Humanoid Omni-directional Locomotion

Thesis B Report

Brock White

October 10, 2011

School of Computer Science & Engineering

University of New South Wales

Sydney 2052, Australia

Supervisors and Assessors

Bernhard Hengst

K17-401E / +61 2 9385 6993

Claude Sammut

K17-401J / +61 2 9385 6932

Contents

Chapter 1

Abstract

This thesis documents the changes made to the locomotion module throughout 2011 as well as the implementation details of the open challenge. These changes include improvements to the infrastructure of the walk engine, getup routines and the integration of kicks into the walk engine. The open challenge entry described received 3rd place in the demonstrations held at robocup and is a bipedal control policy that was learnt in simulation that was then transferred to the Nao [4].

Chapter 2

Acknowledgements

I would first like to thank my fellow team members David, Sean, Bel, Carl, Yongki, Jimmy, Yiming, Nana, Youssef and Vance for their support throughout the last year, and the awesome times we had in the lab. Being part of this team and competition together was one of my primary motivations for being involved in this project and you have all made this time unforgettable for me.

I would also like to thank Jayen, who while having no official involvement in robocup solved many problems that I encountered throughout the year and who listened to my thoughts and gave my ideas.

To my house mates, Luke, Raz, Carl, Dave and Amy, thank you for partying with me in and around my robocup commitments and for making this year so great.

To Claude and Morri, thank you for your advise and recommendations, especially when we were losing sight of competition goals.

To Manuel, it was great working with you on our joint project and partying with you at the UNSW Village.

Lastly, to Bernhard, thank you for everything, the many hours spent planning the open challenge, hours spent on the walk, kicks and for chairing the weekly meetings. Without you

I'm not sure what we would have done.

Chapter 3

Introduction

Robotic soccer offers a unique challenge for the development of bipedal locomotion. Walks developed for use at competitions such as robocup [5] must be capable of withstanding the constant jostling that is common when robots brush shoulders while pursuing a soccer ball. Stability under pressure is not enough however, as teams develop faster more agile walks that are able to beat their oponents to the ball and shoot within seconds.

To move the ball quickly, complex kick motions are also explored with some teams creating dynamic kick motions that are able to adjust to a moving ball [2] and others experimenting with small jab kicks that enable ball stealing maneuvers [8].

This thesis aims to address these issues of humanoid omni-directional locomotion within the context of robotic soccer. The SPL (Standard Platform League) competition at robocup 2011 will be used as a demonstration of this work. This competition consists of group play followed by a knock-out tournament as well as open challenges where teams may demonstrate research.

The walk engine [6] developed in 2010 by Hengst was competitive in the 2010 competition and will be used as a basis for this work. This thesis will aim to improve on the shortcomings of this engine. These shortcomings were primarily, wear and tear on gear joints caused by a large knee bend in the walk and a slow side step that reduced the omni-directional abilities

of the walk. In addition to this a new integrated kick engine will be described that allows for faster kicking motions such as the new dribble kicks.

While this work is an incremental improvement on the previous walk engine, a side project was also developed during this thesis that aimed to learn a gait. This engine was not ready for competitive play but was entered as the rUNSWift entry for the open challenge at competition for which we received 3rd. This bipedal control policy [4] used the ankle joints and altered the step size of the walk to balance the robot as it was pushed by an external source. This control policy is learnt in simulation and then transferred to the Nao.

This thesis will begin with the infrastructure changes necessary to make improvements to the walk engine. The bulk of these changes were in the refactoring of the 2010 walk engine into three well defined modules, the BodyModel, Planner and WalkEngine. We will then discuss the changes required to implement the integrated kick engine and the new conservative walk that aims to address the problems with the 2010 walk engine. Following this we will look at the implementation of the bipedal control policy for the Nao and evaluate its effectiveness.

The thesis will conclude with some suggested future work that may be interesting for the 2012 team to investigate.

Chapter 4

Background

4.1 Introduction

This sections aims to orientate the reader with the concepts and related work relevant to this thesis. We will begin with a look at bipedal walk basics, this will be used to familiarise the reader with the terminology used to describe walks. Following this we will describe and evaluate to 2010 rUNSWift walk which forms the basis of this thesis. Finally, we will briefly review a paper by Hengst et al. [4] as this is the basis of the open challenge.

4.2 Walk Basics

The term gait is used to describe the pattern of movement of the limbs of a robot as it moves across a solid substrate.

Bipedal gaits consist of a continuous cycle in which the robot lifts and then places one foot in front of the other. Before lifting a leg the weight of the robot's body must be transferred to the other leg. While this transfer is occurring both feet are in contact with the ground, this is said to be the gait's double support phase. This is the most stable part of the cycle as it has the most contact with the ground.

Following this the robot's weight is shifted to the side of the body in which the leading leg is situated. The leading leg is lifted and brought forward to take a step. While this leg is in the air we are in the single support phase. In this phase the leg that is on the ground is called the support leg.

Once this leg makes contact with the ground the cycle then continues with the new leading leg then becoming the support leg and so on. This can be seen in Figure 4.1.

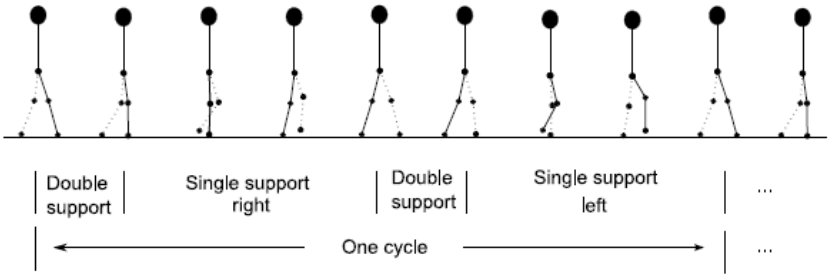

Figure 4.1: A bipedal walk cycle. Left leg is indicated by dotted line and right is solid line [13].

Walks that use sensory feedback to balance are called closed-loop walks, while walks that do not are open-loop walks. A popular technique for closing this loop is the zero moment point (ZMP).

The ZMP refers to a point on the ground where the total moments of the moving bodies is zero. A walk is considered balanced as long as the ZMP remains inside the contact area, i.e. the robot's feet. If the ZMP lies outside of the contact area the resultant moment will tip the robot over. The ZMP is also closely related to the centre of pressure (CoP).

The CoP is the point on a body where the total sums of the pressure field acts, causing a force and no moment about that point. When the body is dynamically balanced the ZMP and CoP coincide. When unbalanced the CoP lies at the edge of the foot and the ZMP does not exist.

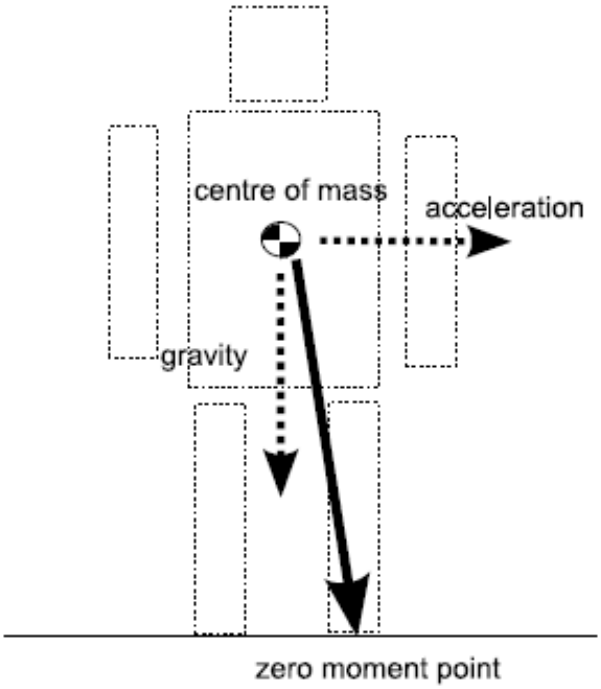

Figure 4.2: Diagram showing the relationship between centre of mass, gravity, acceleration and zero moment point.

4.3 2010 Walk Engine

Full details of the 2010 rUNSWift walk (FastWalk) may be found in the team report [6]. The purpose of this section of the report is to describe some of the salient features of this walk and how it performed at competition. It is important to note that this thesis aims to build on this engine and thus many of the methods used to balance the robot are still relevant.

FastWalk walks with a high frequency step, or patter (approximately 0.38 cycles per second). It has full omni-directional ability with an emphasis on forward speed. Of the walks used at competition it had the second fastest forward speed at approximately 22-24cm/s.

Due to time limitations and the high frequency step of FastWalk, its side step speed was slower then required for competition. This resulted in us having to switch to the walk supplied by Aldebaran whenever we needed to side step quickly at competition.

Stability control for FastWalk is designed by dividing the dynamics into the coronal and sagittal planes.

In the coronal plane stabilization was achieved by synchronising the onset of the leg lift motion with the switch in stance and support foot [6]. The zero-crossing of the measured CoP is used to achieve this. The CoP is negative when the robot is on its right foot and positive when on its left foot. The idea then being, as the CoP moves from positive to negative we know that the robot has just began switching its weight to the other foot. this information is then used to synchronise the leg lift as stated above.

It must be noted that at competition and in practice FastWalk rarely fell over in the coronal plane. This indicates that this method was quite successful at stabilising the robot in that plane.

In the sagittal plane stabilization was achieved through leaning of the torso. This was achieved via filtering sagittal CoP and accelerometer readings in the sagittal plane and using these to adjust the tilt of the torso. In addition to this the head was noted to be quite heavy (476.71 grams) [1] and for this reason a simple compensation was added such that if the head was leant backwards the torso was leant forwards slightly in response.

To kick in 2010 the robot was required to transition through a stable stand state and into a separate motion engine to perform the kick.

4.4 Open Challenge

For the open challenge we implemented a bipedal control policy on the Nao. This control policy is described in full detail in the article by Hengst et al. [4], however this section aims to give a brief overview of this work as this is the basis of the open challenge.

The main idea is that the bipedal control policy is learnt through simulation using reinforcement learning. The policy learnt maps from a simple pendulum model to actions that actuate the ankle and step placement. After learning the policy in simulation the policy may then be transferred to a real robot, in this case the Nao, and executed to balance the robot.

The flat-footed humanoid was modelled as an inverted pendulum (see Figure 4.3) with the pivot located at the centre of pressure. This pivot position was controlled by actuating the ankle joint to change the centre of pressure of the support foot. By moving this pivot position we alter the dynamics of the pendulum such that it will accelerate under gravity in the direction we desire.

The state s of the system is defined by four variables (x, $\dot{x}$, w, t), where x is the horizontal displacement from the centre of the support foot to the centre of mass, $\dot{x}$ is the horizontal velocity of the centre of mass, w is the horizontal displacement from the centre of mass to the centre of the swing foot, and t is the time-step from the start of each walk-cycle [4].

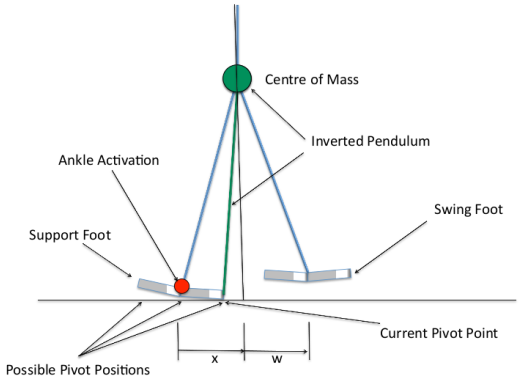

Figure 4.3: The inverted pendulum model of a flat-footed bipedal robot used for simulation and reinforcement learning [4]

The control actions consist of choosing the centre of pressure for the support foot relative to the foot centre and choosing a step change in the swing foot displacement.

This state, control actions and the inverted pendulum model were then modelled in a simulator by Hengst. Screen shots taken from this simulator may be seen in Figure 4.4. An optimal policy was then learnt in this simulator using Reinforcement learning.

The article also describes a similar experiment that was run in parallel with this thesis that aimed to implement this control policy on the Cycloid. To measure the performance of this controller an experiment was conducted to test its reaction upon stepping on a small

obstacle. This obstacle's height was varied so as to test the robustness of the controller at various step heights. The results from this experiment were positive and suggested that with the controller turned on, the cycloid was able to step on obstacles that were 70% higher then if the controller was turned off.

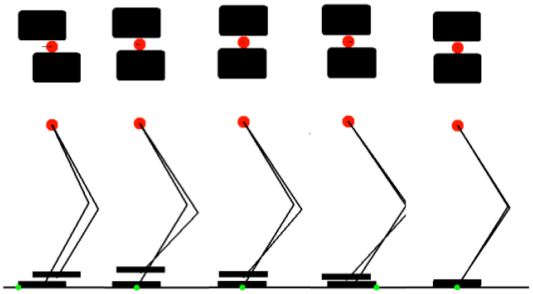

Figure 4.4: A sequence of screen shots taken from the animated simulator showing several frames in plan-view (top) and side-view (bottom) [4].

4.5 Conclusion

This section has oriented the reader with the terminology used in describing humanoid gaits and has also analysed the 2010 rUNSWift walk which will form the basis of this thesis. This thesis will aim to utilise the strengths of this walk engine and improve upon its weaknesses.

In addition to this we have also looked at the article by Hengst et al. [4] which is the theory behind the work done in this thesis on the open challenge.

Chapter 5

Infrastructure

5.1 Introduction

This section aims to document the infrastructure changes that occurred during this thesis. We will begin by describing the new motion interface, following this we will look at the design and layout of the new walk engine components. Lastly, the changes to the PosFile generator and the getup routines will be documented.

5.2 Motion Interface

The old motion interface required the user to set fields in three structs. One struct determined Head movement, one LEDs and the other Body movement. Another value was also set for actionType that is used to determine whether you are performing one of a number of actions including kicks, walks and getup routines.

In the new motion interface the parameters sent to the Body have changed. The parameters were previously forward, left, turn and power. If the actionType was set to WALK, the forward, left and turn parameters specified the desired velocity of the walk in those directions. If set to KICK, forward, left and turn were overloaded to specify the direction of the kick.

In addition to this, the power parameter was used to determine the power of the kick. For more detail see the rUNSWift 2010 report [6].

Under the new interface; forward, left and turn have now been altered to represent the distance the walk is required to cover to reach its destination. The motion planner then determines what velocity to use to best reach this destination.

For kicks; forward, left and turn are no longer used. Instead we now have a new kickDirection and foot parameter. KickDirection is used to determine the direction we are to kick and is specified in radians. Foot is used to determine which foot to use. So for example, kickDirection = 0 with foot=RIGHT is a forward kick using the right foot.

In addition to this there is now a parameter called isFast. This is used to switch between a slower and faster frequency walk and is covered in more detail in chapter 7.

5.3 Walk Engine Design

As our walk engine improves, we will always be aiming for more robust human like gaits. These gaits will almost certainly involve some degree of step planning and body modelling to allow for this planning. The 2010 walk engine utilised basic versions of both of these features and integrated these directly into the walk engine.

To allow our code to move in this direction it was decided that for clarity the walk should be partitioned into three distinct parts. The Planner, BodyModel and WalkEngine.

- **Planner** This component has the responsibility of interpreting action commands and converting these into step requests. In the future we aim for this to involve some form of planning such that we remain balanced. For now this planning simply ensures that we do not change velocity too rapidly. The controller learnt through reinforcement learning that is described in chapter 8 is built into this module.

- **BodyModel** As the name suggests this component aims to model the body. For now

this is just simple ZMP filtering for direct feedback balancing. In the future we are aiming for this to be a pendulum model. The Planner will use this BodyModel to make informed decisions about which steps to take.

- **WalkEngine** This component is responsible for taking step requests and using these to create joint angles. It is also responsible for performing kicks as we have integrated kicks in the walk engine this year.

5.3.1 Planner

The first version of the step planner was simply a direct extraction of the parameter scaling code from the 2010 walk. The idea is that we have input values for forward, left and turn. We then have internal real forward, left and turn values for the step planner. We then ensure that we interpolate slowly towards the required parameters. By interpolating slowly we then ensure that no sudden momentum changes occur and thus we usually remain balanced. The rate we interpolate determines the acceleration capabilities of the walk.

One of the larger changes to this module was that the ActionCommand interface now expects to be given a distance to a goal rather than velocities. For example, a forward value of 1000 means, move 1000mm forwards. The Planner module then decides what velocity the walk should move at to reach this destination optimally. The implementation for this was simple as a true Planner was not developed in this thesis. We simply divided the target distances given through behaviours by a constant. These constants were, 4.0 for the forward component, 3.0 for the left component and 2.0 for the turn.

To clarify, if behaviours request that we walk forward by 100mm, the Planner will set the target velocity to 100/4.0 or 25mm per step. Note, the walk does not immediately walk at this speed, instead the current walk step size is interpolated to this speed.

Another smaller change was to only allow the walk to accelerate when it is operating at a desired step frequency. This was to avoid situations in which the walk starts turning or side

stepping when the walk is first starting to ramp up (approximately 1 second). This was implemented by the following lines of code.

```
if (bodyModel.getRealStepFrequency() > .50 || walkEngine.
   getCurrentActionType() != ActionCommand::Body::WALK) {
    targetWalkPatter(0.0f, 0.0f, 0.0f);
} else {
    targetWalkPatter(targetForward, targetLeft, targetTurn);
}
```

As can be seen, if we are not trying to WALK or the measured step frequency of the walk is greater then 0.50 we interpolate the walk parameters to 0.0. Note, targetWalkPatter is based off a function written by Hengst in 2010 [6] that interpolates walk parameters. If we are in the walk and the frequency is less then or equal to the desired frequency of 0.50, we interpolate towards our actual target velocities. The measured step frequency rises above 0.50 when the robot is unstable. This can be through pushing against another robot, starting to rock as the walk takes its first step after standing or becoming unstable at high speeds. In all of these situations it is desirable to begin slowing down to allow the walk to stabilize.

One main disadvantage to this method was that the conservative walk developed, which is to be discussed in chapter 7, took a few seconds to stabilize as the walk started up. This was because it had no hip movement in the coronal plane to assist with the coronal rock. This meant that we took between 1-2 seconds to begin moving after actions such as kicks or dribble kicks.

5.3.2 Body Model

A basic body model already existed in the 2010 walk. This model was primarily a set of filters over the ZMP sensors and accelerometers. These filtered results were then used to provide feedback based balance.

These filters were abstracted out into a new component called the Body Model in this thesis. The plan was to replace the ZMP based balancing with a more complex pendulum based model. This was the focus of the reinforcement learnt controller discussed in chapter 8.

The BodyModel is essentially a class with getters for all the useful information the Planner and the Walk Engine need to know about the state of the robot.

Some of the more important getters are:

- **getFilHighZMPF** Used to access the high filtered ZMPF which was created by Hengst [6].
- **getFilLowZMPF** Used to access the low filtered ZMPF which was created by Hengst [6].
- **getFilAccX** Used to access the filtered accelerometer values in the X plane created by Hengst [6].
- **getIsBetweenSteps** Used to determine if both feet are on the ground.
- **getRealStepFrequency** Returns the step frequency of the walk. Calculated by averaging the period of the last 3 steps.
- **updateStepFrequency** Used by the Walk Engine to inform the BodyModel that we have just performed another step. This is then used to help calculate the real step frequency.
- **getFootZMP** Used to get the ZMP value for a particular foot.
- **getPressureL/getPressureR** Used to find the total pressure on the left or right foot. This is used to determine if that foot is on the ground or not within the walk engine.
- **getWalkCycle** Used to retrieve the walk cycle object.
- **getPendulumModel** Used to retrieve the pendulumModel. This is used in the open challenge chapter 8.

Because of the need to pass information between the planner, body model and walk engine; two new types were created. The first of these was the WalkCycle class. This class takes the current walk parameters (forward, left, turn, T, t) as input, and outputs the current position of the feet for those parameters. This is useful as it abstracts away the walk cycle code so that we may use it in multiple locations. For example, in the simulator developed by Yusmanthia [14] and in the body Model for the open challenge. The bodyModel requires the walk cycle object for it to calculate the ZMP based off the balancing mechanism developed by Hengst in the 2010 rUNSWift report [6]. The walk engine creates this object and uses *setCurrentWalkCycle()* to pass this to the Body Model.

For the work in the open challenge we modelled the state of the robot as an inverted pendulum. A PendulumModel class was created to track the state of this. This class simply tracks the x position relative to the support foot of the centre of mass of the pendulum and its velocity (dx). This will be discussed more in chapter 8.

5.3.3 Walk Engine

The walk engine is responsible for taking a walk request and converting this into a set of joint angles. A new type was created to allow the Planner to make requests of the Walk Engine. The WalkEngineRequest class was created to fill this role. This class consists of a series of parameters that one needs to set before calling the WalkEngine. The parameters are.

- **actionType** Used to specify if we are in KICK, WALK, DRIBBLE or SHUFFLE.
- **foot** Which kick to use when kicking.
- **kickDirection** What direction to kick in.
- **forwardL** Forward component of left foot.
- **forwardR** Forward component of right foot.

- **left** Left component of walk.
- **turn** Turn component of walk.
- **power** Power of kick.
- **bend** Knee bend of walk.
- **isStopping** Is the walk trying to stop.
- **isStopped** Is the walk already stopped.
- **frequency** Frequency of the walk.
- **hipOffset** Hip offset used for balance.
- **ankleOffset** Ankle offset used for balance within the open challenge.

Upon receiving a WalkEngineRequest, if a change of actionType is requested a special procedure is called that ensures the transition to that actionType happens at the correct time within the walk cycle. This procedure is discussed more in chapter 6. The parameters involved in these special kick requests are actionType, foot, kickDirection and power.

The forwardL, forwardR, left, turn and frequency parameters are used to determine the position of the feet within the walk cycle. These parameters are read directly from the WalkEngineRequest object every cycle.

The hipOffect and ankleOffset parameters are added directly to their corresponding joints. These parameters are read from the WalkEngineRequest object every cycle.

Together these parameters allow the Planner module to control the walk engine.

The core of the walk engine that takes target foot positions and generates the joint angles required to do this remains unchanged from 2010. There was no need to change this as the inverse kinematics developed by Hengst in 2010 [6] works as expected. However there have been a few changes to the walk engine in the layer above this.

Firstly, the kick integration required many changes to the walk engine and is described in full detail in chapter 6. In essence the kicks are executed within the typical walk cycle by altering the parameters that are fed to the lower level inverse kinematic code from last year.

A second change was made to the method by which support foots are switched. Previously, the support foot would switch as soon as t became greater then T/2.0 or t became greater then T. This is the natural place to switch support feet if the robot is stable. However, when unstable this will result in the robot stepping before both feet are on the ground. This was clearly visible by conducting a simple experiment in which I held the robot on its side with one foot touching the ground. In this case both feet would continue walking as if they were both in contact with the ground.

A simple change was made here that now requires a foot to be in contact with the ground before allowing the phase of the walk to switch. This fixed the aforementioned issue but added a new feature to the walk. Now, if a robot is stuck against another and can not put its foot to the ground after stepping, it will pause the walk cycle. It is hard to determine if this is good or bad because in some cases, trying to step while stuck against another robot may lead to our robot falling over. However, the obvious disadvantage is that we are unable to move and if we were able to step we may be able to wriggle free of being stuck. The solution to this problem may be to detect this state and perform some action to free us from being stuck such as to continue the walk cycle after some period of time has passed.

Because the walk cycle is now able to pause to perform kicks, or to wait for a foot to touch the ground before switching phases, some changes were required within the walk cycle generation code. Previously, an iterative method was used such that at each cycle of the walk, a small amount was added to an accumulating total for each of the values of, forwardR, forwardL, leftL, leftR, turnLR [6]. However, there was no concept of a paused walk cycle and so, if we tried to pause t these parameters would keep progressing. To fix this a new value was introduced called rdt (real dt). This was updated for each cycle of the walk. If the walk cycle is paused it is set to 0. If not rdt is set to dt. This value is then used to update all

of parameters listed above as well at t. Note, the code that was used to calculate forwardR, forwardL, leftL, leftR etc, has now been extracted out into the WalkCycle class discussed in subsection 5.3.2. The new walkCycle code does not calculate the parameters in an iterative way, instead the walk cycle parameters are calculated through one instance of t, forwardL, forwardR, left and turn.

5.4 Pos File Generator

The Pos file generator is responsible for generating canned actions from a file. This module was developed last year by Nguyen and Stuart [6]. The generator reads in Pos Files (joint position files), and from these files creates a sequence of joint interpolations. On each line in the file is the angle every joint is required to move to, the stiffness of the joints and following this is the time over which the interpolation should occur.

There were two primary changes to this module this year. These changes were made to accommodate the new goalie dive motions. The first change was to disallow looping of pos files. Previously, when a pos file was finished executing, it would loop again. For a goalie dive this caused obvious issues.

The second change was to allow one to specify the stiffness of all of the joints separately. Previously one could only specify one stiffness that all joints must use. However, to create a dive we needed to remove the stiffness from one arm as to soften our landing. To achieve this one can specify the stiffness in a pos file.

To do this we start a line with a $ symbol. Following this one specifies a stiffness for every joint in the same order as the joints are specified. The stiffness always refer to the set of joints on the previous line and this stiffness will come into effect once the interpolation has reached those particular joints. Note, the stiffness is not interpolated.

5.4.1 Get up routines

The upgrade for the Nao robot during 2011 provided them with longer forearms [1]. This extra length required the front get up routine to be tweaked slightly. The main issue was that during the final stages of the routine the robot's weight was too far forward with the extra weight in the forearms. This meant that the robot fell forward during getup.

The fix for this was to change the end of the getup routine. This change was made to the part of the routine in which the robot first begins standing up. Instead of standing up we added an extra state in the pos file that moves the robot's arms around behind its back Figure 5.1. In doing this we shift the robot's body weight backwards slightly. After this the robot brings its feet together and stands up. The final stages of this routine were altered by Teh at competition so as to be more reliable on the competition fields [10].

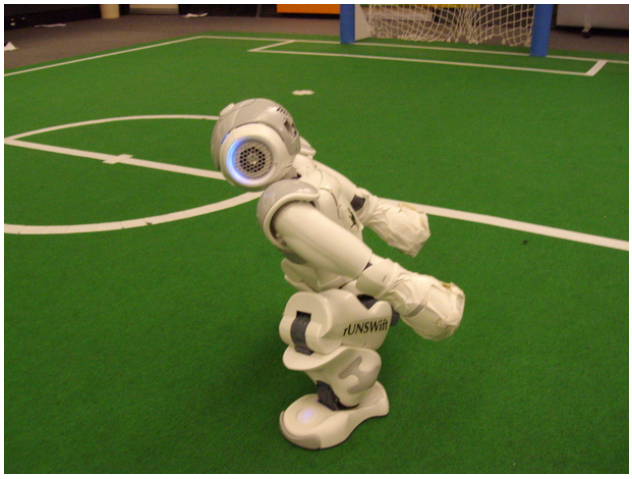

Figure 5.1: Additional state added to the front getup routine to move its body weight towards the back of its body.

An additional change was made to both getup routines. The 2010 getup routines assumed that the robot was always in a state with all joint angles set to approximately 0 before starting its interpolations. This caused problems when the robot fell on its arms. In those cases the robot would try to make impossible interpolations from these positions to the first position in the pos file which had the robot pulling its arms out to its side to begin the getup.

To fix this a new initial position was added to the getup routine that interpolates all joint angles to 0 before starting the getup. Now, if a robot falls on its arm it will move its arms down to the side of its body before starting the getup routine. This seems to be much more reliable and I believe this was a contributing factor to the reliability of the getup routines this year.

5.5 Evaluation

There were two primary benefits to the changes made to the motion interface. The first is that it was now clearer what parameters have to be set when one needs to perform a particular kick action. This clarity was one of the reasons the interface was changed in the first place.

The second advantage was that users no longer need to determine what velocity the walk needs to be set at in order to walk to a destination. Now that the motion interface accepts a destination the users are able to simply pass a destination through to the Planner and let the Planner determine the appropriate velocity to move at. Note, this means that the user does not have control over what velocity the walk has to move at when walking to a destination. This could be a problem if one wished to move to a destination slowly.

To solve this problem there was a variable in the ActionCommand interface called speed. We never ended up using this variable however as in all cases we used the maximum speed of the walk to move to a destination. This was primarily because we had the fast walk for when we needed to move quickly and we had the conservative walk for all other cases. These two walks are discussed more in chapter 7.

The redesign of the walk engine code made it much easier to implement the changes required for the open challenge. This was primarily because these changes were made in the Body-Model and Planner modules which are much simpler to work with than the WalkEngine module. Code that is easier to work with is one of the goals of good design so I believe this

refactoring of the walk engine was worth while.

The getup routines developed in 2010 worked consistently but occasionally had trouble with the front getup routine and when the hands got stuck underneath the body as described in subsection 5.4.1. Both of these problems were fixed during this thesis and this was noticeable at competition as the robot was able to getup always unless it was being pushed by another robot. After these changes the front getup routine now takes 5.8 seconds to stand up and the back get up routine takes 5.5 seconds.

5.6 Conclusion

This section has detailed the changes made to the locomotion infastructure during this thesis. This has ranged from the changes to the motion interface, the refactoring of the walk engine and the changes to the getup routines. Due to the increased flexibility created, the changes described in this chapter have helped to make the rest of this thesis easier to implement.

Chapter 6

Kicks

6.1 Introduction

In robocup it is common for teams to implement separate engines for their walk and their kicks. To transition between these engines teams often passed through a stable stand state to ensure the kick does not cause the robot to fall over. This method was used by rUNSWift in 2010 and B-Human [3].

This method has the obvious disadvantage that the robot must come to a complete stop before kicking thus delaying the kick. Alternatively, the kick engine may be incorporated into the walk engine giving complete control over kick transitions to the walk engine. This section will describe how this kick engine was developed in addition to the various kick types created. Three different kicks types will be discussed.

- **Standard kicks** Similar to those developed in 2010
- **Dribble kicks** Quick small kicks with minimal foot movement. To be used to beat an opponent when at the ball.
- **Shuffle kicks** Kicks that aim to walk while moving the ball to left or right. (Not used at competition)

6.2 Method

To integrate a kick engine into our walk engine it was necessary to determine; at what point in the walk cycle should a kick start, how do we shift the robots COM, what kick motion do we use and how do we transition out of the kick.

While there are similarities between each of the kicks developed, each kick developed has approached this problem in a different way.

6.2.1 Standard Kick

To manage the state of the standard kick, the following variables are used;

- **currentActionType** used internally by the walk engine to determine if the walk engine is currently, kicking or walking.
- **isKicking** used to determine when we are in the process of kicking
- **hasKicked** set to true after we have finished kicking
- **walkEngineRequest** used to store an incomming request from the walk planner at the start of a kick. This is saved so we know what type of kick was requested and is used to avoid future kick requests being confused with the current kick request.
- **T** period of the walk
- **t** our current position within the walk cycle
- **coronalAmplitude** the amplitude of our coronal rock
- **lastKickTime** this is set when we finish a kick
- **footForwardPos** forward position of the foot within a kick
- **footLeftPos** left position of the foot within a kick

- **footAnkleTilt** tilt of the ankle (up/down) within a kick
- **stepHeight** used to increase step height while in kick

Upon receiving a kick request we immediately save this request into the walkEngineRequest variable and set the currentActionType to KICK. After this the walk continues as normal until t = 0 for a left foot kick or t = T/2.0 for a right foot kick. At this point we set isKicking to true, T to 4.5 and hasKicked to false and coronalAmplitude to 25 degrees.

Now that we have switched to kick mode the next step is to transfer weight onto the support foot. This movement to the support foot is executed through the normal walk cycle by executing a walk cycle with different parameters. These parameters are T = 4.5 and a coronalAmplitude of 25 degrees. Once the weight has shifted to the support foot the walk cycle is paused so that we stay balanced on the support foot.

At this stage we execute the particular kick sequence we need. There are two types of kick motions implemented. Side kicks and forward kicks. While executing the kick only the non supporting foot is moved.

After executing the kick we set isKicking to false and then allow the walk cycle to continue from its paused state until we reach the double support phase. At the point hasKicked is set to true, currentActionType is set to ActionCommand::Body::WALK, t is set to 0, T is set to the normal step frequency of the walk and the coronalAmplitude is set to 0.

At this point we have finished the kick.

There are two specific kick motions executed by this kick type. These are described in the following two sections.

6.2.1.1 Forward Kick

By adjusting the stepHeight of the walk engine to be 15mm we allow the walk engine to naturally hold the foot at an optimal height to create our kick motion.

The kick motion itself consists of a 3 part function. The drawback, kick and return foot to neutral state.

All of these functions use the interpolate smooth method discussed in Appendix B.

The drawback phase lasts 0.4 seconds. In this period the foot's forward position is interpolated to -70mm and the leg's kneePitch is interpolated to 20 degrees. Both the foot's forward position and the kneePitch are assumed to be approximately 0 at the start of any kick motion.

The kick phase has a variable period. For maximum power we set the kick phase to have a period of 0.2 seconds and for minimum power we set it to 0.5 seconds. The results section looks at the range of kicks this produces.

The return foot to neutral state lasts for 0.6 seconds. This interpolates both kneePitch and forward foot position back to 0.

6.2.1.2 Side Kick

In a similar fashion to the forward kick, the stepHeight of the walk engine is set to 15mm during the side kick. The side kick motion consists of 3 parts. The drawback, kick and return foot to neutral state. The side kick also uses the smooth interpolation function. For convenience, this section will describe the left side kick. The right side kick may be deduced through symmetry.

The drawback phase is 1.0 seconds. During this phase the following interpolations are made. Forward foot position to 100mm, left foot position to 200mm, kneePitch to 30 degrees. Each of these values are assumed to be 0 at the start of the kick.

The kick phase is variable with the maximum being 0.7 seconds and the minimum being 0.1 seconds. During this phase the following interpolations are made. Foot left to -20mm, ankleRoll to 0 and kneePitch to 0. Note, foot left does not actually reach -20mm, instead it stops at 0mm. -20 is used to increase the velocity of the foot as it passes through the point

at which the ball would lie when being kicked. This increases the power of the kick.

The return to neutral state lasts 0.6 seconds and interpolates all values back to 0.

6.2.2 Dribble Kick

The dribble kick was inspired by the B-Human dribble kick seen throughout the German open [8]. The purpose of this kick is too move the ball a small distance with a quick kick motion. The quicker kick motion is achieved by not transferring the robot's weight completely onto the support foot. Instead we just increase the period of the walk throughout the kick and execute the kick motion during this period of time.

To manage the state of the Dribble kick the following state variables are used.

- **currentActionType** used internally by the walk engine to determine if the walk engine is currently, kicking or walking.
- **hasDribbled** set to true after we have finished dribbling
- **walkEngineRequest** used to store an incoming request from the walk planner at the start of a kick. This is saved so we know what type of kick was requested and is used to avoid future kick requests being confused with the current kick request.
- **T** period of the walk
- **t** our current position within the walk cycle
- **coronalAmplitude** the amplitude of our coronal rock
- **lastDribbleTime** this is set when we finish a kick
- **footForwardPos** forward position of the foot within a kick
- **footLeftPos** left position of the foot within a kick
- **stepHeight** used to increase step height while in kick

The dribble kick is activated in a similar fashion to the standard kick. Once the request is received the request is saved into the walkEngineRequest variable, coronalRockAmplitude is set to 20 degrees, T is set to 1.4 seconds, hasDribbled to false and the currentActionType is set to ActionCommand::Body::DRIBBLE.

The walk cycle then continues as normal. Within the next half walk cycle the kick motion is executed.

After half a walk cycle hasDribbled is set to true, T is set stepFrequency, t is set to 0, coronalAmplitude is set to 0 and currentActionType is set to ActionCommand::Body::WALK.

Similar to the standard kick, their are two types of dribble kicks. The following sections describe the functions used to produce the forward and side kicks.

6.2.2.1 Forward Kick

Their are 2 parts to the forward kick function. The kick and the draw back. The kick phase lasts 0.7 seconds. During this the forward component of the foot is interpolated to 70mm.

The second part of the kick lasts for 0.7 seconds and simply returns the foot to its neutral position at 0.

The kick length of 70mm was tuned to ensure the kick was stable.

6.2.2.2 Side Kick

The side kick consists of 3 parts. The first lasts for 0.7 seconds. During this phase the forward component is interpolated to 10mm and the left component is interpolated to 110mm.

The second phase is the kick phase. This lasts for 0.42 seconds. During this phase the forward position of the foot is interpolated from 10mm to 70mm and the foot left position is interpolated from 110mm to 0mm. Note that the foot's forward position is moved forward thus kicking the ball slightly forward. This is intentional so that the dribble kick will kick the ball at approximately 60 degrees rather then a typical 90 degree side kick. This aims to

kick the ball behind an opponent and out of their field of view.

The third phase is the return to neutral part. This interpolates the forward foot parameter to 0mm.

Note, this describes both the left and right side kick, the correct foot position in the side direction may be deduced.

6.2.3 Shuffle Kick

This kick aimed to move the ball either to the left or right. This was not achieved through a conventional kick motion. That is, if we wish to kick to the left we first move our right foot forward by 60mm and our left foot backwards by 60mm. Following this we side step slowly while the ball is between our feet.

The effect of this is to guide the ball to the left while keeping it between our feet. It also allows us to follow the ball and not have to chase after it afterwards.

Upon receiving a shuffle kick request a target forward value is set for both the right and left feet. For a left kick the right foot is given a target value of 60mm and the left foot is given a target value of -60mm. The right kick has these values reversed.

The two feet are then interpolated to these values over 1 second using the interpolate smooth function. During this we never leave or pause the normal walk cycle.

Now we are in the shuffle kick. To shuffle the ball across we now need to side step. The first iteration of the shuffle kick automatically performed this side stepping. This was later changed to allow the user writing the behaviour to determine what amount of left, forward and turn the walk should perform while in the shuffle kick. The reason for this was that the behaviour could determine the best way to stay lined up with the ball while shuffling it to the side thus keeping the ball under more control.

These forward, left and turn values issued by the behaviour are sent through the normal walk engine and added to the already offset values for the shuffle kick. Once a request is made to

leave the shuffle kick the offsets of 60mm on each feet are interpolated back to 0mm over 1 second. Following this the currentActionType is set to ActionCommand::Body::WALK.

Note, this kick requires less time to enter and leave than the other types of kicks as it never actually interrupts the normal walk cycle.

6.3 Results

6.3.1 Standard Kick

To quantitatively evaluate the Standard kicks an experiment was set up to measure the min and max range of the various kicks. For each kick, five balls are kicked from the middle of the goalie penalty box. A photo was then taken to qualitatively examine the grouping of the balls, i.e. is the kick consistently kicking in the same area of the field. Measured from middle of ball to the middle of the ball where kicked. Following this photo, the distance to the max distance ball and the distance to the min distance ball is measured.

This experiment was repeated for both forward and side kicks, on both feet and for power 0, 0.5 and 1.0.

The table in subsection 6.3.1 summarises the minimum and maximum distance kicked by each of the standard kicks at the power levels 0.0, 0.5 and 1.0.

	0.0	0.5	1.0
Forward (Left foot)	0.98m - 1.31m	2.20m - 2.71m	5.10m - 6.50m
Forward (Right foot)	0.93m - 1.24m	2.54m - 2.89m	2.54m - 2.91m
Side (Left foot)	0.56m - 0.81m	1.53m - 1.85m	1.14m - 1.78mm
Side (Right foot)	0.51m - 0.68m	1.40m - 1.96m	1.46m - 2.16mm

Figure 6.1 to Figure 6.12 show the 5 balls after each run of the experiment.

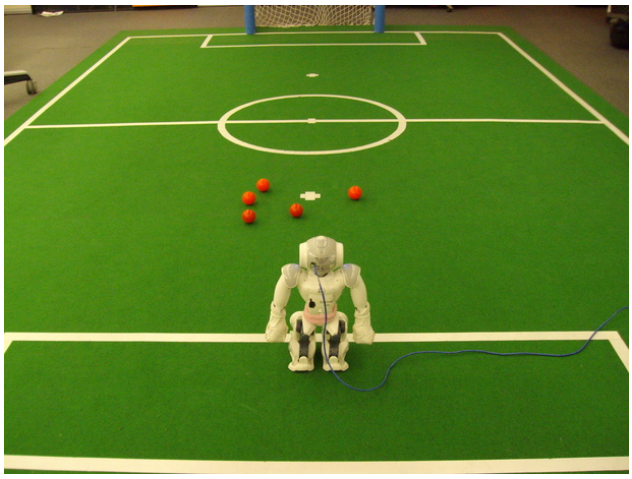

Figure 6.1: 5 Balls - Forward Left kick - 0.0 Power

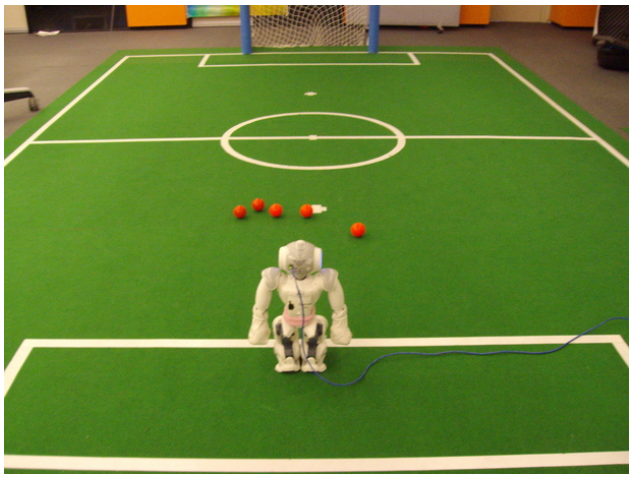

Figure 6.2: 5 Balls - Forward Right kick - 0.0 Power

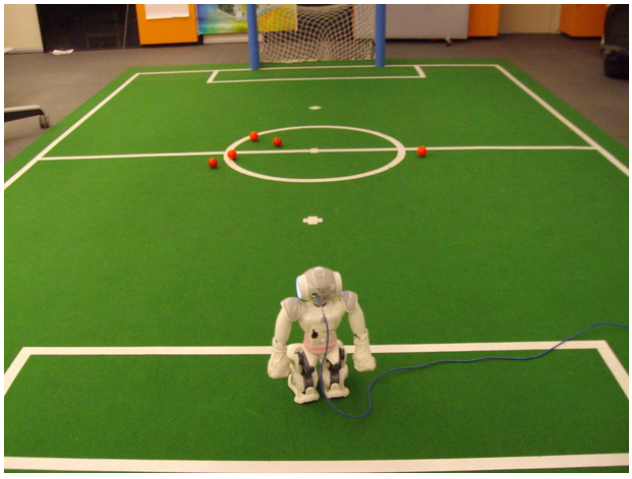

Figure 6.3: 5 Balls - Forward Left kick - 0.5 Power

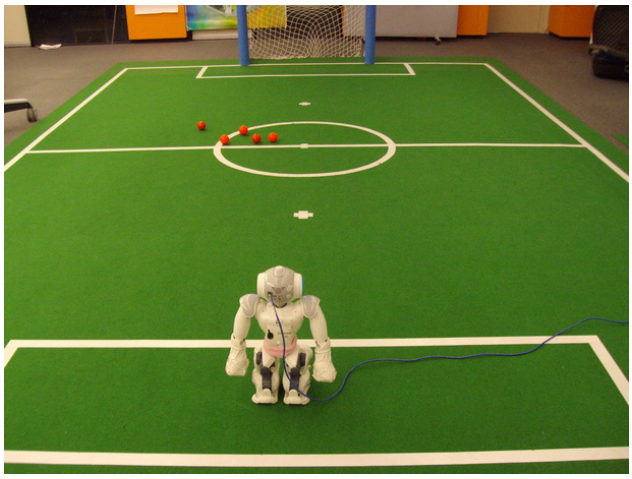

Figure 6.4: 5 Balls - Forward Right kick - 0.5 Power

Figure 6.5: 5 Balls - Forward Left kick - 1.0 Power

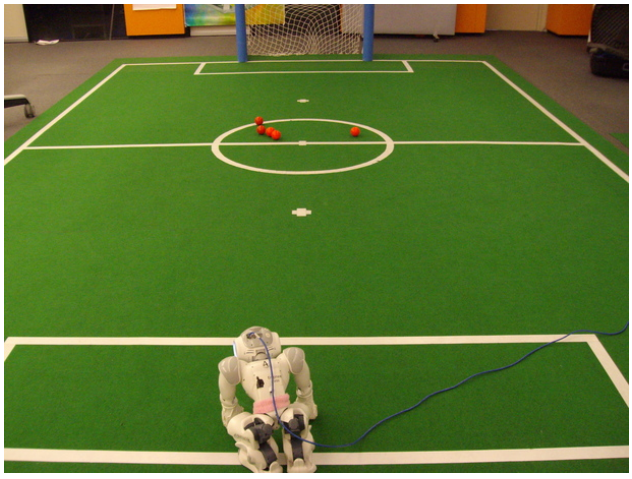

Figure 6.6: 5 Balls - Forward Right kick - 1.0 Power

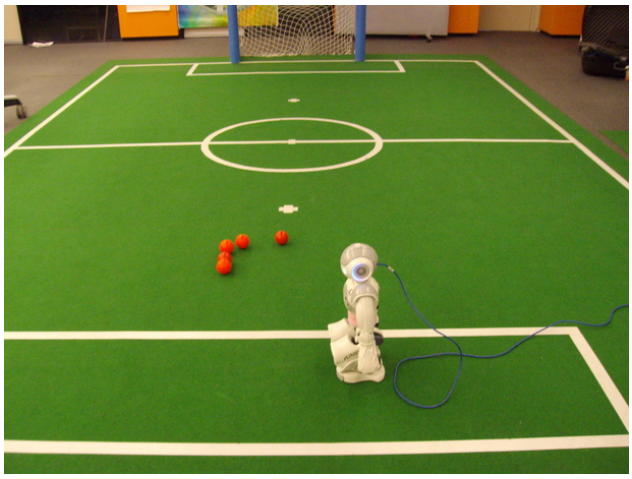

Figure 6.7: 5 Balls - Side Left kick - 0.0 Power

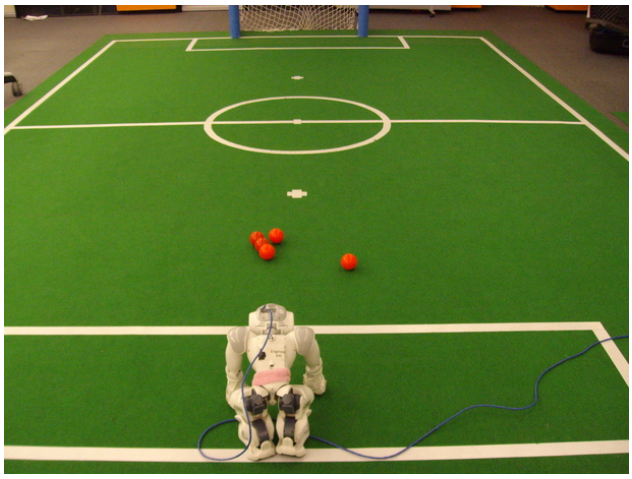

Figure 6.8: 5 Balls - Side Right kick - 0.0 Power

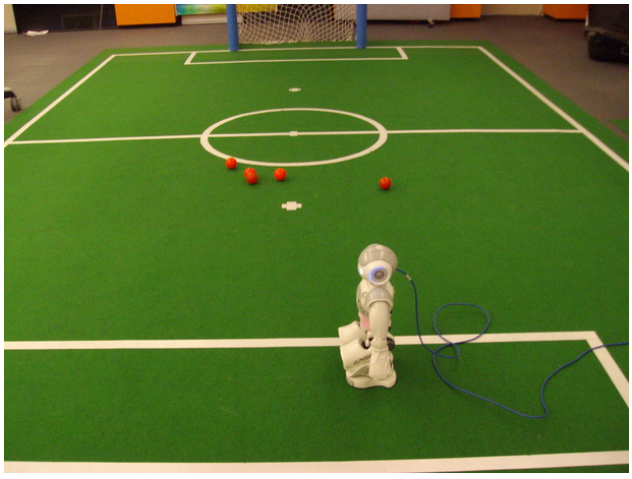

Figure 6.9: 5 Balls - Side Left kick - 0.5 Power

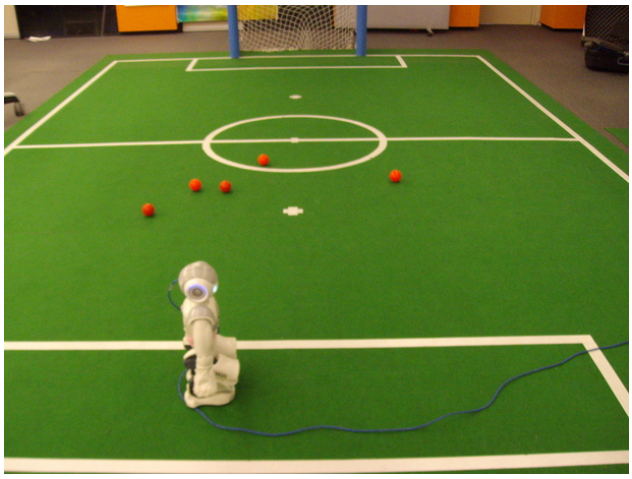

Figure 6.10: 5 Balls - Side Right kick - 0.5 Power

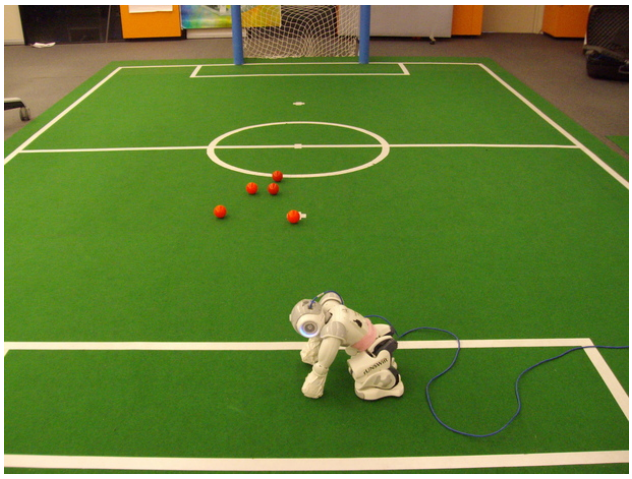

Figure 6.11: 5 Balls - Side Left kick - 1.0 Power

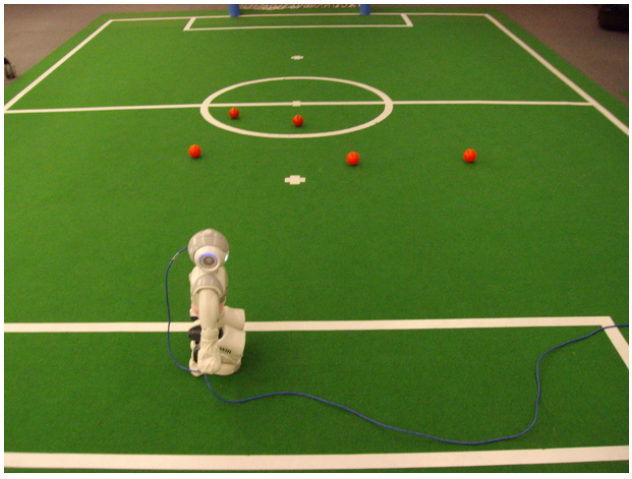

Figure 6.12: 5 Balls - Side Right kick - 1.0 Power

6.3.1.1 Evaluation

The first thing that immediately jumps out from these results is the decrease in range as we increase the power for the side kick on the left foot. This is most likely caused by trying to interpolate the joints too quickly during the kick. The motors are only able to move at a particular max speed, if we try to move them any faster they will simply lag behind their intended target and thus not actually move any faster. This was a mistake in the side kick's power interpolating function. It is likely that on a particular robot that was used to determine the power function for side kicks, the particular set of motors for that robot were able to show increased power if set to 1.0. However, this should have been tested more.

Another reason this issue may not have been fully explored was that side kick's were only used at full power in behaviours. This was because we had dribble side kicks if we required a softer shot.

The second anomaly is that the right side kick is 1/2.0 the power of the left side kick. For power's of 0.0 and 0.5 it is comparable, however at full power it kicks much less. I think, again this has to do with the motor not being able to keep up with the speed I am requesting in the interpolation. However, the left foot is able to keep up. This implies that there is some asymmetry in the robot that causes some motor in the left leg to be more responsive (Note, this occurs on all robots), or there is some error in the code. During my thesis I checked both functions for each leg as they were passed to the motors for the left and right kick. They appeared to be identical so I am unsure why this occurred.

To overcome this the left foot kick was used for long distance shots at competition.

6.3.2 Dribble Kick

The Dribble kick was tested in a similar way to the Standard kicks except that it was only tested under power of 1.0. This was because the Dribble kick was only ever intended to be used with one power setting.

The experiment conducted was to kick 5 balls with each kick type. The minimum and maximum distance to the set of balls kicked was measured and a photo was also taken to document the kick's grouping.

Kick Type	Distance Kicked
Forward (Left foot)	.59m - .91m
Forward (Right foot)	.44m - .70m
Side (Left foot)	.49m - .75m
Side (Right foot)	.65m - 1.29m

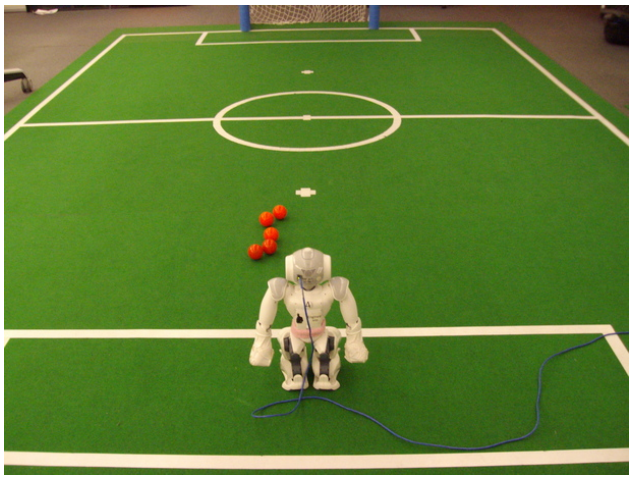

Figure 6.13: 5 Balls - Forward Left dribble - 1.0 Power

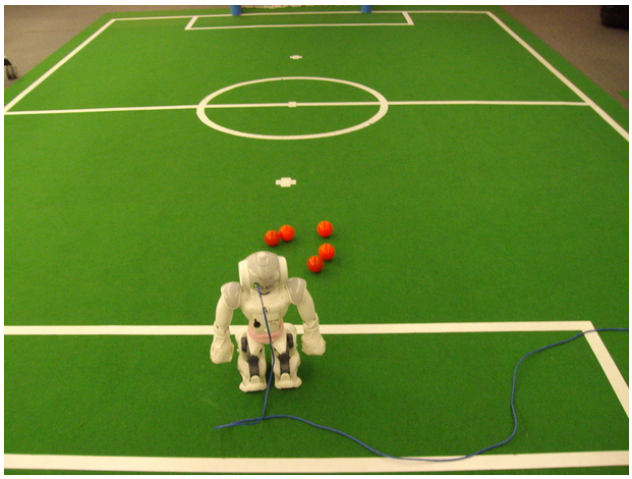

Figure 6.14: 5 Balls - Forward Right dribble - 1.0 Power

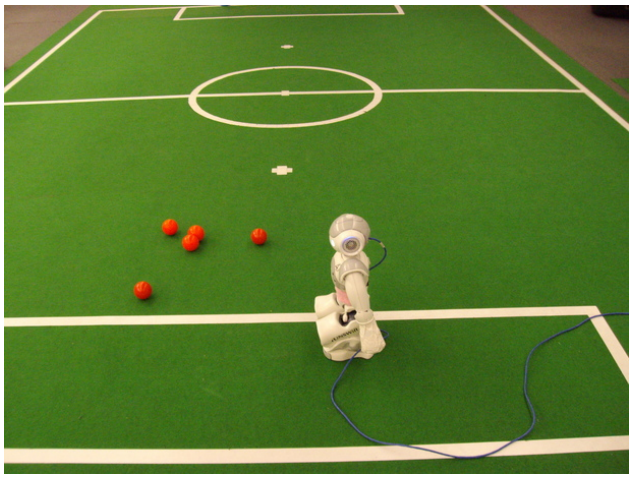

Figure 6.15: 5 Balls - Side Left dribble - 1.0 Power

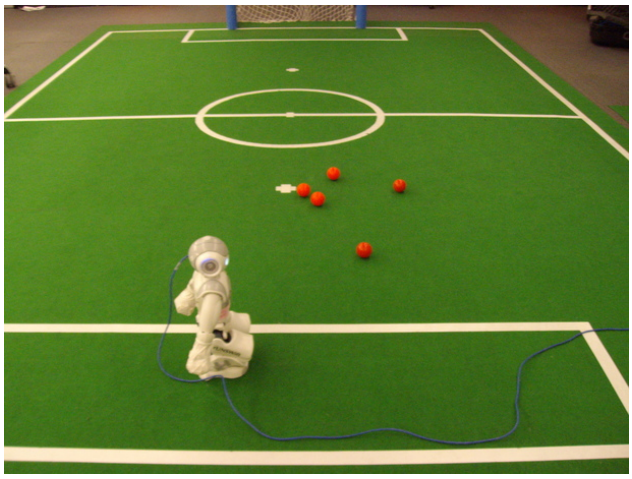

Figure 6.16: 5 Balls - Side Right dribble - 1.0 Power

6.3.2.1 Evaluation

These kicks were quite consistent as can be seen from the data. For these kicks to be effective we only require a small distance large enough to push the ball out of reach of an opponent robot that is also contesting the ball. The side kick seemed to kick further with the right foot than the left. After repeating the experiment for the right foot twice I was unable to determine why this was the case. This is perhaps a similar issue to that of the standard kicks discussed above.

6.3.3 Shuffle Kick

The implementation of the shuffle kick was never fully completed as the dribble kick filled a similar role within our behaviours. Test runs with the shuffle kick against a static opponent showed that it was able to move the ball to the left or right as desired, however was unstable while doing so. Because of this instability the robot would often jerk and this would cause the ball to be kicked further then desired. This instability was hard to remove and this is partly why we settled on using the dribble kicks at competition.

6.4 Conclusion

This chapter has described and evaluated the three types of kicks developed during this thesis. The standard forward kick developed was quite solid and a running joke in the team was that the goalie is likely to score (from our own goalie box). However, due to lack of time the side kick developed was not as powerful as we wanted and caused some shots to fall short during competition. If given more time the side kick could definitely be improved.

The dribble kicks were effective at dominating one on one situations at competition. We were often able to kick the ball before an opponent could line up an effective shot. Most notably, this occurred many times during one of our practice games in which we scored 1-1 against B-Human.

As will be discussed in section 9.1 I believe the shuffle kick and variants on this type of motion will be effective in the future.

Chapter 7

Conservative walk

7.1 Introduction

Two primary goals for this thesis were to increase the side step speed of the walk and to reduce stress on the robot caused by a large knee bend in Fast walk. A walk with a reduced knee bend and larger side step capabilities was created during this thesis to fill this role. This walk was not capable of high speeds however, so it was still necessary to transition between this new conservative walk and another walk with similar parameters to fast walk.

This section will describe this new walk and the transitions in and out of it. We will then look at some results and evaluate this walk and its transitions.

7.2 Method

The conservative walk is the product of countless hours of parameter tuning while I was becoming familiar with the walk engine from 2010. While many different types of walks were tested, this walk was eventually chosen to fill the "conservative" role within our soccer game play.

The method for producing this walk is quite simple. We alter the step frequency of the walk

to 0.50 and adjust the knee bend to 15 degrees. In addition to this it was required to change the step height of the walk to produce a natural coronal rock. The reason for this is that the step height affects how hard the foot pushes off the ground and thus causes varying amounts of coronal rock.

For the conservative walk the stepHeight value within the walkEngine was set to a value of 7.0f.

With the conservative walk completed we now had two walks within the one engine. The difference between the two walks were.

	Fast Walk	Conservative Walk
step period	0.5 seconds	0.38 seconds
knee bend	15 degrees	30 degrees
step function	7.0	6.0f+ABS(forwardL+forwardR)/2.0f*2.0/100.0f

When speed is required we would like to transition into the Fast walk parameters, otherwise we would like to transition into the Conservative walk. In addition, all of the kicks require us to be in the Conservative walk.

A new Boolean variable was added to the ActionCommand::Body struct to allow behaviours to choose which walk they wanted to execute. This variable was isFast.

Internally, the Planner module decides which of the walks is to be executed and how to interpolate between these walks. The Planner module has its own internal isFastMode variable to determine when it is acceptable to be in fastMode. The following code shows how this decision is made:

```
if (request->body.isFast && bodyModel.getRealStepFrequency()
   <= .50 && targetForward > 50 && !bodyModel.getIsStopped()
   && walkEngine.getCurrentActionType() == ActionCommand::
   Body::WALK && ABS(turn) < DEG2RAD(5) && forwardL > 40 &&
   forwardR > 40) {
```

```
        isFastMode = true;
    } else if((request->body.isFast == false) || walkEngine.
        getCurrentActionType() != ActionCommand::Body::WALK ) {
        isFastMode = false;
    }
```

The idea is that if isFast is set in the ActionCommand::Body struct and the walk is stable (real step frequency is <= .50), we are moving forward faster than step size 40mm, our turn parameter is less then 5 degrees and we are currently in the Walk; we set isFastMode to true. The basic idea behind these conditions is that we only want to go into fast mode if the behaviours want to and the walk is currently stable and already moving forward at a reasonable speed. These conditions being true is more likely to lead to a smooth transition.

We then set isFastMode to false if the behaviours request this to happen or we leave the walk state.

After the Planner makes this choice it must then choose how to interpolate the parameters and feed these parameters into the Walk Engine module. A simple constant learning rate interpolation was chosen for this. The code for this is as follows:

```
  if (isFastMode) {
      walkEngineRequest.T += (0.38 - walkEngineRequest.T) *
         .02;
      walkEngineRequest.bend += (30 - walkEngineRequest.bend) *
          .02;
   } else {
      walkEngineRequest.T += (0.5 - walkEngineRequest.T) * .02;
      walkEngineRequest.bend += (15 - walkEngineRequest.bend) *
          .02;
   }
```

This interpolation causes the robot to switch modes over a period of approximately 1.5 seconds.

7.3 Results and Evaluation

The conservative walk was capable of walking at a forward speed of approximately 16cm/s, side step speed of 8cm/s and turn speed of 40 degrees per second. This, coupled with its low wear and tear on the joints as a result of its reduced knee bend made it ideal for all roles in robocup that do not require speed. For example, the supporter and goalie do not need to walk quickly to their goals, instead it is more important to preserve the robots.

One of the primary disadvantages of the conservative walk was that when the walk first started to take steps after standing it took a few seconds to stabilize. This was because at a step frequency of 0.50 some degree of hip rocking is required to create a smooth walk motion. During this thesis we tried to find this hip rock but were unsuccessful. Instead we settled on a no hip rock and thus the walk is unable to take those initial steps smoothly.

This has the obvious impact on competition play in that after standing or kicking, the walk takes between 1-2 seconds to recover and begin walking again. This lag may give the opponent an edge in getting to the ball first.

The second issue to evaluate is the transition between two different gaits. While this is reminiscent of the old transition between the Aldeberan walk and Fast walk from 2010 it is much faster because we stay within the same walk engine and do not need to pass through the Stand state. Because of this the transition does not actually cause us to waste any time while moving about the field. While we chose to keep the parameters in Fast walk that were used in 2010, it was also possible to ramp up these parameters such that Fast walk caused more stress on the robot but was faster. This may have been possible because we only transitioned to this faster walk when we really needed to, thus reducing the overall stress on the robot.

This is similar to how a human uses different gaits for walking and sprinting and may be the direction robotic walks move in the future.

7.4 Conclusion

This chapter has documented the conservative walk and its transition from fast walk. The ability for this walk to avoid the overheating of leg joints was beneficial and the lack of overheating was noticeable throughout development during 2011. However, the slow start-up phase of the walk was undesirable for competitive soccer play and caused us to react slowly after kicking. I believe that future work should look into modelling the coronal plane so that this slow start-up phase may be avoided.

Chapter 8

Open Challenge

8.1 Introduction

This section aims to document the implementation of the bipedal control policy described in an article by Hengst et al. [4]. The changes required to implement this primarily occurred within the Planner and BodyModel modules. This control policy was learnt in simulation and uses a combination of ankle movement and foot placement to balance the robot. At robocup 2011, this policy was run as the rUNSWift entry in the open challenge and placed 3rd out of 20 teams.

8.2 Method

8.2.1 Body Model

The implementation of the bipedal control policy on the Nao required the BodyModel to calculate the pendulum model. A class was made to encompass this pendulum model. This class consisted of two fields, x and dx. It is then the BodyModel's responsibility to fill these fields with the appropriate information.

The pendulum.x and pendulum.dx values are filtered through a simple constant gain Kalman filter.

The observation update (see figure 8.2.1) uses a combinations of kinematic information and an on-board IMU (inertial measurement unit) to calculate pendulum.x. Pendulum.dx is then calculated through the difference of pendulum.x between the current frame and the last frame.

Note, pendulum.x jumps by a large amount as the phase of the walk changes because pendulum.x always refers to support foot [6]. Because of this we do not update pendulum.dx during a change of support foot, this is why we check the walkCycle's phase before updating pendulum.dx.

It must also be noted that pendulum.x is corrected whenever we have a change in support foot. The code for this may be seen in figure 8.2.1.

```
float alpha = 0.2;
float beta = 0.25;

pendulumModel.x = (1 - beta) * pendulumModel.x + (beta) *
   supportFootPosition;
pendulumModel.x = (1 - beta) * pendulumModel.x + (beta) *
   imuX;
if (pendulumModel.walkCycle.leftPhase == currentWalkCycle.
   leftPhase) {
   pendulumModel.dx = (1 - alpha) * pendulumModel.dx + (
      alpha) * (pendulumModel.x - lastPendulumModel.x) *
      100;
}


if (pendulumModel.walkCycle.leftPhase != currentWalkCycle.
```

```
leftPhase) {
    if (!currentWalkCycle.leftPhase) {
        // switching to left support phase
        pendulumModel.x = -pendulumModel.x/1.2;// + forwardR -
            forwardL;
    } else {
        pendulumModel.x = -pendulumModel.x/1.2;// + forwardL -
            forwardR;
    }

}
```

Now all we need to complete the observation update, is the actual inputs, supportFootPosition and imuX.

The supportFootPosition value is determined solely from the Kinematics module [6]. This is achieved by taking the origin of the support foot and creating a kinematic chain from that support foot into the coordinate system of the body. We then take the origin of the support foot into this space. The x component of the result of this calculation will then give us the x offset of the support foot from the centre of mass of the body. This is our first observation input.

This was the only input that was used for the majority of this thesis, and does provide us with a lot of information. The idea is that when the robot is off balance and is leaning in one particular direction, the joints will give slightly. This give is detected in kinematics and shows as a small x offset that can be then used as input into our controller. The problem then was the controller was unable to detect larger x offsets due to increasing body incline as we allowed the robot to step on higher obstacles, i.e. obstacles larger then approximately 8mm.

To remedy this it was decided to introduce a small IMU component to this implementation

of the controller. The IMU sensor within the Nao is capable of measuring angular offsets in the sagittal and coronal plane to a resolution of 1 degree. Although this is not an accurate measurement, if used in combination with the kinematic information, it can yield a more sensitive measurement of x. The IMU outputs an angle, however we require an offset in millimetres for the purposes of the filter.

This conversion was made by assuming the robots COM was approximately 300mm above the ground. After this, basic trigonometry may be used to find the opposite side of the triangle using the adjacent and the angle supplied by the IMU. This value is then fed directly into the filter as can be seen in subsection 8.2.1.

The learning rates of alpha and beta were hand tuned during this implementation. It must be noted that beta was originally 0.5, however this was decreased to 0.25 after adding the additional IMU component.

After the observation update is complete the prediction update is executed. This update simply adjusts pendulum.x and pendulum.dx with normal pendulum dynamics under gravity. These updates are described in the article by Hengst et al. [4].

8.2.2 Action Lookup

The policy learnt in simulation is written to a file. This file lists every action for every part of the state space. A class called RLPlanner was created to read this file in and parse it. This file was stored in a 4 dimensional array.

A public method was created within the RLPlanner class to allow one to make queries of this policy. This method took as input a BodyModel and an ActionCommand. Its output was an Action, in this case, the ankle movement to make. The BodyModel was an input as this contains the pendulumModel and therefore the x, dx and t required to query the state space. The ActionCommand was given as input as this contains the current behaviour request and thus allows us to determine the goal which is required to query the state space.

Note, the goal is either 0 for backward, 1 for stand still or 2 for forward.

8.2.3 Policy implementation

After determining the action by querying the RLPlanner the Planner module must now determine how to implement this policy on the robot. There are two parts to this implementation. The first is the ankle rotation control, and the second is the step control.

8.2.3.1 Ankle Control

In simulation after taking an action the simulator was able to immediately change the pivot point. This phenomenon is not possible in real life as the ankle takes time to rotate and apply pressure to that part of the foot. Moving the ankle quickly to apply pressure should in theory provide results similar to simulation as there is less lag in the pressure being applied to either the front or back of the foot. However, if moved too quickly the ankle begins to make a chatter noise as loose parts of the feet clap against each other. This makes the walk sound ugly and harsh on the robot.

A number of variations on the controller and this implementation were tried throughout this thesis. Early versions had only three actions. Apply pressure to the back, middle or front of the foot. The implementation of this within the Planner would then have to decide how to apply this pressure to the back, front or middle of the foot. This was achieved by using the ZMP over the support foot to detect where the pressure on the foot currently lies. If the pressure was already in the correct place, no rotation was applied to the ankle. If the pressure needed to be moved towards the other side of the foot, the appropriate rotations were applied. With this method, the rate at which the ankle rotated had to be manually tuned. We found that the performance of the controller was very dependent on these tuned parameters.

The final version of the policy increased the original 3 actions, (forward, middle, back) to a

more discretised 11 actions. These 11 actions represent the ZMP being at various positions on the foot. These actions are than converted to a ZMP like figure by $(10 * a) - 50$ as can be seen in figure 8.2.3.1. This figure c, is then compared to the actual ZMP of the support foot and is used to determine the rotation of the foot. That is, if the target ZMP is 50 and the measured ZMP is 50, the rotation of the support foot is set to 0 degrees.

The value of k, as can be seen in figure 8.2.3.1, is used to determine the maximum rotation of the ankle in degrees. In this example, this maximum rotation is approximately 1.2 degrees.

```
float k = 1.2/57.0;
float a = action.ankleRotationL;
float c = (10 * a) - 50;
float p = bodyModel.getFootZMP(!bodyModel.getWalkCycle().
   leftPhase, sensors);
ankleRotationHigh =  DEG2RAD(c - p)*k;
```

8.2.3.2 Step Control

The step control was much simpler then the ankle controller. As described in the article by Hengst et al. [4] the swing foot was hard coded to move as a function of x. To implement this the walk engine had to be altered slightly such that it did not attempt to generate its normal walk cycle pattern. Instead the walk engine moved the foot directly to the position requested by the Planner module.

The Planner then set the foot position as follows:

```
if (bodyModel.getWalkCycle().leftPhase) {
    walkEngineRequest.forwardR =  -x/1.5f;
    walkEngineRequest.forwardL =  x/1.5f;
 } else {
    walkEngineRequest.forwardR =  x/1.5f;
```

```
        walkEngineRequest.forwardL =  -x/1.5f;
    }
```

The division by 1.5 is to account for some energy loss during the gait cycle [4]. Values of 1.2, 1.5 and 2.0 where tested when trying to determine the best value for this constant. Lower values such as 1.2 result in a more jerky and quick response to disturbances on the robot, while large values lead to the robot not being responsive enough. The value of 1.5 was chosen because this seemed to react to disturbances in a smooth but responsive way.

8.3 Results

At competition, this controller was run under the open challenge as rUNSWift's entry. The demo for this open challenge was as follows. Two robots were placed on the field, one with the controller and one with no controller. Both robots were given a slight forward walk and a step obstacle was placed in front of each. During this demo the controller behaved as expected and did not fall. This demo received 3rd in the open challenges [7] out of 20 teams which shows that at a subjective level, the controller was convincing.

After competition, a more objective experiment was run. The idea was to test the controller at different step heights and measure the probability of the robot falling over at these step heights.

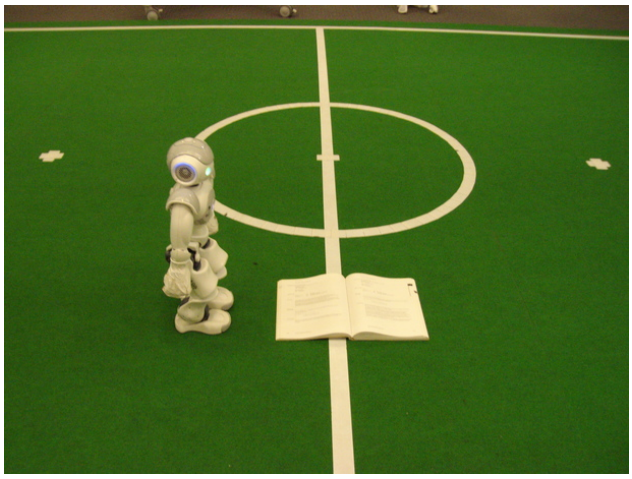

Figure 8.1: Experimental set-up.

The experiment set up was to use the pages of a book to vary this step height (see Figure 8.1). The controller was tested at 2mm, 4mm, 6mm, 8mm, 10mm and 12mm step heights. A control was also tested that used no controller.

For each step height the robot was allowed to step onto the book 20 times. The number of steps it took onto the book that led to it falling over were counted in each run. This number was then converted to a probability of the robot falling over, e.g. 4 falls out of 20 steps is an 80% chance of staying up right. Figure 8.3 shows a tabulated summary of the results of this experiment.

It must be noted that at step heights of 10mm and 12mm the robot had trouble lifting its foot high enough to step onto the obstacle. The experiment had to be run for some time before the 20 steps onto the obstacle had been completed.

After 12mm the obstacle was too high and the robot was unable to step onto it. At this point the controller never fell as it was unable to be perturbed.

	Controller	No Controller
2	1.0	1.0
4	1.0	1.0
6	1.0	0.0
8	1.0	0.0
10	0.85	0.0
12	0.6	0.0

8.4 Evaluation

The results from the variable step height experiment in Figure 8.3 were encouraging. The version of the walk that contained no controller fell over in all step heights after 6mm. The only reason it was able to remain stable at 2mm and 4mm, was that the robot's natural dynamics enabled it to walk up and over the obstacle being used in the experiment.

While using the controller, the robot was able to perform reliably up until 8mm (a 100% increase over not using the controller). While the experimental results indicate a 0% chance of the robot falling at 6mm and 7mm it must be noted that nothing is ever perfect and that if run for hours the robot would almost definitely fall over. However, in the tests run, the controller was left to step on the obstacle for at least 50 steps and never fell.

At 10mm and 12mm the robot was struggling to step onto the obstacle, however when it did it showed a decrease in performance as can be seen in Figure 8.3. This shows that if the robot was able to step higher the performance of the controller would likely degrade further. Part of the reason for this predicted degradation in performance is that the controller is not sensitive to large angular offsets from large perturbation as the primary method for sensing its body lean is small offsets detected within the robot's own kinematics.

8.5 Conclusion

This chapter has described the implementation of the bipedal control policy for the Nao that was used for the open challenge as rUNSWift's entry in 2011. Future work in improving the underlying pendulum model and the filtering of sensor values to create this model on the Nao may provide better results. In addition to this, the ability for this controller to walk about and change directions still remains untested on the Nao.

Chapter 9

Future Work

This chapter aims to document some ideas that could be explored in the future.

9.1 Dribbling

Dribbling is a concept that has only just started to be explored in the Standard Platform League. To dribble means that a robot moves the ball in a desired direction through the use of small nudges on the ball. To do this well a robot should also keep the ball relatively close to its body such that another robot is unable to steal it.

The winning team, B-Human utilised small jab dribble kicks that moved the ball approximately 1m in either the front or side directions. This could be used to get around oponents. This is similar to our approach to dribble kicks as described in this report. Another team, HTWK dribbled the ball by walking into it slowly. In fact, in our game against HTWK in which we lost 2-5 they did not kick the ball once. Instead they dribbled the ball into the goal 5 times [9].

However, HTWK's approach had the disadvantage that they were only ever able to move the ball forwards as they needed to walk into it. I believe that if kicks such as the shuffle kick were perfected, a robot would be able to keep the ball close to its legs at all time and

move the ball omni-directionally as it moves about the field.

A combination of using these specialised kicks to move around oponents and a forward dribble would allow one to maintain heavy possession of the ball in a game of robot soccer.

9.2 Coronal Rocking

As described in this thesis, the lack of coronal rock within the conservative walk caused this walk to have an unstable and slow start-up period of around 1-2 seconds. While various degrees of coronal rock were tried we believe that a more complex coronal model needs to be investigated in order to truly create a smooth rocking motion. I believe improvements in this area would improve the current walk engine by leaps and bounds.

9.3 Arm Actuation for Avoidance

At competition the team HTWK [9] fielded a walk that had its arms positioned behind its back at all times while walking. Their reasoning behind this was that the robot was less likely to get stuck on another robot as it passed by it while chasing the ball. This was important for their dribbling strategy as they needed to brush past other robot's constantly throughout the game.

At competition I began work on something similar to this that allowed behaviour to choose between putting the robots arms behind its back or leaving them at the side. While walking at top speed the behaviour is then able to choose to leave the arms to the side to create a more stable walking motion. However, once the walk slows and an opponent is detected nearby, we may then choose to put the robots arms behind its back to avoid getting stuck. This was tested in a minimal way at competition and seemed to work, however we were knocked out in the quarter finals and so we were never able to test this in a real game. I believe there is potential in actuating the arms in this way as currently, the arms do not

react at all to their surroundings which is one of the reasons robots get stuck against each other.

Chapter 10

Conclusion

This report has documented the changes made to locomotion over the course of 2011 and the implementation of the bipedal control policy for the Nao used in the open challenge.

The aim of this thesis was to improve the weaknesses in the 2010 rUNSWift walk engine such as wear and tear on the joints due to a large knee bend in the walk and a slow side step speed. An additional aim of this thesis was to develop a machine learnt controller that would run on the Nao.

The first aim was met with the development of the conservative walk in chapter 7 and its transitions from fast walk. While making these changes to the walk engine a new integrated kick engine was created. This kick engine allowed for new quick dribble kicks that are used to steal a ball from an opponent. These types of kicks are the future of competitive robotic soccer and I feel the developments made in this thesis are a step in the right direction.

The controller we learnt in simulation and transferred to the Nao ended up being used as rUNSWift's entry into the open challenge and was able to perform as expected. In the step experiment conducted it was hard to make the Nao fall over when the controller was turned on. In fact, we noted a 100% increase in the height of the obstacle a robot was able to stand on once the controller was enabled. For the open challenge we received 3rd place out of 20 teams for the demo of this experiment.

I believe that this work has shown the potential of a dynamic walk engine that is able to change its parameters on the fly. These parameters such as step height or step frequency may be adjusted to conserve energy and reduce overheating when the robot is not required to move quickly. This is similar to how a human adjusts its gait depending on the speed it is required to move. Other parameters such as those involved in the arm may be adjusted to avoid getting stuck on another robot when walking in close proximity.

However, the conservative walk developed did have problems in that it had a slow start up time after it began to walk after standing. In future work, a coronal model would help to improve the performance of the conservative walk and make this type of walk and transitioning more viable.

Appendix A

Contributions

This section is a listing of my main contributions to robocup over 2010 and 2011. This is to inform future members of modules I was involved with so that they may ask me questions if assistance is required.

- **Off-Nao** The infastructure of Off-Nao [6].
- **Forward Kinematics** Forward kinematics and its manual calibration tools in Off-Nao [6].
- **2010 Behaviours** Striker, supporter and role switching in 2010 behaviours [6].
- **Automatic Kinematic Calibration** Automatic calibration of kinematics using hill climbing [12].
- **Walk Engine** Walk used in 2011 with transition between conservative and fast walk.
- **Kicks** Kicks used in 2011 including the new dribble kicks.
- **Getup routines** new getup routines for robot with longer arms in 2011.
- **Striker** Striker component of 2011 that allows robot to line up and kick the ball.

Appendix B

Interpolate Smooth

This function is used in various parts of the WalkEngine to generate smooth motions for joint movements. The function is $start + (end - start) * (1 + cos(\pi * tCurrent/tEnd - \pi))/2$. To summarise, this function takes a section of the cosine function and stretches it out to the desired length with the multiplication $(end - start)$.

An example of this function with a start = -1, end = 1 is Figure B.1.

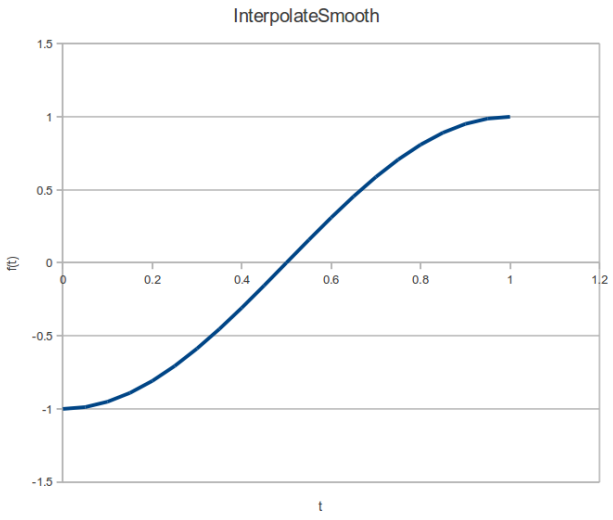

Figure B.1: Example of the interpolate smooth function.

Bibliography

[1] Aldeberan. Aldeberan documentation. http://users.aldebaran-robotics.com/docs/site_en/index_doc.html.

[2] E. Chatzilaris, I. Kyranou, J. Ma, E. Orfanoudakis, A. Panakos, A. Paraschos, G. Pierris, N. Spanoudakis, J. Threlfall, A. Topalidou, D. Tzanetatou, E. Vazaios, S. Cameron, T. Dahl, and M. G. Lagoudakis. Unconventional ball kicks with the nao. 2011.

[3] Colin Graf and Thomas Röfer. A closed-loop 3d-lipm gait for the Robocup standard platform league humanoid. http://www.b-human.de/index.php?s=publications, 2010.

[4] Bernhard Hengst, Brock White, and Manuel Lange. Learning to control a biped with feet. *Humanoids 2011 - Accepted*, 2011.

[5] Board of Trustees. Robocup. http://www.robocup.org, 2011.

[6] Adrian Ratter, Bernhard Hengst, Brad Hall, Brock White, Benjamin Vance, David Claridge, Hung Nguyen, Jayen Ashar, Stuart Robinson, and Yanjin Zhu. rUNSWift team report 2010. 2010.

[7] Robocup. Robocup 2011 SPL results. http://www.robocup2011.org/en/content.asp?PID={CCC1C5F7-D4F9-494C-A1D4-%62DDE434E5B1}&PageFn=Results.

[8] BHuman Team. Teambhuman's Youtube channel. http://www.youtube.com/user/TeamBHuman.

[9] HTWK Team. Htwk's Youtube channel. http://www.youtube.com/user/naoteamhtwk.

[10] Belinda Teh. Ball modelling and its applications in robot goalie behaviours. 2011.

[11] Miomir Vukobratovic and Branislav Borovac. Note on the article "zero-moment point - thirty five years of its life". *I. J. Humanoid Robotics*, 2(2):225–227, 2005.

[12] Brock White. Automatic calibration of the forward kinematics for the nao humanoid robot. http://runswift.cse.unsw.edu.au/confluence/display/rc2011/Kinematics.

[13] Tak Fai Yik. Locomotion of bipedal humanoid robots: Planning and learning to walk. 2007.

[14] Yongki Yusmanthia. Simulation and behaviour learning for robocup SPL. 2011.