fitnessx += dx;
       preZmpX = devix;

       float dy = fabs (deviy - preZmpY);
       fitnessy += dy;
       preZmpY = deviy;
    }
    else
    {
       // do nothing
    }

    // Call the newwalk control function
    NewWalk::controlFunction ();

    // Increase compare count (and total run count also)
    compareCount++;

    // When finishing
    if (compareCount >= MAX_COUNT)
    {
       fitnessx = ((float)fitnessx) / ((float)(MAX_COUNT - 102));
       fitnessy = ((float)fitnessy) / ((float)(MAX_COUNT - 102));

       fprintf(stdout, "%f\n", ((0.5 * fitnessx) + (0.5 * fitnessy)));

       // Set slave as finished
       this->finished = true;
    }
 }
```

Figure 6.18: controlFunction of the NewWalkGA Slave. Note that this also calls NewWalk's orginal controlFunction

6.7.2 GANewWalk

Now that we have a slave to run inside the simulator program to give the score of a particular walk, or more importantly, to give a score of a particular set of values (for the binding variables), we need a genetic algorithm to determine the best set of values.

The genetic algorithm we use is written using the GAlib library written by Matthew Wall. This library allows us to use a genetic algorithm without having to write all the internal functions ourselves. All we need to do is write an objective function and to setup the genetic algorithm library to run.

The objective function we use is simply to output the set of values for any given individual to a file and run the simulator program (the simulator is specially compiled to run with graphics off and to only run the NewWalkGA slave) that will load the values from the file.

```
float
objective(GAGenome & c)
{
        static int          count = 0;
        int       i;

        GABin2DecGenome & genome = (GABin2DecGenome &)c;

        float y;

        for (i=0; i<NUM_VARS; i++)
   {
                vars[i] = (float)genome.phenotype(i);
   }

        y = (float)(100 * do_run());

   fprintf (allScores, "%f, %f, %f, %f, %f\n", y, vars[0], vars[1], vars[2], vars[3]);

        count++;
        fprintf(stderr, "%04d: %f\n", count, y);
        return (y);
}
```

Figure 6.19: The Objective function of our genetic algorithm

Our objective function relies on a function called do_run. This function writes the values stored in an individual to a file and then runs our specially modified version of simulator

program (graphics off, using NewWalkGA slave). Once the simulator has finished it outputs a score that is passed back to the objective function. Now we have a score for a given individual.

Now all we have to do is setup the necessary components of our genetic algorithm. Our setup includes code to log all the data that the genetic algorithm is using (we log all the scores of all the individuals plus the four values that they each contain, plus we log all the best scores of each generation and the final set of values calculated to be the best), plus we specify the range for our values to be.
We also set how large we want our population to be, how many generations to run, how much mutation will occur during the running of the genetic algorithm and finally how much cross-over will occur.
We also need to specify that we want to minimize the scores that we get, i.e. the best scores are the lowest ones. This is since we use the deviation from the perfect ZMP to calculate our score, so the walk with the least deviation from the perfect ZMP will be the best walk.

The graph of the zero moment point is shown below. Notice that the first one hundred and two values are zero since they are ignored by the fitness function. This is since NewWalk uses the first hundred and two counts of the gait phase to setup the walk.

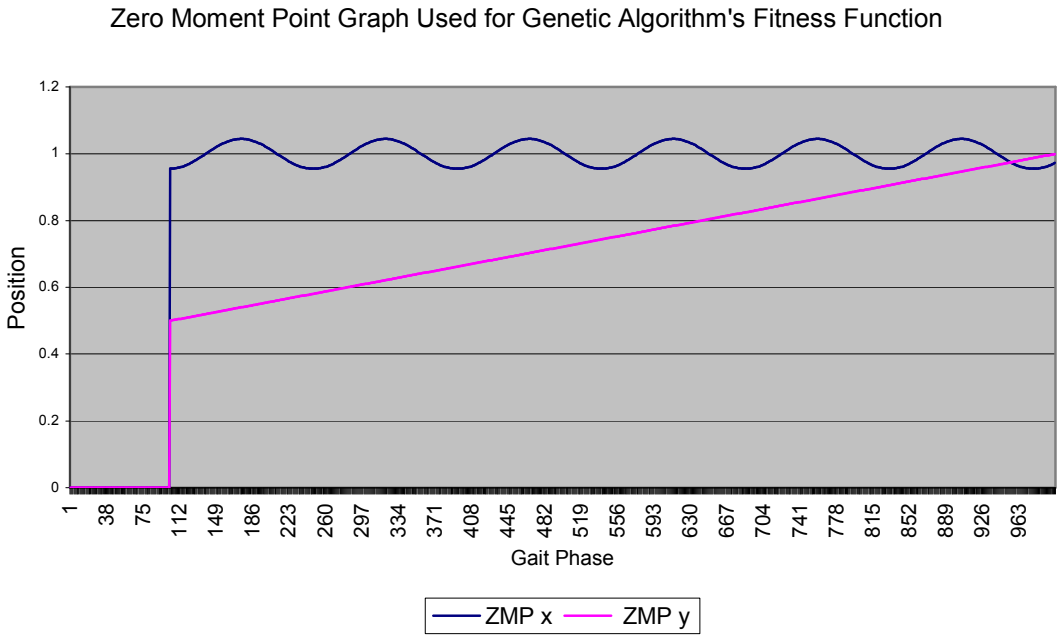

Figure 6.20: Graph of zero moment point used in fitness function

As you can see, it's a perfect sine curve for the x-axis and a straight linear line for the y-axis.

Once we have set all this up, all we need to do to calculate our best set of four values for our binding variables is to start the genetic algorithm. Now we can run the program, which can take anywhere from twelve hours to seventy-two hours (dependant on processing power of the computer which it is run on, the size of our population and finally the number of generations we run).

Once it is finished it will give us the four best values to use. These values may not be the best possible values to use, since the initial population is randomly created, but usually they are one the best possible combinations that can be found (there may be many combinations that work sufficiently).

```
int
main(int argc, char **argv)
{
        unsigned int seed = 0;

        // Declare variables for the GA parameters and set them to some default
        //values.

        int     i;
        int     popsize  = 150;
        int     ngen     = 75;
        float   pmut   = 0.0050;
        float   pcross = 0.6;

        // Create a phenotype for gains.

        GABin2DecPhenotype map;

        map.add(16, 20, 35);
        map.add(16, 20, 40);
        map.add(16, 10, 25);
        map.add(16, 10, 25);

        allScores = fopen ("scores.csv", "w");
        fprintf (allScores, "scores, knee, ankle place, ankle move, hip move\n");

        // Create the template genome using the phenotype map we just made.
        GABin2DecGenome genome(map, objective);

        // Now create the GA using the genome and run it.  We'll use sigma
        // truncation scaling so that we can handle negative objective scores.
        GASimpleGA ga(genome);
        ga.minimize();
        GASigmaTruncationScaling scaling;
```

```
        ga.populationSize(popsize);
        ga.nGenerations(ngen);
        ga.pMutation(pmut);
        ga.pCrossover(pcross);
        ga.scaling(scaling);
        ga.selectScores(GAStatistics::AllScores);
        ga.scoreFilename("ganwrun.dat");
        ga.scoreFrequency(1);
        ga.flushFrequency(1);
        ga.evolve(seed);

        // Dump the results of the GA to the screen.
        genome = ga.statistics().bestIndividual();
        fprintf(stdout, "Final result:\n");
        for (i=0; i<NUM_VARS; i++)
                fprintf(stdout, "%d: %f\n", i, (float)genome.phenotype(i));

        cout << "best of generation data are in '" << ga.scoreFilename() << "'\n";

        // Print to a log file
        FILE *vars = fopen ("nwgavars.cfg", "w");
         for (i=0; i<NUM_VARS; i++)
        {
                 fprintf (vars, "%f\n", (float)genome.phenotype(i));
        }
        fclose (vars);
        fclose (allScores);
        return 0;
}
```

Figure 6.21: Code to setup and run the genetic algorithm to calculate the best values for the four binding variables

6.8 Results from automated fine tuning

Although we only use a genetic algorithm to fine tune four variables, it shows a marked improvement in robot stability.
Using the hand-tuned variables, as seen above, we see the zero moment point of the robot in the x-axis, move the position on the sine curve around in the vertical direction. This is mostly due to the instabilities that occur when the variables aren't properly tuned (i.e. if the ankle place value isn't properly tuned, the foot will not impact correctly and the robot will at best cause small impact shakes and at the worst will not allow the robot to move its weight forward onto that foot).

The results of the genetic algorithm calculation are shown below. Firstly we will show the progression of scores for the individuals over the desired number of generations. These scores show the max score, minimum score and best score for each generation. It also shows the standard deviation as well.

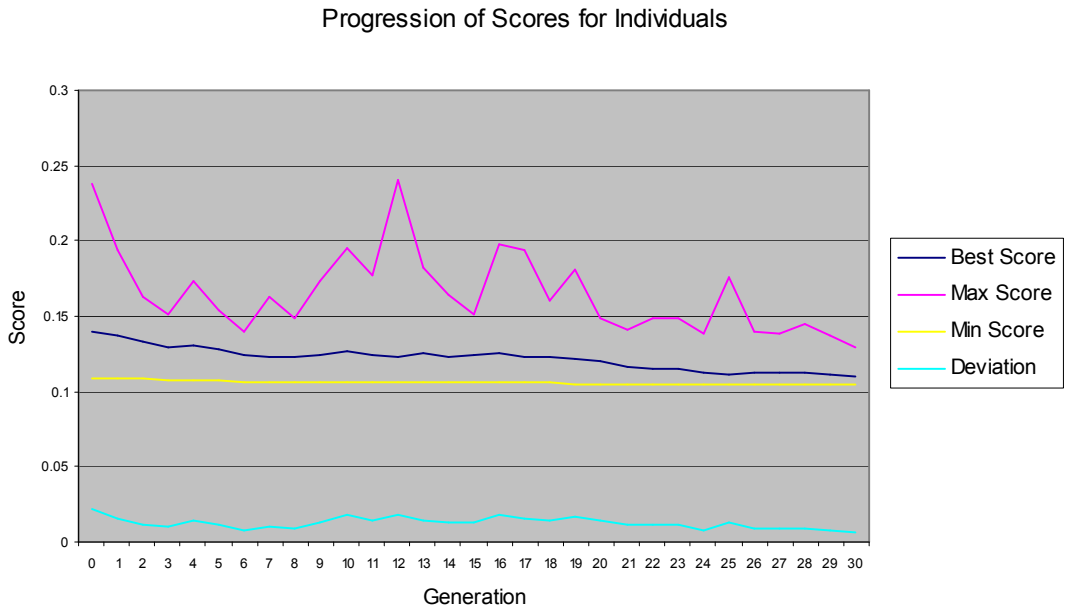

Figure 6.22: Graph showing the Progression of Scores for individuals over all generations of a single run of the genetic algorithm used to find the best values for the four stage binding values

Now we see the final results after this calculation has been run.

Knee Bend Angle	22.272373
Ankle Place Value	39.250172
Ankle Move Value	16.447701
Hip Move Value	23.228657

Figure 6.23: List of best values found for the four stage binding variables calculated using a genetic algorithm

Its possible that there are many combinations of these variables that will yield stable results from the robot. This is improved upon by improving the quality and accuracy of the fitness function.

Now we will see the zero moment point graph of the walk, taken from the simulator program.

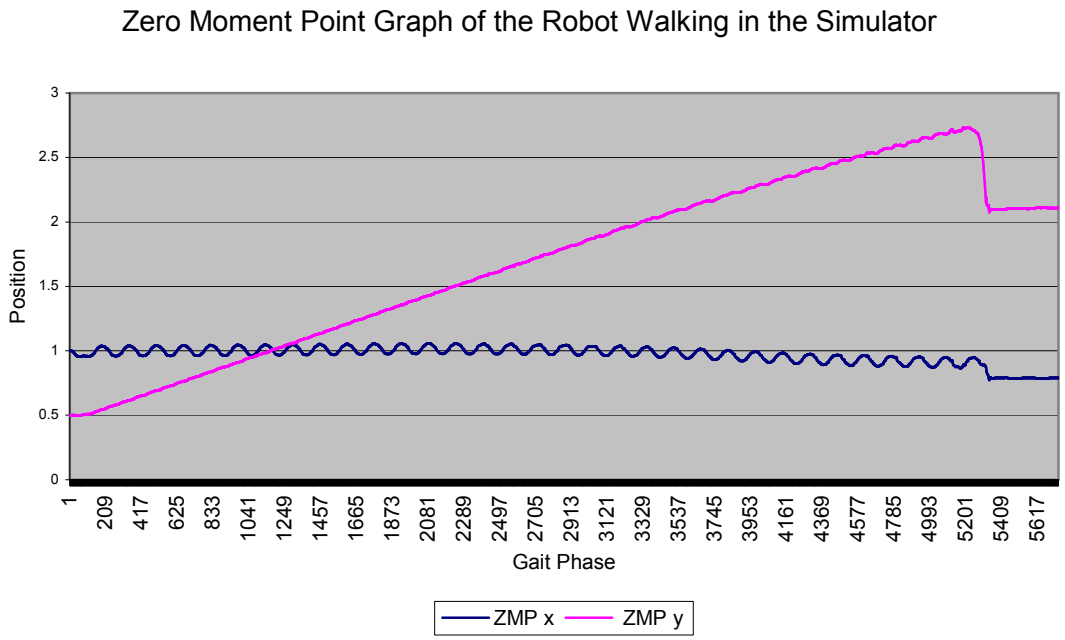

Figure 6.24: Zero Moment Point graph of the GuRoo robot walking in the simulator program

As you can see, the stability of the robot is good for most of the walk and is able to take many steps before losing its stability. The move in the position of the graph of the zero moment point is not due to the walk, but due to problems in the foot contact model and inaccuracies in describing the weight distribution of the robot.

The fall is mostly caused by oscillations in the y-axis of the zero moment point graph. The cause of this oscillation is currently unknown, but is most likely due to the inability of the robot to adjust to momentum while the robot is moving. To overcome this, the robot might need to increase its speed accordingly in order to maintain stability.

7 NWTurn

By using NewWalk, we are able to satisfy one of our primary goals for this thesis. However, a secondary goal of this thesis is to develop the ability of the robot to turn and walk. This slave, called NWTurn (for NewWalk Turn), attempts to achieve this goal.

Previously, if we were to attempt to implement turning and walking, we would have to have written a behaviour that encompasses both the walking and turning aspects. However, with the new MasterSlave system that has been implemented as part of this thesis, we are able to separate the walking and turning behaviours into two separate behaviours. This is important, since it means that NWTurn can be used in conjunction with any other walk that may be / have been developed.

7.1 Design

Like NewWalk, NWTurn is design by breaking down the components of a turn into various stages. However, NWTurn only uses a single stage to accomplish turning. This stage uses the right leg twist joint.

7.2 Implementation

The implementation of NWTurn is very similar to NewWalk. It uses a slightly modified implementation of the timing system used in NewWalk, augmented to allow the timings to be controlled not from inside the controlFunction of NWTurn, but from another slave. It also contains a function that sets up the initial turning parameters (turn angle, period of each turning movement and the speed at which to turn).

```
void NWTurn::setupTurn (float angle, float period, float speed)
{
  destAngle = angle;
  this->period = period;
  this->speed = speed;
  currentAngle = 0.0f;
  turningAngle = 0.0f;
}
```

Figure 7.1: Function to setup the parameters of a turn

The idea behind NWTurn is that it will always run like a normal slave, i.e. be registered with MasterControl which then runs NWTurn's controlFunction. However, the controlFunction will not do anything until another slave calls its setupTiming function.

So in order to use NWTurn, we need to add extra stages to NewWalk which signal when to call the setupTiming function of the NWTurn slave (NewWalk will have a pointer to a NWTurn slave). These extra stages are described earlier in the chapter about NewWalk.

```
void NWTurn::setupTiming (int timing)
{
   // No angle to turn to
   if (destAngle == 0)
      return;

   float baseTime = master->getTime ();

   // Set the timings for the turn
   stageAct[0] = true;
   stageStart[0] = baseTime;
   stageTime[0] = period;

   if ((currentAngle >= (fabs (destAngle))))
   {
      // Finished turning, reset parameters
      destAngle = 0.0f;
      period = 0.0f;
      speed = 0.0f;
      currentAngle = 0.0f;
      turningAngle = 0.0f;
```

```
        return;
    }

    // Left leg part of turn
    if (timing == NWT_TIMING1)
    {
        turningAngle = (float)fabs(destAngle) / speed;
        currentAngle += turningAngle;
    }
    // Right leg part of turning
    else if (timing == NWT_TIMING2)
    {
        turningAngle = (float)fabs(destAngle) / speed;
        currentAngle += turningAngle;
        turningAngle *= -1;
    }
}
```

Figure 7.2: Code to setup the timings of NWTurn. This function is usually called by another slave

7.3 Results using NewWalk and NWTurn

Below we can see a graph of the Zero Moment Point while a turn is occurring. This is taken from the simulator program. As you can see, the zero moment point in the x-axis is supposed to deviate downwards, this indicates that the robot has turned.

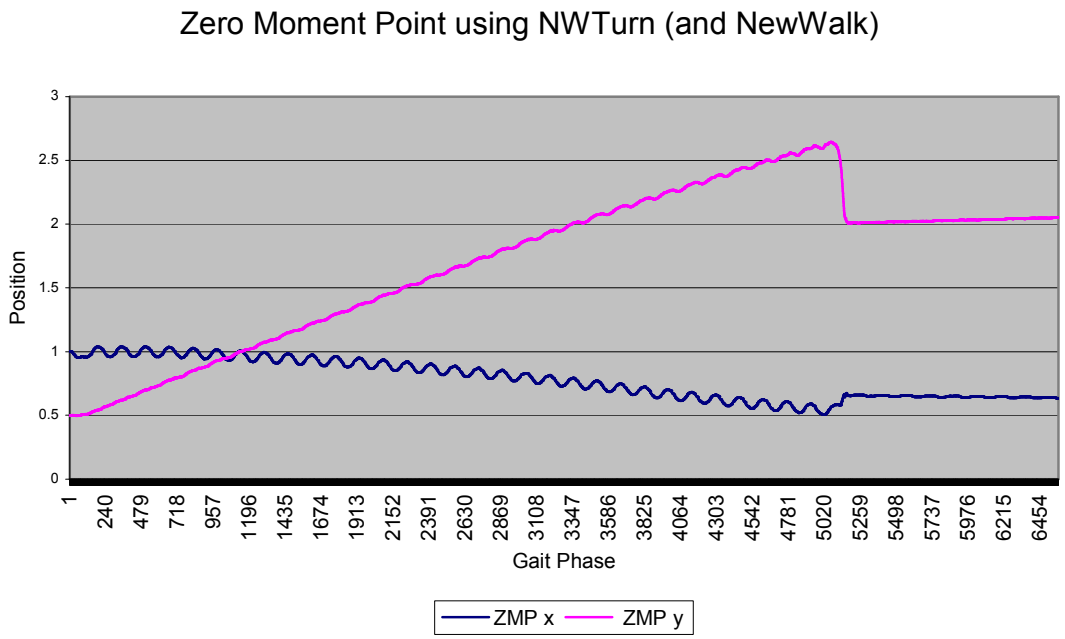

Figure 7.3: Graph of the position of the zero moment point from a walk (NewWalk) using NWTurn to turn the robot twenty degrees

As you can see, the robot still moves the same distance as a normal walk (using NewWalk), so this shows that NWTurn provides minimal instability to the walk, which is what we desire.

8 Evaluations and Future Work

8.1 Evaluation

8.1.1 MasterSlave

The MasterControl class, being the brains of the MasterSlave system, achieves a number of the desired goals of the MasterSlave system.

This goals achieved include:

- Allowing more than one behaviour running at any given time
    - Any slave derived from SlaveControl can be registered and run
- Allowing more than one instance of any behaviour to run
    - Any slave can be registered and run, even different instances of the same class since MasterControl does not do any type checking
- Providing a single well controlled conduit to allow manipulation of joints
    - To set joint velocities
    - Tracking current joint positions
- Providing a cleaner mechanism for allowing behaviours to interact
    - With the robot
    - Reduces chances of 'wild' behaviours that can cause chaos
- Provide a suitable base for future behaviour management system
    - Update function currently runs slaves in first to register, first to run order
    - However, since all slaves are based on standard base class and list of registered slaves is kept, a system to order or prioritise slaves can be implemented in/on top of MasterControl

The SlaveControl class achieves our remaining goals of the MasterSlave system:

- Force all the behaviour code to be encapsulated inside a single class
    - All code must be contained inside classes

    - All behaviour code is kept in separate files which can easily be updated in the main code base
- Provide a cleaner mechanism for allowing behaviours to interact
    - With each other
    - With the MasterControl class (using a common known interface)
- Provide a common base for developing behaviours
    - Base class for all slaves
    - Gives a 'running-start' to building slaves

As you can see, the MasterSlave behaviour control system is able to achieve all the goals that we set out for it.

8.1.2 NewWalk

The primary goal of NewWalk was to allow the GuRoo robot to travel stably in a straight line. Also the goal was develop NewWalk so that the walk was configurable in real-time.

NewWalk is able to achieve the goal of real-time reconfiguration. A walk can be started, stopped and the step size and leg lift height are able to be changed during run-time.

Using the simulator program we partially achieve the goal of having a stable straight-line walk. Although the robot only walks a short distance (two to three metres), it still is a stable walk. Although most of this instability is due to the fact that a lot of the walk is still manual-tuned. The timings are tuned by hand, something that should be tuned using a genetic algorithm or similar offline calculation technique.

We can see in the graph of the zero moment point of a walk above that it stays mostly stable for the majority of the walk. However, it tends to gain a small oscillation in the y-axis. The cause behind this instability is still unknown, although it could be anything from bad foot contact model, poor robot simulation model that fails to properly model the

weight distribution of the robot to inability of the walk to account properly for momentum.

8.1.3 NWTurn

The secondary goal of this thesis was to extend the straight-line walking algorithm to encompass turning. NWTurn does achieve this goal, however it is heavily dependent on the stability of the walk itself to ensure that it's turning is stable.

From the results above, we can see that the turning is moderately stable, however since we are turning the entire leg, we can assume that it will have an effect on the stability of the walk in progress.

Obviously, the idea behind NWTurn was that it would also use a similar set of variables to NewWalk. These variables would then use an offline calculation, probably a genetic algorithm, in order to maintain the stability during a turn. Since the turn would effect the angle that the ankles hit the surface, we need these variables to ensure the walk cycle is unaffected during a turn.

However, these ideas for NWTurn have not been implemented. Since the primary focus of this thesis is for the straight-line walking, the current stability of the turning algorithm was deemed to be suitable.

8.2 Future Work

Although this thesis achieved its goals, there is still although of work to be done in order for the goals of the GuRoo project to be achieved. We will now look at the future work that needs to be done from the perspective of the work done on this thesis.

8.2.1 MasterSlave

Although MasterSlave achieved all its goals, the system however is still only a base for a more sophisticated system. This is simple to see since we have always described the MasterSlave system as a behaviour 'control' system. Thusly it does not concern itself with what behaviours are being run, just ensuring that they are all run successfully.
On top the MasterSlave system, a behaviour 'management' system would be the next logical step. This management system would then decide which behaviours can run and what type of actions they can perform on which joints.
Also a more sophisticated system for allowing the behaviours to access sensor information should implemented as part of this management system.

8.2.2 NewWalk

Clearly NewWalk does allow the robot to walk stably in a straight-line for a short period of time. However, the goal is to allow the robot to walk stably for a much longer period of time. One of these ways would be to move all the timings that are currently manual-tuned to be tuned using an offline calculation (i.e. genetic algorithm). An offline calculation would then be better able to determine the best combination of all these variables to perform a nice stable walk.

Also, given that the robot is now beginning to gain a variety of sensors (including inertia sensors and gyroscopes) we can use the data from these sensors to adjust the walk in real-time.

8.2.3 NWTurn

As stated earlier, the NWTurn algorithm needs to be extended to use a number of variables in a manner to NewWalk. These variables would be calculated offline in order to offset the instability that turning may cause.

Also, a possible extension would be allowing the algorithm to use sensor feedback in order to determine the current heading of the robot. Currently the turning algorithm assumes that the turn is always successful, however by using actual data from the robot, the algorithm will always turn to the specified angle.

9 Conclusions

As stated above, this thesis project was moderately successful in achieving its goals. The MasterSlave behaviour control system, developed as part of this thesis, extended the robot to allow multiple behaviours to run simultaneously on the robot. This allows for much more sophisticated combination of behaviours to be performed on the robot with ease. Also be encasing all the behaviours inside a C++ class based system, we are able to maintain a neat code structure.

The results of the NewWalk algorithm showed that it was moderately successful in walking in a stable straight line. Although there are a number of problems with the walk, it does provide not just a suitable base to build upon to create a stable walk, but the infrastructure that NewWalk is built upon (breakdown of stages, timings of stages, the timing control system, the use of a genetic algorithm to bind the stages together) can be used as a basis for future gait generation research.

NWTurn is also a viable basis for not just building an algorithm to provide the robot with turning capabilities, but also as a basis for future work that involves multiple behaviours working together. Clearly though, NWTurn does need more work, mostly in the areas mentioned previously in this document.

As stated, although this thesis achieved its goals and provides a useful basis for future work (especially the MasterSlave system), there is still alot of work to be done before the goals of the GuRoo project are satisfied.

10 References

[McMillan, 1995] McMillan S, Computational Dynamics for Robotic Systems on Land and Under Water, Doctoral Dissertation, Ohio State University, Department of Electrical Engineering, 1995.

[Wall, 1995] GAlib: A C++ Genetic Algorithm Library by Matthew Wall, Copyright (c) 1994-1996 MIT, 1996-2000 ,

[Smith, 2001] Simulator Development and Gait Pattern Creation for a Humanoid Robot, Andrew Smith, University of Queensland, 2001.

[Yik, 2002] Evolving a Locus Based Gait for a Humanoid Robot, Tak Fai Yik, University of New South Wales, 2002

11 Bibliography

Christopher Hing, 2002, *Development of a Dynamic Gait for a Simulated Humanoid Robot*, University of New South Wales, New South Wales

Smith, R.L, 1998, *Intelligent Motion Control with an Artificial Cerebellum*, in *Electrical and Electronic Engineering*, University of Auckland, Auckland.

M. Vukobratovic, A.A. Frank and D.Juricic, 1970, *On the stability of biped locomotion*. IEEE Transactions on Biomedical Engineering, 17(1):25-36

Brian R Durwood, Gillian D Baer, Phillip J Rowe, 1999, Functional Human Movement, , Reed Educational and Professional Publishing Ltd

Wilhelm Braune and Otto Fischer, 1987, *The Human Gait*, Springer-Verlag, Berlin Heidelberg

Adam Drury, 2002, *Gait Generation and Control Algorithms for a Humanoid Robot,* University of Queensland, Queensland

D. Kee, G. Wyeth, A. Hood and A.Drury, 2003, *GuRoo: Autonomous Humanoid Platform for Walking Gait Research*, Autonomous Minirobots for Research and Edutainment, Feb 2003, pp. 13-22

Wyeth, Kee, Wagstaff et al, 2001, *Design of an Autonomous Humanoid Robot*, Proceedings of the Australian Conference on Robotics and Automation (ACRA '01), November 14-15, Sydney 2001

12 Appendix

Contained in this appendices are all the class definitions for the MasterSlave classes (contained in MasterSlave.h), NewWalk class (in NewWalk.h), NewWalkGA class (in NewWalkGA.h) and for the NWTurn class (in NWTurn.h).
Also we have the code for the GANewWalk program that runs separately to the simulator/robot programs.

12.1 MasterSlave.h

```
/*************************************************************************
******
 * Copyright 2003, Gordon Wyeth

*************************************************************************
*****
 *     File: MasterSlave.h
 *   Author: Tim Pike
 *  Project: Humanoid 2.0 Dynamechs 4.0
 *  Created: 1st July 2003
 *  Summary: MasterSlave Behaviour Control System
 *           Rewrite of Central.cpp

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

#ifndef _MASTERSLAVE_H_
#define _MASTERSLAVE_H_

#define MAX_SLAVES 10

// Joint modification actions
#define JOINT_MODIFY_ADD 1
#define JOINT_MODIFY_SUB 2
#define JOINT_MODIFY_OVERWRITE 3

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

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

//////////////////////////////////////////////////
/// Joint Class
//////////
/// Contains all the information revelant to a particular joint.
/// Use setDesiredJointVel (..) to add velocities
//////////////////////////////////////////////////
class Joint
{
protected:
        ////////////////////
        /// Name of the joint
        ////////////////////
        char* name;

        ////////////////////
        /// The desired joint velocity (radians)
        ////////////////////
        float desired_joint_vel;

        ////////////////////
        /// Current Joint Position (radians)
        ////////////////////
        float current_joint_pos;
public:
       ////////////////////
       /// Constructor (default)
       ////////////////////
```

```
Joint ();
//////////////////////
/// Contstructor (specific joint name)
//////////////////////
Joint (char* name);

//////////////////////
/// Deconstructor
//////////////////////
~Joint ();

//////////////////////
/// Get the name of the joint
//////////////////////
char* getName () { return name; }

//////////////////////
/// Sets the desired joint velocity
//////////////////////
void  setDesiredJointVel (float velocity);

//////////////////////
/// Returns the current desired joint velocity (radians)
//////////////////////
float getDesiredJointVel ();

//////////////////////
/// Returns the current position of the joint
/// NOTE: This is the theoretical position and may not
/// reflect the actual position.
//////////////////////
```

```
        float getCurrentJointPos ();

        //////////////////////
        /// Reset the actual theoretical joint position
        //////////////////////
        void resetActualJointPos ();

        //////////////////////
        /// Updates the actual joint pos and clears the desired joint vel
        //////////////////////
        int update ();
};

/// Forward declaration of the slave control class
class SlaveControl;

////////////////////////////////////////////////////
/// MasterControl Class
////////
/// Responsible for keeping track of all the joints
/// and the slaves the are currenting controlling the joints
/// Will call the controlFunction () of the slaves each time
////////////////////////////////////////////////////
class MasterControl
{
protected:
        //////////////////////
        /// Internal array to allow easier interface with old system
        /// ie. old system uses desired_joint_vel[] as a pointer which
        /// points to msDesiredJointVel here which gets updated in Update ()
        //////////////////////
```

```
int msDesiredJointVel[TOTAL_MOTORS];

//////////////////////
/// Array of all the robots joints
/// Corresponds to desired_joint_vel[]
/// Array positions are defined in jointnum.h
//////////////////////
Joint *joints[TOTAL_MOTORS];

//////////////////////
/// Amount of time the master control has been running
//////////////////////
float time;

//////////////////////
/// Running status of the master controller
//////////////////////
bool finished;

//////////////////////
/// Array of all the slave controllers
/// Pointer will be NULL if no controller in that position
//////////////////////
SlaveControl *slaves[MAX_SLAVES];

//////////////////////
/// Number of slaves currently registered
//////////////////////
int numSlaves;

            //////////////////////
```

```
        /// Phase of the gait?
        ////////////////////
        int gait_phase;

public:
    ////////////////////
    /// Constructor
    ////////////////////
    MasterControl ();
    ////////////////////
    /// Destructor
    ////////////////////
    ~MasterControl ();

    ////////////////////
    /// Registers a slave control with the class
    /// which adds the pointer to the list of slaves
    /// which get run (by calling slave->controlFunction ()) every
    /// update ()
    ////////////////////
    bool registerController (SlaveControl *control);

    ////////////////////
    /// Remove a slave control from the list of slaves
    /// Returns true is successful, false is unsucessful
    ////////////////////
    bool unregisterController (SlaveControl *control);

    ////////////////////
    /// Kills all the slaves currently registered
    ////////////////////
```

```
void killAllSlaves ();

//////////////////////
/// Reset All the Joint Positions
//////////////////////
void resetAllJointPositions ();

//////////////////////
/// Returns a specified joint
//////////////////////
Joint *getJoint (int joint);

//////////////////////
/// Modify a desired joint velocity by a given amount
/// a given way (ie. Add, subtract, overwrite etc.)
//////////////////////
void modifyJointVel (int joint, float velocity, int action);

//////////////////////
/// Returns the current time
//////////////////////
float getTime () { return time; }

//////////////////////
/// Velocity profile. Generates cosine wave from max_acceleration and start_time
//////////////////////
float Velocity (float start_time, float acceleration);

//////////////////////
/// Velocity profile. Generates cosine wave from period and start time
//////////////////////
```

```
        float Velocity2(float per, float stage_time);

        //////////////////////
        /// Update the master controller
        /// Updates the time, calls the slave functions etc.
        //////////////////////
        void update ();

                //////////////////////
                /// Get the current gait phase
                //////////////////////
                int getGaitPhase ();

                //////////////////////
                /// Set the gait phase to a new phase?
                //////////////////////
                void setGaitPhase(int phase);

                /// increment the gait phase
                //////////////////////
                void incrementGaitPhase();
};

////////////////////////////////////////////////////
/// SlaveControl Class
////////
/// Generic class to hold all the robot control code
/// When a instance of this class (or a deriative of it) is
/// registered with the MasterControl class, the controlFunction
/// function will be called everytime by the MasterControl::update ()
/// function.
```

```
//////////////////////////////////////////////////
class SlaveControl
{
protected:
        ////////////////////
        /// Name of the slave
        ////////////////////
        char* name;
        ////////////////////
        /// Pointer to the master controller
        ////////////////////
        MasterControl *master;

        ////////////////////
        /// Running status of the slave
        ////////////////////
        bool finished;
public:
        ////////////////////
        /// Constructor (default, do not use)
        ////////////////////
        SlaveControl ();

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

        ////////////////////
```

```
        /// Deconstructor
        //////////////////////
        virtual ~SlaveControl ();

        //////////////////////
        /// Return the slaves name
        //////////////////////
        char* getName () { return name; }

        //////////////////////
        /// Check if the slave is finished
        //////////////////////
        bool isFinished () { return finished; }

        //////////////////////
        /// Stops the slave and unregisters it from the master controller
        //////////////////////
        void slaveKill ();

        //////////////////////
        /// This function is called when the slave is registered
        /// with a master controller and finished is false.
        /// Called by update function of the master controller
        //////////////////////
        virtual void controlFunction () = 0;
};

#endif // _MASTERSLAVE_H_
```

12.2 NewWalk.h

```
/*****************************************************************************
******
 * Copyright 2003, Gordon Wyeth

*****************************************************************************
*****
 *     File: NewWalk.h
 *   Author: Tim Pike
 *  Project: Humanoid 2.0 Dynamechs 4.0
 *  Created: 19th July 2003
 *  Summary: Encapsulation of NewWalk algorithm inside MasterSlave system
 *           using SlaveControl class.

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

#ifndef _NEWWALK_H_
#define _NEWWALK_H_

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

/// Number of stages in gait
#define NUM_STAGES 40

/// Timing defines for timing cycles
#define TIMING_FIRST 1
#define TIMING_CYCLE 2
#define TIMING_CYCLE_TWO 3
```

```
#define TIMING_STOP 4

/// Simulator only stuff
#define SIMULATOR

#ifdef SIMULATOR

#define ZMP_AVG_DIV 2
//#define ZMP_CALC_MAG

#endif

/// Forward declaration of NWTurn class
class NWTurn;

////////////////////////////////////////////////////
/// Structure for holding the changing variables of the walk
////////////////////////////////////////////////////
typedef struct _walkfactorsstruct {
   /// Size of step
   float stepSize;

   /// Amount of lift leg
   float legLift;

   /// True if want robot to stop walking
   bool stop;
} WalkFactors;

////////////////////////////////////////////////////
/// NewWalk Class
```

```
////////
///
/// Encapsulated inside a derieved SlaveControl class
//////////////////////////////////////////////////
class NewWalk : public SlaveControl
{
protected:
        ///////////////////////
        /// Activation of a stage
        ///////////////////////
        bool stageAct[NUM_STAGES];

        ///////////////////////
        /// Start time of a stage
        ///////////////////////
        float stageStart[NUM_STAGES];

        ///////////////////////
        /// Running time of a stage
        ///////////////////////
        float stageTime[NUM_STAGES];

        ///////////////////////
        /// Holds the current walk factor settings
        /// Only accessible by the class
        ///////////////////////
        WalkFactors currentWalkFactors;

        ///////////////////////
        /// Turning slave
        ///////////////////////
```

```
        NWTurn *turnSlave;

#ifdef SIMULATOR
        //////////////////////
        /// ZMP log file
        //////////////////////
        FILE *zmpLog;
        //////////////////////
        /// Torso Velocity
        //////////////////////
        float nwZMP[ZMP_AVG_DIV];
#endif

public:
        //////////////////////
        /// Holds the desired walk factor settings
        /// This is accessible by anyone
        //////////////////////
        WalkFactors desiredWalkFactors;

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

        //////////////////////
        /// Constructor (with pointer to Master Control)
        /// Use this constructor to instance a class since
        /// you need a pointer to the master control
        //////////////////////
```

```
        NewWalk (char* name, MasterControl *master);

        //////////////////////
        /// Deconstructor
        //////////////////////
        virtual ~NewWalk ();

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

        //////////////////////
        /// Setup the timings for the gait
        /// Parameter indicates if first cycles of gait or not
        //////////////////////
        void setupTiming (int timing);

#ifdef SIMULATOR
        //////////////////////
        /// Logs the velocity of the torso
        /// SIMULATOR ONLY
        //////////////////////
        void logTorsoVel ();

        //////////////////////
        /// Logs the Zero Moment Point (including a low-pass filter)
        /// SIMULATOR ONLY
        //////////////////////
```

```
        void logZMP ();
#endif

};

#endif // _NEWWALK_H_
```

12.3 NewWalkGA.h

```
/*************************************************************************
******
 * Copyright 2003, Gordon Wyeth

*************************************************************************
*****
 *     File: NewWalkGA.h
 *   Author: Tim Pike
 *  Project: Humanoid 2.0 Dynamechs 4.0
 *  Created: 22nd Augst 2003
 *  Summary: Slave to control the NewWalk gait algorithm in order to use GA program
 *           to tune the walk.

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

#ifndef _NEWWALKGA_H_
#define _NEWWALKGA_H_

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

#include <dmLink.hpp>
#include <dmArticulation.hpp>
#include <dmRigidBody.hpp>
#include <dmContactModel.hpp>

extern dmSystem *G_robot;
```

```
#define MAX_COUNT 240

//////////////////////////////////////////////////
/// NewWalkGA Class
////////
/// This class derives from NewWalk and it will call NewWalk's
/// controlFunction from its own, but it also adds the necessary
/// mechanisms to allow the GA stuff to work (this keeps the
/// GA stuff separate from the NewWalk algorithm.
//////////////////////////////////////////////////
class NewWalkGA : public NewWalk
{
protected:
        float comparsion[MAX_COUNT];
        int compareCount;

        float preZmpX;
        float fitnessx;
        float preZmpY;
        float fitnessy;

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

        ////////////////////
        /// Constructor (with pointer to Master Control)
        /// Use this constructor to instance a class since
```

```
        /// you need a pointer to the master control
        //////////////////////
        NewWalkGA (char* name, MasterControl *master);

        //////////////////////
        /// Deconstructor
        //////////////////////
        virtual ~NewWalkGA ();

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

#endif // _NEWWALKGA_H_
```

12.4 NWTurn.h

```
/*************************************************************************
******
 * Copyright 2003, Gordon Wyeth

*************************************************************************
*****
 *     File: NWTurn.h
 *   Author: Tim Pike
 *  Project: Humanoid 2.0 Dynamechs 4.0
 *  Created: 29th September 2003
 *  Summary: Turning algorithm.  Slave is designed to be run as a normal slave, but
 *           all timing control is done from another slave (ie. NewWalk)

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

#ifndef _NWTURN_H_
#define _NWTURN_H_

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

#define NWT_MAX_STAGES 1

#define NWT_TIMING1 1
#define NWT_TIMING2 2

//////////////////////////////////////////////////
/// NWTurn Class
```

```
////////
///
/// Turning algorithm
//////////////////////////////////////////////////
class NWTurn : public SlaveControl
{
protected:
        ///////////////////////
        /// Activation of a stage
        ///////////////////////
        bool stageAct[NWT_MAX_STAGES];

        ///////////////////////
        /// Start time of a stage
        ///////////////////////
        float stageStart[NWT_MAX_STAGES];

        ///////////////////////
        /// Running time of a stage
        ///////////////////////
        float stageTime[NWT_MAX_STAGES];

        ///////////////////////
        /// Angle to turn to
        ///////////////////////
        float destAngle;

        ///////////////////////
        /// Current angle turned to
        ///////////////////////
        float currentAngle;
```

```
        //////////////////////
        /// Current angle to turn
        //////////////////////
        float turningAngle;

        //////////////////////
        /// Period to do turning at
        //////////////////////
        float period;

        //////////////////////
        /// Speed to do entire turn at
        /// Speed is defined as value 0<->1
        //////////////////////
        float speed;

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

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

        //////////////////////
```

```
        /// Deconstructor
        //////////////////////
        virtual ~NWTurn ();

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

        //////////////////////
        /// Setup the timings for the turn
        /// Parameter indicates area to setup timing for
        //////////////////////
        void setupTiming (int timing);

        //////////////////////
        /// Setup the turn
        /// Parameters indictate angle, speed, period of movement etc.
        //////////////////////
        void setupTurn (float angle, float period, float speed);
};

#endif // _NWTURN_H_
```

12.5 GANewWalk.cpp

```
/*************************************************************************
******
 * Copyright 2003, Gordon Wyeth

*************************************************************************
*****
 *    File: gaNewWalk.h
 *  Author: Tim Pike
 * Project: Humanoid 2.0 Dynamechs 4.0
 * Created: 5th September 2003
 * Summary: Use the GALib library to find the best values for the variables of
 *          the NewWalk GuRoo walking algorithm.
 *
 * Notes: GALib, Copyright 1995-1996 Massachusetts Institute of Technology
 *        Program based on:
 *        ex9.C, mbwall 10apr95,
 *        Copyright 1995-1996 Massachusetts Institute of Technology
 *
 *        Also Modified by Jon Roberts (jmr@itee.uq.edu.au) for GAControl.

*************************************************************************
*****/
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <ga/ga.h>

#define VAR_FILE    "../Models/nwvars.cfg"
```

```
#define RESULT_FILE         "nwtmp.dat"

#define NUM_VARS 4

float     vars[NUM_VARS] = {0};

FILE *allScores;

float objective(GAGenome &);

/*
 * This function writes the file that the humanoid program reads
 */
int
write_vars()
{
        FILE    *fw;
        int     i;

        if ((fw = fopen(VAR_FILE, "w")) == NULL) {
                fprintf(stderr, "write_vars: cannot open file %s\n", VAR_FILE);
                return -1;
        }

  for (i=0; i<NUM_VARS; i++)
                //fprintf(fw, "#define %s %d\n", name[i], gain[i]);
    fprintf (fw, "%f\n", vars[i]);

        fclose(fw);

        return 0;
```

```
}

/*
 * This function reads the results file generated by the humanoid program
 */
float
read_result()
{
        FILE    *fr;
        float    result;

        if ((fr = fopen(RESULT_FILE, "r")) == NULL) {
                fprintf(stderr, "read_result: cannot open file %s\n", RESULT_FILE);
                return -1;
        }

        fscanf(fr, "%f", &result);

        fclose(fr);

        return result;
}

/*
 * This function performs a single cycle
 */
float
do_run()
{
        static char      string[BUFSIZ];
        float            result;
```

```
        write_vars();

    // Uncomment for unix
    //sprintf(string, "../Simulator/humanoid > %s", RESULT_FILE);

    // Win32
    sprintf (string, "simulator.exe > %s", RESULT_FILE);

        system(string);
        result = read_result();

        return result;
}

int
main(int argc, char **argv)
{
        unsigned int seed = 0;

        // Declare variables for the GA parameters and set them to some default values.

        int     i;
        int     popsize  = 150;
        int     ngen     = 75;
        float   pmut   = 0.0050;
        float   pcross = 0.6;

        // Create a phenotype for gains.

        GABin2DecPhenotype map;
```

```
        map.add(16, 20, 35);
map.add(16, 20, 40);
map.add(16, 10, 25);
map.add(16, 10, 25);

allScores = fopen ("scores.csv", "w");
fprintf (allScores, "scores, knee, ankle place, ankle move, hip move\n");

        // Create the template genome using the phenotype map we just made.
        GABin2DecGenome genome(map, objective);

        // Now create the GA using the genome and run it.  We'll use sigma
        // truncation scaling so that we can handle negative objective scores.
        GASimpleGA ga(genome);
ga.minimize();
        GASigmaTruncationScaling scaling;
        ga.populationSize(popsize);
        ga.nGenerations(ngen);
        ga.pMutation(pmut);
        ga.pCrossover(pcross);
        ga.scaling(scaling);
ga.selectScores(GAStatistics::AllScores);
        ga.scoreFilename("ganwrun.dat");
        ga.scoreFrequency(1);
        ga.flushFrequency(1);
        ga.evolve(seed);

        // Dump the results of the GA to the screen.
        genome = ga.statistics().bestIndividual();
        fprintf(stdout, "Final result:\n");
```

```
        for (i=0; i<NUM_VARS; i++)
                fprintf(stdout, "%d: %f\n", i, (float)genome.phenotype(i));

        cout << "best of generation data are in '" << ga.scoreFilename() << "'\n";

    // Print to a log file
    FILE *vars = fopen ("nwgavars.cfg", "w");
    for (i=0; i<NUM_VARS; i++)
    {
       fprintf (vars, "%f\n", (float)genome.phenotype(i));
    }
    fclose (vars);
    fclose (allScores);


        return 0;
}


float
objective(GAGenome & c)
{
        static int          count = 0;
        int       i;

        GABin2DecGenome & genome = (GABin2DecGenome &)c;

        float y;

        for (i=0; i<NUM_VARS; i++)
    {
```

```
                vars[i] = (float)genome.phenotype(i);
    }

        y = (float)(100 * do_run());

    fprintf (allScores, "%f, %f, %f, %f, %f\n", y, vars[0], vars[1], vars[2], vars[3]);

        count++;
        fprintf(stderr, "%04d: %f\n", count, y);
        return (y);
}
```