THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

rUNSWift Robocup SPL 2013
Special Project A Report

Dan Padilha

ddp@cse.unsw.edu.au

August 17, 2013

Supervisors: Bernhard Hengst, Maurice Pagnucco

School of Computer Science & Engineering
University of New South Wales
Sydney 2052, Australia

Abstract

This report documents the work done by Dan Padilha for UNSW's Robocup Standard Platform League (SPL) team **rUNSWift** during Semester 1, 2013.

The report describes 2013's Open Challenge entry, which applied Reinforcement-Learned sagittal and coronal stabilisation policies to the robots. This aimed to combine the research of previous rUNSWift contributors to demonstrate self-stabilisation in various bipedal behaviours. The entry achieved third place overall in the Open Challenge.

Documentation for the new behaviours framework, known as *NewSkillz*, is also provided. The framework makes writing robot behaviours a much simpler process by abstracting away much of the complexity and redundancy involved in state changes, and its consistent API allows for the easy implementation of a behaviours simulator. A port of 2012's behaviours is demonstrated to work on both the robots and in the simulator with no changes necessary between them.

Finally, the 2013 Drop-In Player Challenge is described and rUNSWift's implementation and performance detailed.

Contents

1 Outline **1**

2 Reinforcement-Learned Stabilisation **2**

2.1 Introduction 2

2.2 Background & Related Work 3

2.3 Theory & Implementation 4

2.4 Results 11

2.5 Evaluation 12

2.6 Future Work 12

2.7 Conclusion 13

3 NewSkillz Behaviour Framework **14**

3.1 Introduction 14

3.2 Theory 14

3.3 Implementation 16

3.4 Future Work 19

3.5 Conclusion 19

4 Drop-In Player Challenge **20**

4.1 Introduction 20

4.2 Implementation 20

4.3 Results & Evaluation 21

4.4 Future Work 21

1

Outline

Reinforcement-Learned Stabilisation

This section describes rUNSWift's entry to the 2013 Standard Platform League (SPL) Open Challenge. This involved demonstrating the application of Reinforcement-Learned bipedal control policies on the Nao robots.

A simple introduction to Reinforcement Learning is given, demonstrating the lead-up work done by researcher Bernhard Hengst. [5] Further state measurement work by Roger Liu is also discussed. [6] Finally, the implementation of control policies, in a redesign of the work done by Brock White, is described. [9] This defines a generic way to interpret Reinforcement-Learned control policies on the Nao hardware.

Some simple tests allow a proof-of-concept to be demonstrated, and, perhaps most importantly, major focus points for future research are identified.

All work implemented has been merged into the `master` branch of the rUNSWift codebase.

NewSkillz Behaviour Framework

This section describes the new behaviours framework for future use in the rUNSWift codebase. As the justifications for the new framework are made in other reports [2], this section only touches briefly upon these. Instead, the focus of this section is primarily to document the *NewSkillz* framework and provide a gentle learning curve for future skills implementation.

All work implemented exists in the `new_skillz` branch of the rUNSWift codebase.

Drop-In Player Challenge

This section describes the 2013 Drop-In Challenge, which involved robots from different competition teams being required to demonstrate communication and team-work as a soccer team. Described are the communication standard implemented and the observed results from the Challenge. As the Challenge is expected to be run again in the future, some points of further work are also identified.

All work implemented has been merged into the `master` branch of the rUNSWift codebase.

2

Reinforcement-Learned Stabilisation

2.1 Introduction

A major goal of the RoboCup Standard Platform League is the development of improved bipedal control of the Nao robots. The teams aim to demonstrate that the robots are able to execute increasingly human-like bipedal behaviours, whether it be walking, kicking a ball, or self-stabilisation. The ability to execute these types of movements, to be able to switch between them fluidly, and to react against external forces to avoid falling over is a key research area in robotics. The responsiveness required by such movements is what is to be expected of future robots particularly when emulating human behaviours, as in playing a game of soccer.

Such behaviour is extremely difficult or practically impossible to be programmed directly into a robot. Careful study and analysis of human movement is required and mathematical models must be derived to approximate it. Even then, such models tend only to work in ideal situations, such as walking over a perfectly smooth and flat ground. In order to approach the behaviour that is inherent in humans – that of adapting to any possible situation – it is necessary that machine-learning principles are considered in order to adapt to any change in the robot's environment. For example, a hand-programmed walk will react differently over a rough and bumpy surface than over a flat and smooth surface, as it is confined to a small subset of possible robot states, whereas a machine-learned walk should in theory be able to react to either.

For the 2013 Open Challenge, we explored the application of Reinforcement-Learned behaviours for bipedal movements on the Nao robots. We aimed to demonstrate that simulator-learned behaviours could be applied directly to the Nao robots without the need for significant tuning in software. The Open Challenge entry was titled *"Stability Control through Machine Learned Behaviours"* [7].

The learned behaviours aimed to react to a greater number of states than hand-programming would allow for and to allow relatively simpler implementation of more advanced behaviours. Applied to the Naos, we aimed for self-stabilisation without the need for hard-coding in a variety of states including: changing support feet at different frequencies (walking on the spot); standing upright; standing on either foot; and seamlessly switching between these.

2.2 Background & Related Work

Reinforcement Learning (RL) is a method of deriving software models for behaviour through machine learning. RL is used to automatically determine a model for actions to be taken given an *environment state* such that a *reward* is given. The sum of future rewards is the *value* of the action in a given state. In the context of stability control for bipedal robots, RL results in a software model known as a *policy*. The policy states how the robot should actuate its motors given that it is in some environment state, wants to reach some position (defined by the reward function), and wants to do so optimally (by maximising value).

An RL model is defined by: [5]

States – the set of all states the robot can be expected to act in.

Actions – the set of all actions the robot can potentially take.

Transitions – the definition of how states transition into other states.

Reward – the definition of how good being in a certain state or doing a certain action is.

RL works by finding a function for the *value* of taking a particular action when in a particular state and following the optimal policy thereafter. The optimal value is therefore the beset one of the possible actions that can be taken in a given state. The value of each state-action is dependent on the reward – taking actions which lead to rewards quickly is of higher value, taking actions which lead to slow or negative rewards is of lower value.

The work presented in this report relies primarily on the application of RL research done by Hengst [5]. Hengst's work involved the use of a simulated environment for the robot, allowing for fast reinforcement learning of optimal policies for various behaviours (defined by their reward functions), including:

- Standing upright and still.
- Standing on one leg.
- Rocking between both legs.

Hengst's work allows for a much simpler implementation of the policies in real hardware with less need for manual learning or tuning. Testing this is one of the primary aims of the work outlined in this report.

White's [9] 2011 implementation of a simulator-learned policy for a walking gait on the Naos is also an important motivator. White was able to show that a simulator-learned walk running on the Naos was able to react to obstacles significantly better than one with no RL policy. However, White also found that state measurement was very difficult due to noisy sensors and limited data, and the work was not merged into the competition code.

Further work by Liu [6] in 2012 focused primarily on improving state measurement by introducing heavy filtering of the Inertial Measurement Unit (IMU) located in the chest of the Naos. While the demonstrated filters worked well, Liu found that the filtered values were not sufficient when introducing more complex motion such as a walking gait. This was due to the filter's underlying assumptions that the robot was standing still and upright, and approximated an inverted pendulum model.

2.3 Theory & Implementation

The work presented involves an attempt to combine the previous works of Hengst, White, and Liu in order to produce these self-stabilising behaviours, and to explore the application to walk stabilisation. Below we detail the theory and implementation of the major aspects of this work, which include: learning the control policies, measuring the current state, and interpreting the policies in the software.

2.3.1 Learning the Control Policies

In learning the policies for the behaviours, Hengst [5] defines a robot model, including as much accuracy in the links, joints, masses, etc. as possible – an example of this is seen in Figure 2.1.

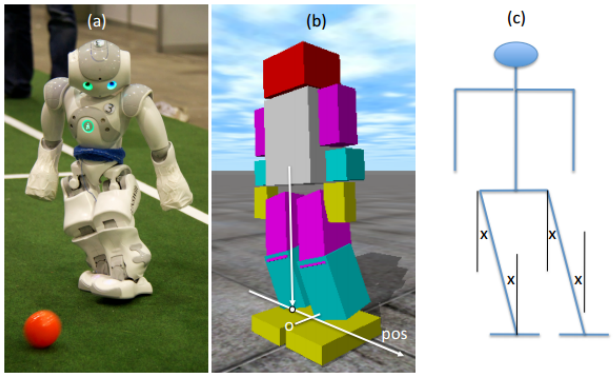


Figure 2.1 – The Nao robots modelled as a $23DOF$ (Degree of Freedom) system in (b). [5]

Hengst defines the *state* of the robot using two variables:

- position of the middle of the torso in $m$eters
- velocity of the middle of the torso in $^{m}/_{s}$

The goal or reward function of a behaviour is therefore based on getting to some state (as in standing still) or reaching some states at a certain time (as in rocking back and forth). Example policies can be seen in Figure 2.2.

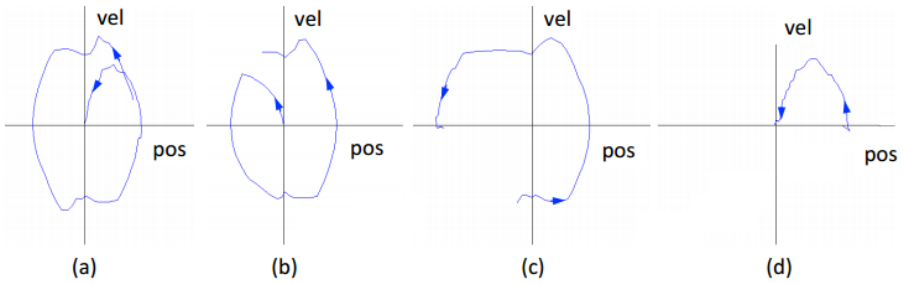


Figure 2.2 – Example of learned policies in the two state variables (position and velocity). The policy defines what actions to take which will lead to the state approaching some goal (for example, zero position and velocity for standing still, as in (a)). [5]

Knowing how the state of the environment changes based on actions is what the transition function defines. The transition function is calculated by observing the change between states of the simulation when subjected to a random choice of actions (from the set of actions that could be taken) – the random simulation essentially explores the state space of the model through the actions. This is a computing-intensive process – taking upwards of several hours to complete depending on the resolution – but is also practically impossible to define through real-life tests due to the complexity of analysing the environment, something a simulation can do very easily.

Once the state transitions are known, it is a simple case of calculating the value of each state-action for any given reward function, and then finding the optimal action to take for a given state-action (that is to say, the action to take depends not only on the current state, but also on the previous action taken). The list of optimal actions for each state-action is then known as the *optimal policy*. A reward function can be easily created for any goal desired, such as standing still or walking on the spot, so long as the goal states are in the set of explored states. Knowing the state transitions also allows for quick switching between behaviours by simply picking which policy to follow.

A total of 7 goal states were defined: 6 in the coronal plane (side-to-side motions), and 1 in the sagittal plane (front-to-back motions). These goals were as follows:

1. Standing still (sagittal)
2. Standing still (coronal)
3. Fast pace rocking between feet
4. Medium pace rocking between feet
5. Slow pace rocking between feet
6. Standing on right leg
7. Standing on left leg

The RL process learns the actions required for optimal stabilisation of these goals, and the result can be used to easily transition between the goals by simply following the policy of a new goal. The optimal policies, which define the action required for each possible state, are then defined. Some simplified example policies can be seen earlier in Figure 2.2.

The optimal policies are then given as a simple output of numbers, being the state-action pair and the corresponding optimal action to take. An example output from a control policy can be seen in Figure 2.3. This is known as a *policy file*, which can then be interpreted by the robot's software.

2.3.1.1 Sagittal Goals

In the sagittal plane, only one goal is defined: standing still and upright.

As described by Hengst [3], the goal defined for standing still is to return to a neutral torso position (with no velocity and at a zero position) and then take no action.

As a sagittal policy, we are concerned with movement of the torso in the sagittal plane (in other words, leaning forward or back). The possible actions involve actuating the hip and ankle pitch in order to return the leaning torso to the neutral position. The motor adjustment is defined very simply as:

```
// Policy table for 6 Coronal behaviours
// State-action is defined by: Goal, lateral torso position,
//                             lateral torso velocity, last action taken
// Action: 0 for ankle roll left, 1 for none, 2 for ankle roll right
//  state-action    action
   0 -0.08 -0.4  0     0
   0 -0.08 -0.39 0     1
   0 -0.08 -0.38 0     1
   0 -0.08 -0.37 0     2
```

Figure 2.3 – An excerpt of a control policy for coronal behaviours.

$$\delta = 0.01 * (\text{action} - 2) \tag{2.1}$$

The value of δ is the *adjustment* in radians added to the hip and ankle pitch angles. We let `action` be an integer from 0 to 4, so that Equation 2.1 defines a ± 0.02, ± 0.01, or $+0$ radians adjustment to the motors.

On every tick, the current state is measured, the next action is looked up from the policy, and the motor angles are adjusted:

```
float action     = sagittalRL->getAction(0, pos, vel, last_action);
float adjustment = 0.01 * (action - 2);
jointAngle[Joint.LAnklePitch.iD()]  += adjustment;
jointAngle[Joint.RAnklePitch.iD()]  += adjustment;
jointAngle[Joint.LHipPitch.iD()]    += adjustment;
jointAngle[Joint.RHipPitch.iD()]    += adjustment;
```

2.3.1.2 Coronal Goals

In the coronal plane, six goals are defined. The policies for these goals are a little more complex than for sagittal, so further information should be sought in the technical documentation provided by Hengst. [4]

The coronal goals include: standing still; fast, medium, and slow pace rocks from foot to foot; and standing still on either leg.

As some of the goals require a swaying of the robot's hips from side to side to shift the weight between either foot, part of the policy defines a *hip sway* variable as simply a constant multiplied by the current lateral position of the torso.

For the coronal plane, only side-to-side movement is important, so the actions involve actuating the hip and ankle roll motors. The adjustment to the motors is defined as:

$$\delta = 0.045 * (\text{action} - 1) \tag{2.2}$$

where δ is the adjustment in radians, and actions are 0 to 2, such that the possible adjustments are ± 0.045 or $+0$ radians.

Again, on every tick the current state is defined, the next action is looked up from the policy, and the motors are adjusted. It is important to note that some other updates and calculations are also

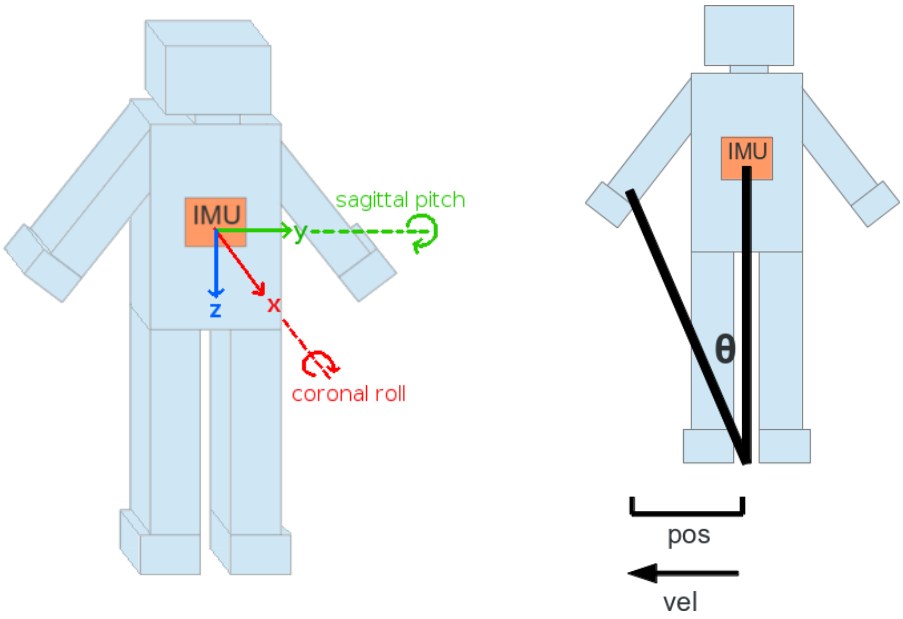
performed depending on which goal is required (for example, lifting up the legs when necessary). Notice that the hip sway adjustment is added to the ankle and hip rolls in opposition to each other, in order to maintain the center of mass over the support foot.

```
float action      = coronalRL->getAction(goal, pos, vel, last_action);
float adjustment  = 0.045 * (action - 1);
float hipSway     = 4 * position;
jointAngle[Joint.LAnkleRoll.iD()] = -hipSway + adjustment;
jointAngle[Joint.RAnkleRoll.iD()] = -hipSway + adjustment;
jointAngle[Joint.LHipRoll.iD()]   = hipSway;
jointAngle[Joint.RHipRoll.iD()]   = hipSway;
```

2.3.2 State Measurement on Hardware

The Nao robots contain an Inertial Measurement Unit (IMU) in their chest, consisting of 2 gyrometers and a three-axis accelerometer. [1] By knowing the location of the IMU, most importantly its height from the bottom of the feet (dimensions given by Aldebaran Robotics), and ensuring this matches with the simulator, one can use simple trigonometry to calculate the required position and velocity state variables.

As seen in Figure 2.4a, we can define a simple axis system from the IMU, with the x-axis pointing towards the front of the robot, the y-axis towards the side, and the z-axis pointing down to the ground. This coordinate system applies to the three-axis accelerometer, which provides a $^m/_{s^2}$ acceleration reading for each of these axes. The accelerometer readings can be used to help calibrate the gyrometer, as the accelerometer should expect to read $9.81^m/_{s^2}$ for the z-axis and $0^m/_{s^2}$ for the x- and y-axes when the robot is standing upright and still.



(a) Axes **(b)** State variables

Figure 2.4 – (a) The definition of the x-, y-, and z-axes from the IMU's accelerometers, and the lean angles from the gyrometers.
(b) The coronal state variables are the lateral position and velocity along the y-axis of the IMU. θ is the coronal roll angle given by the gyrometer.

The position and velocity of the IMU is taken with respect to its neutral position, when the robot

is standing upright. In this way, it is expected that the coronal (side to side) position of the IMU is exactly centered between the two feet. The two gyrometers provide an angular measurement of the torso, being the angle of lean about the x-axis (which we shall call the *coronal roll*) and the lean about the y-axis (the *sagittal pitch*), as seen in Figure 2.4a. When standing upright the torso is expected to be at a pitch angle of 0° relative to the hip pitch (which depends on the way the legs are bent while in a standing position), and a roll angle of 0°.

If we assume small perturbations to the torso (in other words, only small pitch and roll angles will be in the state space of the policy), then we can approximate the position of the torso by simply using the gyrometer measurements and the known height of the IMU from the base of the feet. As seen in Figure 2.4b, which shows a lean in the x-axis (or a coronal roll), we can easily calculate the horizontal displacement of the IMU with:

$$x_t = H * \sin\theta_t \tag{2.3}$$

where x_t is the position state variable at time t, H is some constant (the height of the IMU), and θ_t is the angle of the torso lean at time t.

Furthermore, for small angles of $|\theta| < 30°$, we know that $\sin\theta \approx \theta$ (in radians). Thus, we can approximate the lateral position of the IMU by simplifying Equation2.3:

$$x_t = H\theta_t \tag{2.4}$$

The velocity can be found in one of two ways:

- Integrate the accelerometer readings in the desired axis over time, or,
- Differentiate the position readings of the desired axis over time.

Both of these are relatively simple to achieve. In theory, integrating the accelerometer readings should provide an accurate velocity at all instants of time. Differentiating the position however means that only the instantaneous velocity is reported, as in:

$$v_t = (x_t - x_{t-1}) * f \tag{2.5}$$

where v_t is the velocity at time t, x_t is the position at time t, and f is the tick rate of measurements (i.e. $100Hz$ for the motion thread). This gives a $^m/_s$ measurement as required.

For the sagittal pitch, a similar process is followed. If we project lines extending out from the feet in the orientation they are facing, and take a line exactly in between them, then the position of the body in the sagittal plane is the distance of the torso from its neutral position, as seen in Figure 2.5.

Work was done on similar state measurement by Liu [6] through extensive filtering of the IMU's measurements. This had resulted in an inverted pendulum model which calculated a filtered angle for the torso lean. This filter remained in place for angular measurements. In order to maintain a relatively stable position state variable, a simple filter was used:

$$x_t^\star = \alpha * H\theta_t + (1 - \alpha) * H\theta_{t-1} \tag{2.6}$$

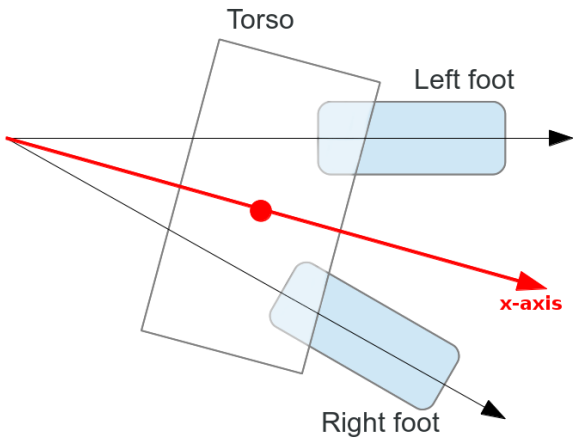


Figure 2.5 – The sagittal x-axis is always defined as being forward from the IMU (or center of the torso). The neutral point (i.e. the origin) is the location of the IMU when the robot is in a standing pose, shown here as the red dot.

For the filter, $x_t^\star$ is the filtered position state variable at time t, $H\theta_t$ is as in Equation 2.4, and α is a filtering rate between 0 and 1 (how much of the current measurement to use in the filter).

Attempts were made to also measure the state variables through integration of the accelerometer data, but, as the results in Section 2.4 show, these measurements were too unstable to be of any use.

2.3.3 Generic Policy Interpreter

As defined in Section 2.3.1 and seen in Figure 2.3, the reinforcement-learned control policies are output as a *policy file* of (state-action, action) tuples. For the sake of generality, we will now refer to these as (state, action) pairs. The policy files are then parsed by the Nao's software during runtime initialisation and used as necessary for making decisions about actions to take. A simple parser and interpreter was used by White for the 2010 RL implementation. [9] The work presented here is a redesign of White's `RLPlanner`, as well as the application of the improved policies created by Hengst.

2.3.3.1 Generic Policy File

First, we define a generic policy file format as in Figure 2.6:

```
// Comments are lines beginning with //
// The first non-comment line defines the resolution
   1     0.002   0.01  1
// Each subsequent line defines a (state, action)
// goal  pos     vel  last    action
   0    -0.080  -0.40  0        0
   0    -0.080  -0.39  1        1
   1    -0.080  -0.38  0        1
   1    -0.080  -0.37  1        2
```

Figure 2.6 – Definition of a policy file.

We define some terms:

State – A list of values defining a state. For Hengst's RL policy files, we use the tuple of (goal, position, velocity, last action taken) – these four values are defined as the state of the robot at any given time.

Action – A single value of type `Action` which defines what action to take. The `Action` type is defined in the codebase, currently as an `integer` type.

Resolution – This line contains exactly one number for every state variable. This number defines the exact distance between states for the respective state variable. In the example above, the third state variable (velocity) can only differ in steps of size 0.01 exactly. Thus, -0.40 and -0.39 are valid velocity states, but -0.395 is not.

2.3.3.2 RLPlanner

On initialisation, the class parses a generic policy file and stores it in a lookup table. Depending on the resolution of the files, this may take up to several seconds per file.

A generic policy file is parsed by creating a map from states to an `Action` type. The mappings are stored in a `C++ map<vector<int>, Action>` variable, essentially an n-dimensional lookup table, where n is the number of state variables.

First, a vector of state variable resolutions is stored.

Next, each state is parsed. To allow for simpler state approximation and mapping lookup, each value is converted to an integer by dividing the resolution of the variable. For example, a value of -0.39 in a resolution of 0.01 becomes the integer -39. The values of the state are stored as a single integer vector.

A simple interface function can be used to lookup an action from this table:

```
// Get the next action given the state variables x1, x2, ..., xn
Action a = rlplanner->getAction(x1, x2, ..., xn);
```

The `getAction` function automatically rounds the state variables to the nearest state before performing the lookup.

2.3.3.3 Using the RLPlanner

A motion generator may use a policy file by creating an instance of the `RLPlanner` class.

A motion generator, such as `StandGenerator` (which defines the behaviour for standing upright and still), simply creates an instance of `RLPlanner` for each required policy file, then on every tick looks up an action to perform, and then performs it. The actions performed are those defined in Sections 2.3.1.1 and 2.3.1.2.

Initialising a new policy simply requires:

```
RLPlanner *coronal = (RLPlanner*)(new RLPlanner("/path/to/policy.rl"));
```

2.4 Results

Several policies were tested through the `StandGenerator` with mixed success. The sagittal standing still controller proved to be most successful and is described in more detail below. The similar coronal controller resulted in random oscillations of the robot due to the extra hip sway. These random oscillations sometimes diverged and quickly destabilised the robot, even with extremely heavy filtering of the gyrometer measurements. Due to a lack of reasonable results with the coronal policies, the goals of rocking from foot to foot were not tested at all.

Sagittal self-balancing control when standing still was found to be an improvement over no control policy. Upon being disturbed, the robot would actuate its hip motors and oscillate its body back into an upright position. When reacting to pushes by human touch, the difference with and without the control policy was readily visible, though it proved somewhat difficult to quantifiably verify.

Simple tests were performed of releasing a mass suspended on a pendulum such that it would strike the standing robot with a given force – however, finding a mass that was heavy enough to cause a noticeable impact, yet soft enough as not to cause damage, proved somewhat difficult. The set-up for these tests can be seen in Figure 2.7.

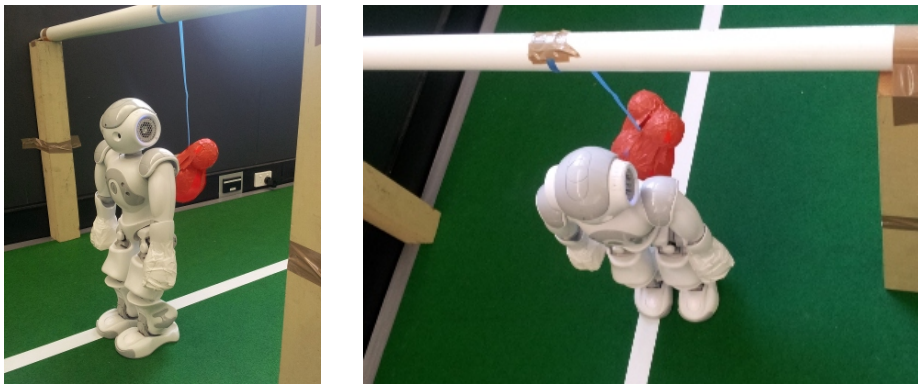


Figure 2.7 – A simple experimental set-up for testing the stabilisation reaction in the robots upon impact. A mass (in red) is released from a height such that the pendulum is parallel to the ground. Unfortunately no suitable masses could be found to provide quantifiable results.

Instead, a different test was performed. This involved releasing the robot from an angled backward lean such that the only the heels of the robot were in contact with the ground. The response of the robot from release until a stable upright position was timed. The results of these tests can be seen in Table 2.1 below. The tests clearly show an improved response time for self-stabilisation when using the sagittal control policy.

	no controller				sagittal controller			
	1	2	3	**avg**	1	2	3	**avg**
5°	2.16	2.23	2.05	**2.15**	1.58	1.40	1.77	**1.58**
10°	2.68	2.28	2.47	**2.48**	2.19	1.91	2.39	**2.16**
15°	2.77	2.80	2.69	**2.75**	2.45	2.37	2.65	**2.49**
20°	3.26	2.98	3.09	**3.11**	2.99	3.30	3.07	**3.12**
25°	–	–	–	–	–	–	–	–

Table 2.1 – Table of time taken for robot to self-stabilise after being released from a sagittal lean. Times are in *seconds*. If the robot did not become stable, a time of – is given.

This work was presented at Robocup 2013 in The Netherlands for the Standard Platform League Open Challenge. Hengst's theoretical results [5] were described, a short video of simulated policies was shown (also by Hengst), and the functional sagittal standing policy was demonstrated against a robot with no control policy. rUNSWift's Open Challenge entry was awarded 3^{rd} place.

2.5 Evaluation

The results of the stability response test show that for small perturbations the robot could recover stability significantly better with a control policy. Furthermore, the results also showed that performance did not ever degrade when compared to no control policy, so the sagittal control policy was not observed to cause diverging oscillations. This is not the case for the coronal control policies, where diverging oscillations could not be dampened even through heavy filtering of state measurements.

A minor point to be made was when the robot was in a standing still state, very tiny adjustments were still being made by the motors, causing a very small (barely perceptible) oscillation of the torso back and forth. It is likely that this was caused by the significant noise apparent in the gyrometer measurements – however, due to the nature of the noise, defining some noise threshold for adjusting the motors was not a simple task. The constant actuation of the motors is undesirable as it could potentially cause more wear and tears than without the control policy.

The three-axis accelerometer was found to be too noisy to be of any significant use for any state measurements. While standing still, the accelerometer would report a large acceleration reading in the z-axis (as expected due to gravity), but also non-zero and wildly fluctuating accelerations in the x- and y-axes. Heavy filtering of these was found to be ineffective, as the offset from zero tended to drift randomly and in a non-Gaussian manner (making it difficult to filter). Integrating the non-Gaussian noise of course lead to a runaway velocity, which made the accelerometer unusable for measuring state.

Similar issues were observed with the gyrometers, which provided extremely noisy measurements. However, the gyrometers provided a somewhat more stable and accurate reading for the torso angle than the noisy accelerometers. The angle measurement was enough for the sagittal goal to work as mentioned above.

2.6 Future Work

State measurement is currently the biggest obstacle to overcome. Finding a reliable and accurate way to measure the robot's body state is by far the biggest contributor to future stability work. More complex state measurement may involve the use of the Nao's joint position sensors to create a map of the robot's body in three-dimensional space, along with accelerometer and gyrometer data to determine attitude.

It is possible that simpler state measurement, such as tried in the work presented, may still allow for sufficient results through the use of more advanced filtering. The seemingly non-Gaussian nature of the noise in the sensors presented a challenge not able to be overcome by the author. However, as the results showed, even crude filtering could be used to some effect for the simpler stability problems like standing still.

It is quite likely that relying on just the angle measurement of the torso for coronal policies is not sufficient, at least in the real-life tests. A potential theory for this may be the opposing forces of

the hip sway on the ankle and hip joints acting to restore the torso to a zero angle but leaving the torso in a laterally displaced location. However, as the position state variable is calculated only based on torso angle, it's possible that the robot would assume it to be back in a neutral position in this case. Further investigation is required.

Further improvements can also be made to the generic policy file interpreter. Most importantly, sharing a single policy file between multiple generators is a function that should be added, as currently the system may read and store in memory the same policy file multiple times if called by multiple motion generators. No memory or performance issues were found during testing, but it is possible that large `C++` maps may not be the most efficient structure for storage, and this should be further investigated if deemed necessary.

2.7 Conclusion

The overall contributions of the author are as follows:

- The implementation of a generic RL Policy Interpreter with the use of generic policy files for future use in RL applications.
- The identification of areas where significant future work must be done for successful application of RL work.
- A self-stabilising upright stand as a proof of concept of Hengst's RL work being applicable to the Naos.
- A 3^{rd} place entry in the SPL Open Challenge of Robocup 2013.

The work presented proves that control policies learned on simulations of the robots can be successfully used to perform self-stabilisation on the robots. This proves that the theory is sound and applicable to real-life hardware, and eventually applicable to more stable bipedal behaviours. A generic policy file format has been defined and an improved policy interpreter written – both tools can be further improved for use in future exploration of RL applications.

Major failure points have been identified in state measurement. If the work presented here is to be expanded on, more must be done on improving the accuracy of measuring the robot's current position and velocity, or some other way of defining the robot's state must be found.

3

NewSkillz Behaviour Framework

3.1 Introduction

The work presented here describes the new behaviours framework for rUNSWift and is the combined effort of the following authors: Beth Crane, Jack Murray, Dan Padilha, Stephen Sherratt, and Alexander Whillas.

The *NewSkillz* behaviour framework is a new implementation of the `Python`-based control logic for the rUNSWift codebase. The new framework aims to replace the existing `Python` skills by allowing for a much easier learning curve, easily understandable code, the removal of redundancy through abstraction, and less need for complex state transition models. Furthermore, the framework is abstracted in such a way that allows the same behaviour code to be run on the robots as well as in simulators with no modification required.

The 2012 behaviours were ported in a mostly-working state to the *NewSkillz* framework, as a proof-of-concept. Examples in this section will stem from this proof-of-concept implementation.

As it required the effort of many authors, information about *NewSkillz* is spread throughout various reports. In particular, more information about the integration with a simulator and the reasoning behind the framework can be found in the report by Crane and Sherratt. [2] The focus of this section therefore is to provide a gentle introduction to implementing skills using the *NewSkillz* framework, as well as documentation of its major features. Note that the terms *behaviour*, *skill*, and sometimes *state*, are used interchangeably throughout the text.

3.2 Theory

The behaviours framework is what controls the behaviour of the Nao robots when playing soccer. Specifically, this refers to all high-level decisions the robots must make. That is to say, the behaviours do *not* define how a robot's body moves in order to make it walk, how the robot analyses a video image to find the ball, or how a robot communicates with its team-mates.

Instead, the behaviours framework defines how a robot decides whether to go for the ball or move to another position, whether to kick the ball or line itself up at a better angle, whether to dive for the ball as a goalie, and so on. This high-level decision-making requires rapid development during the competition in order to adapt to the opposing team strategies. For this reason, a high-level programming language (`Python`) is used for the greatest ease of development.

The *NewSkillz* framework differs from the 2012 skills framework in that it is not programmed explicitly as state-machines. Instead, the states arise from inherent properties of the framework while still being as simple to reason about. State is chosen through a tree-like structure of skills where each skill decides whether to *delegate* its decision to another skill, or to perform some action in a *tick*.

The skill tree can therefore be thought as a form of decision tree or state-machine. Each skill in the tree is expected to delegate a decision down to a skill below it. Leaf skills – those which do not need to delegate – are required to perform some sort of action in the real world, such as kick the ball.

To better illustrate this concept, the skill tree for the proof-of-concept port of the 2012 skills is shown in Figure 3.1.

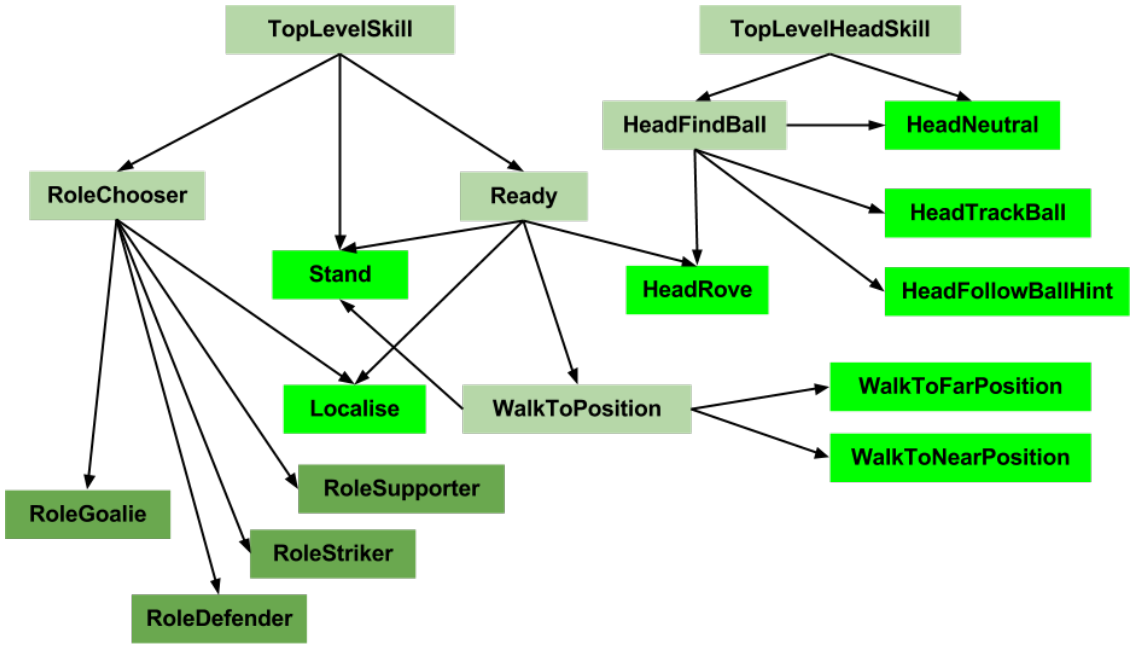
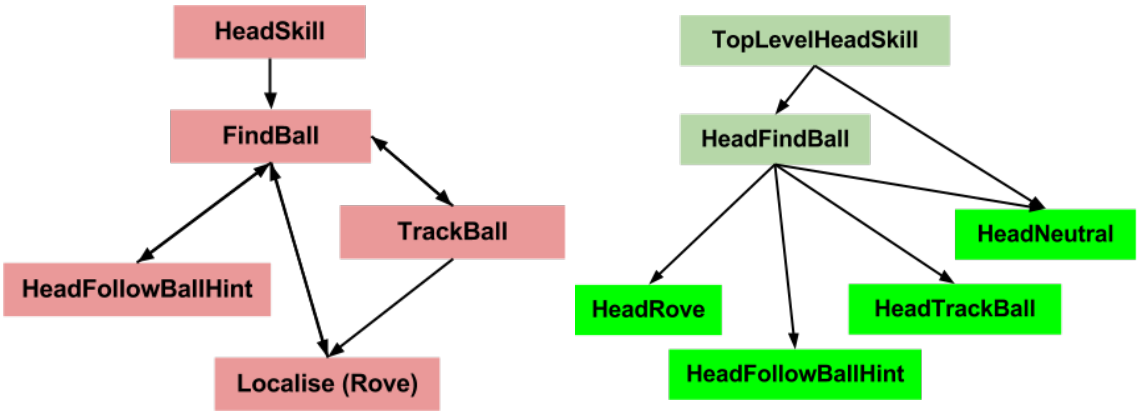


Figure 3.1 – Tree of the proof-of-concept port of the 2012 skills. Skills in bright green are leaf skills (which must perform some action). Skills in dark green are un-expanded nodes (i.e. the skills they delegate to are not shown). An arrow from a skill represents a delegation to another skill.

Notice that the tree contains two root nodes, being `TopLevelSkill` and `TopLevelHeadSkill`. This allows for decisions to be made for the body and head separately.

The skill tree system provides cleaner and less redundant code through the delegation mechanism. It is easy to imagine this by taking the state machine of the head as in the example provided by Figure 3.2.

While the 2012 state machine may look simpler, code inspection reveals a much more complex system. For starters, each skill (which we will refer to as a state) in the 2012 framework must include a state transition function defining when to transition to a different state. States can transition to states which transition back to the parent states, or they can transition in only one-way – this requires redundant checks for the condition to transition back into a state. Furthermore, any state can potentially cause an action or only perform state transitions. While this framework allows for much flexibility, it requires a much greater learning curve and unintuitive coding style.



(a) 2012 framework

(b) *NewSkillz* framework

Figure 3.2 – Example of the state machine approach of the 2012 skills framework, and the *NewSkillz* skill tree approach.

By contrast, the *NewSkillz* skill tree may look equally complicated, but becomes significantly easier to understand and write for once the required properties are understood:

- All state transitions are one-way.
- A state/skill may only transition to another state, or perform an action, but not do both.

These properties inherently create a state-machine, so they allow for the same flexibility as the 2012 framework, albeit not in the same way. For example, a *NewSkillz* skill cannot possibly be written in such a way that it can do both state transitions *and* perform an action. Instead, the programmer is forced to partition the skill into a decision and its possible actions, leading to simpler modular code.

3.3 Implementation

3.3.1 Codebase

The rUNSWift codebase is split into the `C++` backend and the `Python` behaviours framework. The behaviours framework is run through the `Perception` thread (running at $30Hz$) in

`runswift/robot/behaviour/python/PythonSkill.cpp`

This defines the `C++` to `Python` interface. It also ensures that **live-reload** for skills works – a robot can be left running in a crashed state, modified skills can be pushed to it, and it will simply continue running as if it had never crashed.

The `Python` entry-point is located in `runswift/image/home/nao/data/behaviours/cpp_glue.py`

On every tick, `cpp_glue` ticks both the `TopLevelSkill` and `TopLevelHeadSkill`. Because the body's `TopLevelSkill` is ticked *after* the head, any skills in the body skill tree can choose to delegate to a skill in the head tree and thus **override** the head tree's actions.

All skills are located in `runswift/image/home/nao/data/behaviours/skills`

3.3.2 In Real Life (IRL)

The `IRL` class is an abstraction of the state of the robot, and also an interface to perform actions, such as moving the head, walking, or kicking the ball.

The `IRL` abstraction works to such that skills can rely on the state (known as `Sense`) and `Action`s, regardless of whether they come from the rUNSWift codebase, or from the simulator. [2] This clearly allows for the same skills to be used for both platforms, with only the `Sense`s and `Actions` being written separately.

3.3.2.1 Sense

The `Sense IRL` defines a set of functions which provide the state of the robot at any instant. Running on the robots, the `Sense IRL` works simply as a wrapper for the rUNSWift `Blackboard`, which provides a shared state between all threads. Examples of `Sense`s include: `is_ball_visible()`, `game_state()`, `is_lost()`, and so on.

`Sense`s are defined in `runswift/image/home/nao/data/behaviours/robot_sense.py`

Any skill can access the `Sense` functions by simply calling `self.irl.sense.`*`some_sense_function()`*.

3.3.2.2 Action

The `Action IRL` defines a set of functions for interacting with the

Any leaf skill can perform an action by calling `self.irl.action.`*`some_action(argument)`*. By definition, actions are only allowed to occur inside the `tick` function of a skill (see Section 3.3.3.1).

3.3.3 Skills

The `Skill` class is the "bread and butter" of the *NewSkillz* framework. This class abstracts away state transitions entirely from the programmer and provides the `IRL` to all skills automatically. The class also defines the argument-passing interface for skills.

Every skill in the skill tree is a subclass of `Skill`. `Skill` provides two functions: `tick` and `delegate`. When implementing a skill, exactly **one** of these functions must be overloaded, depending on whether the skill expects to *delegate* its decision to another skill, or *tick* an `IRL` Action.

Both the `tick` and `delegate` functions can be defined with required and optional arguments. These arguments are supplied when skills delegate to other skills. This process is described more in Section 3.3.3.2.

3.3.3.1 Tick

The `tick` function is defined only for leaf skills which do not expect to delegate to any other skills, and instead expect to perform some `IRL` Action.

Good design is to have very atomic leaf skills that perform a single action, perhaps with some parameters, but with little to no conditional logic.

3.3.3.2 Delegate

The `delegate` function is defined only for skills which expect to delegate to another skill to perform some action. These skills perform all the logical decisions about states.

Good design should ensure that skills cannot possibly delegate in a cyclic manner (i.e. traversing the skills tree should guarantee arrival at a leaf skill).

3.3.4 Geometry

A simple geometry library is implemented which defines some `Vector`-like objects and all associated functions.

The geometry library is in `runswift/image/home/nao/data/behaviours/geometry.py`

There is one primitive class defined in geometry known as a `Point`. A `Point` simply represents an (x, y) tuple. Three subclasses are defined: `Vector`, `UnitVector`, and `DirectionVector`. All of these perform in exactly the same way, and differ only by their initialisations:

Point takes an x and y coordinate.

Vector takes a length and a direction in radians.

UnitVector takes a direction in radians and becomes a vector of length 1.

DirectionVector takes a direction in either degrees or radians and becomes a unit vector.

These geometry classes should be used for any representations of points on the field, direction facing, point to walk to, relative distances, and so on. The `Point` superclass defines many useful vector functions, including rotations, lengths, subtractions, etc.

An example works as such: if a robot is located at a $robot\_pos = Point(x, y)$, is facing in the $robot\_facing = DirectionVector(45°)$, and wants to turn towards the ball at $ball\_pos = Point(x, y)$, then the amount the robot needs to turn is given by the angle:

$$(robot\_pos \gg ball\_pos).rotate(robot\_facing).angle() \tag{3.1}$$

Here, the $\gg$ (right-shift) operator returns the vector from $robot\_pos$ to $ball\_pos$. This vector is then rotated such that the $robot\_facing$ vector becomes the $x-$axis. This gives the vector to the ball relative to the robot's facing direction. Finally, one can extract the angle which the robot needs to rotate to in order to face the ball.

Alternatively, and even more succinctly:

$$robot\_facing.angle\_to(robot\_pos \gg ball\_pos) \tag{3.2}$$

The geometry library thus allows for much simpler calculations to be made for many vector operations.

3.3.5 Debugging

All `Python` runtime exceptions and errors are caught by `cpp_glue`. `cpp_glue` keeps watch on the skill-tree in order to print debug information about it, and catches all `Python` exceptions in order to place the robot into "Emergency mode".

Under "Emergency mode", the robot stops all movement, flashes all its LEDs in random colours, and audibly requests to be picked up. The stack trace produced by `Python` shows the faulting skill as well as its parent skill, but no further back. If a bigger stack trace is required, simply print out the skill name in the `tick` function of `Skill`.

Thanks to the live-reload functionality built into `cpp_glue`, new skills can be pushed to the robot while in "Emergency mode". Once a `Python` file is modified on the robot, *NewSkillz* reloads all skills and the robot continues from whatever state it was in (as the `Blackboard` state is preserved).

3.4 Future Work

Further work needs to be done on the *NewSkillz* framework. The architecture is in place and the design works well both for writing skills and for implementation within a simulator, so future work should look at improving the design where necessary. Some possible improvements include:

- Implement a method of locking state to a specific leaf skill. When, for example, the robot decides to kick by calling a `KickBall` leaf skill, the framework should not switch to another leaf skill until the action is completely done. As safeguards are already in place in the underlying `C++` code (no further actions can be performed until a kick is finished), this has yet to be an issue, but it may be in future.
- A graphical interface, most probably with an `ncurses` library, for dynamically displaying the skill tree while the skills are running. This could show the current decision path taken through the skill tree at any given moment, giving an easy way to diagnose bugs.

Currently only a semi-working port of the 2012 skills framework exists which runs on the robots. Any further work on rUNSWift behaviours should focus primarily on continuing the porting of the older skills framework which provide advanced competition-ready behaviour. It is also worth noting that the 2012 skills were improved during 2013, so the port should focus on the latest.

3.5 Conclusion

A major advantage resulting from the *NewSkillz* framework is the significant cut down in redundant code due to the `IRL` abstraction. By design, all skills have access to the `IRL`, and therefore to the `Blackboard` – which needed to be passed around to everything in the 2012 framework. The standard `Skill` class also helps to abstract state transitions away, allowing for a simpler skill tree while retaining the flexibility of state transitions.

The *NewSkillz* framework therefore allows for simple implementation both on hardware and in a simulator. Future work on rUNSWift should ensure that the behaviours are written to a competition-ready state. The framework allows for more readable and easier to understand code, which should help significantly ease the pain of rapid development and debugging during the Robocup competition.

4

Drop-In Player Challenge

4.1 Introduction

One of the challenges in the 2013 SPL competition was the Drop-In Player Challenge, which required robots to communicate and cooperate on teams comprising robots from other teams. This work expands on rUNSWift's existing modular framework as defined by the 2010 design [8].

4.2 Implementation

The challenge expected robots to communicate using a common *C* structure, sent as a single network packet and broadcast to the entire team on a specific port. The necessary broadcast structure, seen below, was extremely close to the one already existing for robot communication by the 2010 design.

```
struct DropInBroadcastInfo {
    char header[4];     // "PkUp"
    int playerNum;      // 1-5
    int team;           // 0 is red 1 is blue
    float pos[3];       // position of the robot in cm, x,y,theta
    float posVar[3];    // position covariance
    float ballAge;      // seconds since this robot last saw the ball. else -1
    float ball[2];      // position of the ball
    float ballVar[2];   // covariance of position of the ball
    float ballVel[2];   // velocity of the ball
    float penalized;    // seconds the robot has been penalized or -1
    float fallen;       // seconds the robot has been fallen or -1

};
```

The only change deemed necessary for a cooperative robot was therefore to swap the Receiver and Transmitter modules with ones capable of converting between the `DropInBroadcastInfo` structure and rUNSWift's `BroadcastInfo` structure.

New variables required, such as ball velocity, were easy to implement as they were already calculated and stored in the Blackboard by the perception thread. Some rUNSWift specific variables, such as current action being performed, had to be removed. The result were two new modules: DropInReceiver for analysing information from team-mates, and DropInTransmitter for broadcasting data to team-mates.

One can easily switch between the DropIn and normal Receiver/Transmitter modules by setting the configuration option `game.type` to `DROP_IN`.

4.3 Results & Evaluation

As expected, rUNSWift's robots reacted as if they were playing on teams with their own robots. They demonstrated team work with the other robots by switching roles depending on their teammates' positions. The behaviour was enough to give rUNSWift a 3^{rd} place overall in the challenge.

While no issues with the implementation were observed, it is worthy of noting that for this year the `DropInBroadcastInfo` structure was considered an *optional* part of the challenge. Reading the broadcast packets showed that at least half of the teams were not sending the same structure, and as such could only be considered to be sending garbage (as there is no way of determining what information they were providing). This resulted in less team-work than expected, as some robots were effectively not communicating.

4.4 Future Work

The lack of standardisation for communication was acknowledged by the judges in hindsight; the competition is expected to be run again in the future with tighter restrictions. As such, there is enough reason for improvements to be made to the system.

Furthermore, it is possible that defining and proposing a communications standard to be used by all teams may be something to look into. Passing the ball between robots is expected to be a major aspect of the competition in the near future and drafting a proposal may give the teams room to add more advanced features, such as robots calling when they are open to accepting a pass, or to where they are passing to.

Bibliography

[1] Aldebaran Robotics. *NAO Software 1.14.3 documentation – NAO H25 Inertial Unit.* http://www.aldebaran-robotics.com/documentation/family/robots/inertial_robot.html.

[2] Beth Crane and Stephen Sherratt. "rUNSWift 2D Simulator; Behavioural Simulation Integrated with the rUNSWift Architecture". UNSW School of Computer Science and Engineering, 2013.

[3] Bernhard Hengst. File-Note: "Reinforcement Learned Sagittal Balancing Policy for Nao", 2013.

[4] Bernhard Hengst. File-Note: "Reinforcement Learned Sideways Stepping Policies for Nao", 2013.

[5] Bernhard Hengst. "Reinforcement Learning of Bipedal Lateral Behaviour and Stability Control with Ankle-Roll Activation". In *WSPC Proceedings*, 2013.

[6] Roger Liu. "Bipedal walk and goalie behaviour in Robocup SPL". UNSW School of Computer Science and Engineering, 2013.

[7] Dan Padilha and Bernhard Hengst. rUNSWift 2013 Open Challenge Entry: "Stability Control through Machine Learned Behaviours", 2013.

[8] 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.

[9] Brock White. "Humanoid Omni-Directional Locomotion". UNSW School of Computer Science and Engineering, 2011.