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

Four Legs and a Camera:

The longest of journeys start with the first step

Ryan Cassar (3100199)

Thesis submitted as a requirement for the degree
Bachelor of Science (Computer Science) Honours

Submitted: September 15, 2006

Supervisor: Dr William Uther

Assessor: Professor Claude Sammut

Abstract

Locomotion is a fundamental requirement for the Robocup 4 Legged League competition. Any improvements made in this area can have wide reaching effects. Fortunately, rUNSWift has a well established walking engine that works well. However, rUNSWift over the last two years been experiencing more robot injuries than other teams, and this is usually attributed to our walking style.

In response to this, a new walk was developed with the robot's longevity in mind. With the ideas raised, a more flexible and gentler walk was developed, but failed to exceed the game performance of last years walk.

However, during development several useful tools were implemented that helped the progress of other areas. In particular the python tools written in order to easily test walks, helped prototype the behaviours faster and more effectively.

Acknowledgements

First and foremost I would like to thank Dr William Uther for giving up his time to guide rUNSWift throughout the year. Thanks must go to the 2006 team, Andrew Owen, Oleg Sushkov, Michael Lake and Eric Enyang Huang. Thanks to Brad Hall for keeping us healthy and sane throughout the competition and the months leading up to it, and Claude Sammut for his constructive criticism. The whole team appreciates the NUBots' and MiPal's effort in our friendly games. Apologies to my friends and loved ones, who I neglected in place of my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives an overview of some of the background required to understand the remainder of the report. Firstly the domain of the problem is described, then a short description of past work. Finally a brief explanation of what is to come in Chapter 3 and in the report overall.

1.1 Robocup 4 Legged League Domain

Robocup is an international robotics competition held annually. The 4 legged league is the only league in the competition with a standard platform. Each competitor fields a team of four Sony AIBO ERS-7s. Technical specifications for them can be found in [10]. The robots are placed in a largely controlled environment, where they are designed to go head to head against another team, playing robotic soccer. The field's dimensions are shown in Figure 1.1.

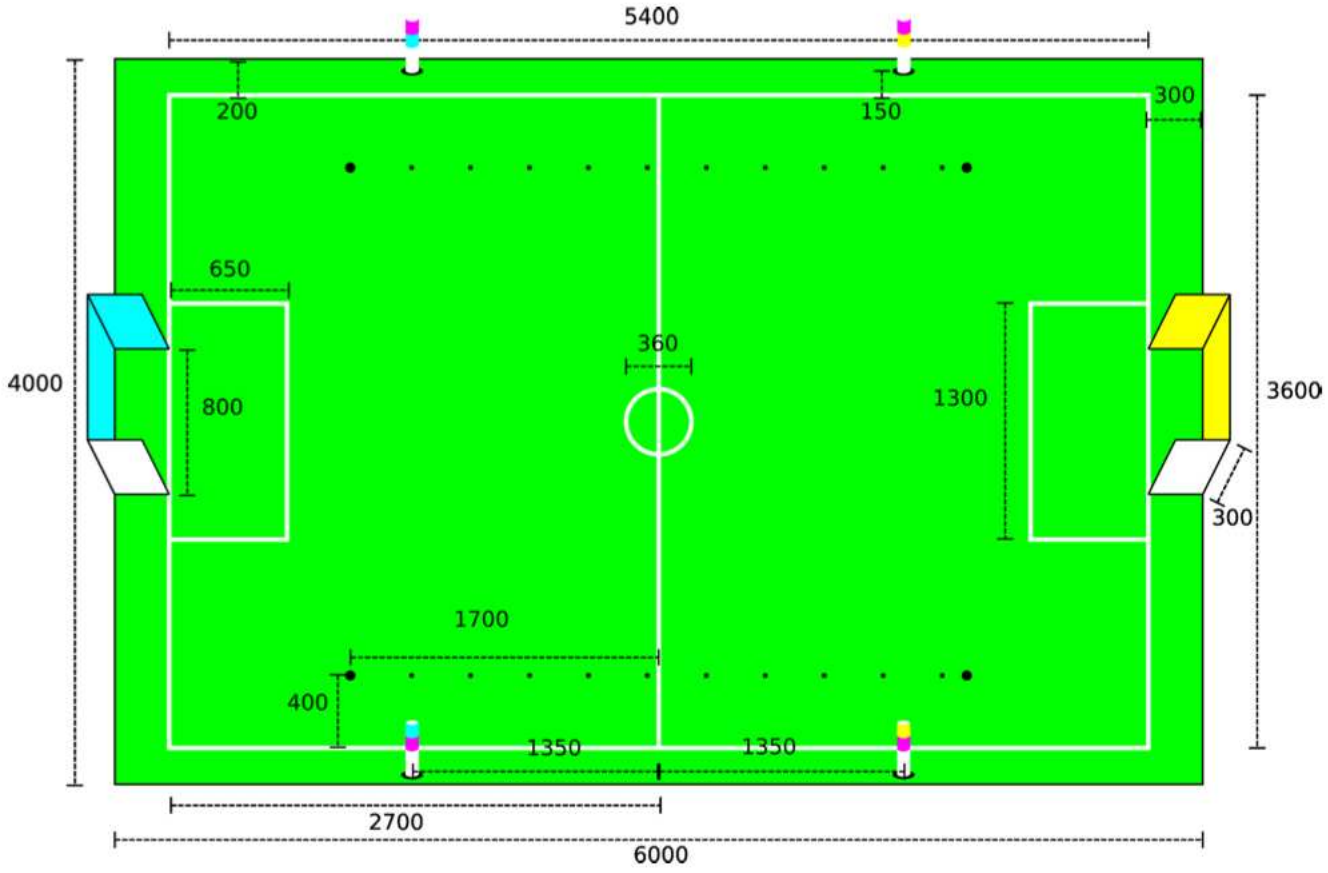


Figure 1.1: Picture of the field, as seen in the 2006 Rules [2]

The environment is "largely" controlled, as we are guaranteed certain properties of the field, like it's dimensions, but other aspects are more vague. Colour choice for the landmarks and field for example are open to interpretation. At this years competition the "pink" landmarks were very similar to the red uniforms. Andrew Owen discusses this problem and vision related issues in his work [3]. It is also very common for fields to have uneven lighting, where corners of

the field are much darker than the centre. The actual material the field is made of is also under specified. The surface can be very hard or very soft, and the robots must to be flexible enough to be able to walk on whatever carpet the competition provides.

Figure 1.2: Photo of a robot shooting on goal, as seen in the Sydney Morning Herald [1]

A basestation acts as a referee's whistle, broadcasting game events to all eight robots. Game events include a goal being scored for example. The challenge is that the team of robots must be programmed to run completely autonomously, aside from the referee's whistle.

Just like human players, the robots are required to follow rules or they will be penalised. Although knowing all the rules is not necessary to understand this report, if interested, the complete ruleset can be found here [2].

1.2 The Then

When designing an algorithm, it is often a good idea to break the problem down into simple steps. In the case of a walking engine, it seems intuitive that these simple steps are moving each leg along points of a locus. Then at any point in time we only need to be aware of what part of the locus we are up in order to give us our next point. At this stage, inverse kinematics can be used to discover what joint angles are needed next. This approach matches every walk rUNSWift has developed over the years.

Adopting this method means locomotion is a matter of deciding which locus to use for each leg at the start of each step. Completely arbitrary loci would be difficult to work with, so each walk has some closed form representation. Previous years loci have included rectangles [4], trapeziums [6], planar quadrilaterals [6], semi-ellipses [7] and SkellipticalWalk's compound trapezium/ellipse [12].

1.3 The Now

Each year has added more complexity to the locus parameterisation, up to the SkellipticalWalk of last year. SkellipticalWalk has a very rich parameter set, but is not particularly easy to understand. It should also be noted that all of the previous loci have been restricted to a plane.

This year goes back to basics, with a simple representation that also allows for a locus to move in all of three-space. Details are explained in Chapter 3.

1.4 Report Overview

Chapter 2 gives background information related to the body of the report. It discusses why it is important to research this area, and related problems with working with real legged robots.

Chapter 3 is the bulk of the report. It details the new walking engine and how machine learning was applied to the problem. The chapter concludes by discussing the calibration process.

Chapter 4 describes some of the supplementary tools that were developed to further the research.

Chapter 5 contains research that does not fall under the realm of Chapter 3.

Chapter 6 has experimental data relating to the main matter, and discusses the results.

Chapter 7 wraps up with final conclusion and suggestions for future research.

Chapter 2

Background

In this chapter we will discuss briefly the reasons walking is important in the four legged league domain, and associated issues. This is followed by expanding on Chapter 1 as previous work of another top team, GermanTeam is accredited.

2.1 Importance of Superior Walking

As the size of the four legged league field gets larger over the years, faster and more manoeuvrable walks are growing increasingly more important. Dominating over possession of the ball gives a much better chance to dominate the game. Beating your opposition to the ball is the first step to dominating possession of it. Walking faster than your opposition is an easy way to get to the ball first. Thus, a fast walk can improve the general ability of the team to dominate the game.

All this aside, the results of the competition this year show that a team with a faster walk does not necessarily overpower a slower team. Several teams arrived with faster forward walking speeds and movement in general, but failed to provide the necessary strategy to exploit the extra time gained. The lesson to take away from this is that it is the strength of the team's weakest link that will most likely determine the outcome.

2.2 Problems with Legged Locomotion

Legged locomotion is significantly more complicated than wheeled locomotion. Wheels have two degrees of freedom, speed and direction. Disregarding slipping on whichever surface you are moving over, and troublesome motors, wheels come pretty close to doing what you ask them to.

The ERS-7's have three degrees of freedom in each leg, an abductor, a rotator and a knee joint. Specifics can be found in the technical manual [10]. In the wheeled case, the speed and direction can be seen as operating in independent dimensions, where the direction does not affect the speed and visa versa. However, legged locomotion, changing the type of movement in a joint can cause very different results depending on the other joint angles.

The highly dependant nature of each dimension in the joint space makes it difficult for people to predict the result of a change in joint policy. This means it is challenging to improve on an existing walking style.

2.3 Previous Work

Walking is a somewhat critical skill in the legged league, so it is no surprise other teams have performed research in the area. As such, in addition to building on the work of previous rUNSWift teams, investigating other team's progress can prove interesting. In particular, GermanTeam2005 [11] inspired the idea for the simple and flexible walk developed by rUNSWift this year. The locus type is explained in detail in Chapter 3.

Chapter 3

Gait Optimisation

Gait optimisation breaks down into three major parts. The first is the way the legs actually move in order to locomote, the walk locus. Secondly, how to learn better parameters of the walk. Lastly, how accurately we can measure what actually happened after attempting particular walks, the odometry.

3.1 A New Way of Walking

As of 2006, the rUNSWift used a walktype known as SkellipticalWalk [12]. Although this walk is faster and more manoeuvrable than previous rUNSWift walks, it is also somewhat destructive and does not take advantage of the robot's full range of movement. In an attempt to improve this, a new walk type entitled "PolygonWalk" was developed.

3.1.1 PolygonWalk

As the name suggests, PolygonWalk uses a polygon for it's walking locus. Each vertex is given a point in 3-space, (x, y, z). Each edge is given it's own percentage of the overall time for the step, allowing different speeds at different parts of the locus. In addition to this, the front and back legs are given different shaped loci. Therefore, if there are n free vertices in the polygon, then the locus would have $((3+1)*2*n)$ parameters. As $n \to \infty$ the polygon can more closely approximate an arbitrary curve.

This gives rise to a trade off. Increasing the value of n, gives more flexibility to the walk, but significantly increases the size of the parameter space. This is an issue when attempting to search the space in order to learn a better parameter set. A SkellipticalWalk type locus that is already known to be sufficient can be approximated with $n = 5$ so this was chosen as the number of free vertices in the locus.

Each leg has a home point where it is positioned in the stationary stance. This home point coincides with the origin of the locus. When changing loci between steps, the movement is only continuous if both loci start and end at the same point. To this end, the home point is included as the first point in every locus. As such, PolygonWalk actually follows a 6-gon, with the 6th vertex fixed at the origin. Fixing the 0th vertex at the origin also forces the stationary stance to be a sensible representation of the general walking stance.

A comparison of the SkellipticalWalk type locus and the similar PolygonWalk type locus is shown in Figure 3.1.

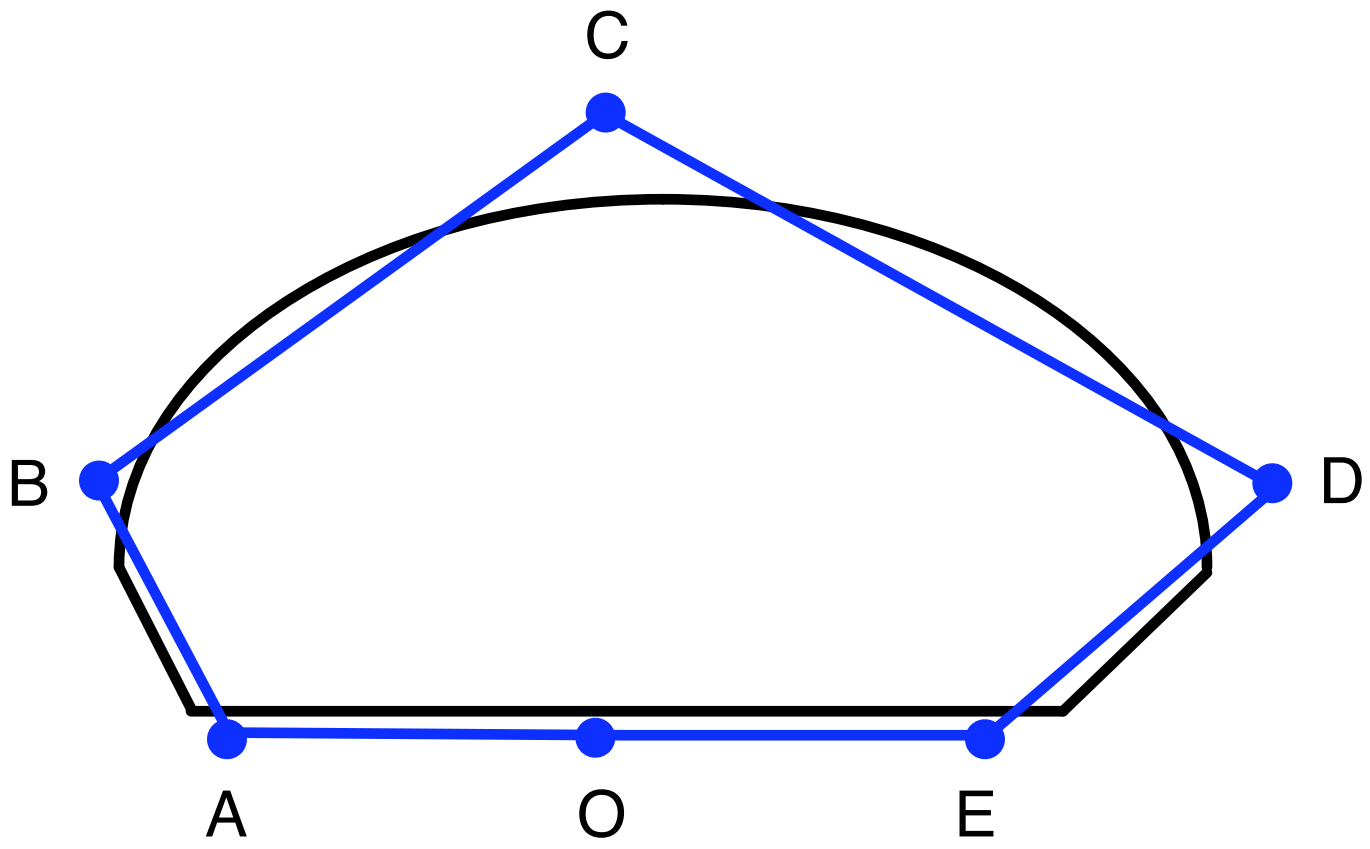


Figure 3.1: A SkellipticalWalk locus and the resulting PolyipticalWalk overlaid.

3.1.2 Extra Parameters

In addition to the forty parameters required for defining the shape of the polygon, PolygonWalk has a further two extra parameters it shares with SkellipticalWalk, sidewaysShift and frontHomeAngleDeg. The battery is located on the left hand side of the robot, and is much heavier than the body of the robot. This makes the robot asymmetrical, so the sidewaysShift adds a constant sideways component to all points on the locus, in an attempt to restore symmetry. The frontHomeAngleDeg parameter moves the contact point with the ground of the robot's front cowlings.

A list of all the standard parameters, and extra parameters for both SkellipticalWalk and PolygonWalk are in Table 3.1. For an explanation of the various parameters, see [12].

Standard Parameters	SK	PW	Extra Parameters	SW	PW

halfStepTime	✓	✓	frontTrnLftHeight	✓	×
frontH	✓	✓	backTrnLftHeight	✓	×
backH	✓	✓	frontDutyCycle	✓	×
frontFwdHeight	✓	×	backDutyCycle	✓	×
backFwdHeight	✓	×	frontLeadInFrac	✓	×
frontF	✓	✓	frontLeadDownFrac	✓	×
frontS	✓	✓	frontLeadOutFrac	✓	×
backF	✓	✓	frontLeadUpFrac	✓	×
backS	✓	✓	backLeadInFrac	✓	×
			backLeadDownFrac	✓	×
			backLeadOutFrac	✓	×
			backLeadUpFrac	✓	×
			thrustHeight	✓	×
			canterHeight	✓	×
			sideOffset	✓	✓
			turnCenterF	✓	×
			turnCenterL	✓	×
			frontLeftH	✓	×
			frontBackForwardRatio	✓	×
			frontBackLeftRatio	✓	×
			frontRollWalkRatio	✓	×
			frontHomeAngleDeg	✓	✓

Table 3.1: Standard and extra parameters for PolygonWalk and SkellipticalWalk.

3.1.3 Locomotion Realisation

So far, it has been shown that given a locus shape, the legs of the robot can trace out this path in an attempt to walk. But what we really need is a method of realising the behaviour's desires. The behaviour module requests a walk in amounts of "forward", "left" and "turn" it wants to make. The task therefore is to translate "forward", "left" and "turn" into loci. It turns out there are at least 2 ways of thinking about how to do this.

Sum of Individual Components

This is perhaps the most natural way to think about interpreting walk commands. That is, construct the appropriate loci for the forward, left and turn components, and then add the vertices pair wise. Nothing tricky there, except when trying to use large values for any two of these components. This often makes a locus that the robot is incapable of following, and the result is usually not what was intended.

To prevent excessive slipping and fumbling around when behaviours requests an "impossible" walk, when using this method the walking engine must clip the components of motion to reasonable levels before processing them. This restricts the possible manoeuvrability of the robot.

However, after clipping the components, the robot mostly moves as requested. This approach allows for an easy method for calibration, where each component can be calibrated independently. Indeed, this is how the components of motion have been combined in previous years.

Moving Turn Centre

This is a somewhat less intuitive way of interpreting walk commands. Every movement can be seen as moving along some arc. This translates to using a turn locus, but moving the centre of the turning circle such that it is not necessarily at the centre of the robot. Now we just need to derive the co-ordinates of the translated centre and to use the given turn amount as the angle to move through.

Given our three components of motion [1], we can derive Figure 3.2.

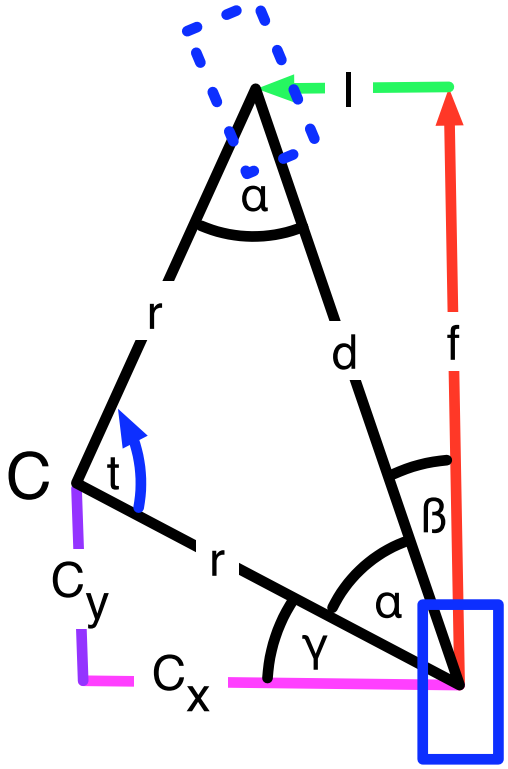


Figure 3.2: C marks the centre of the turning circle. The robot starts at the solid blue rectangle and the result of the motion moves it to the dotted rectangle.

$$
\begin{aligned}
d &= \sqrt{f^2 + l^2} \\
\alpha &= \frac{\pi - t}{2} \\
\beta &= \arctan \frac{l}{f} \\
\gamma &= \pi - \alpha - \beta \\
r &= d\frac{\sin \alpha}{\sin t}
\end{aligned}
$$

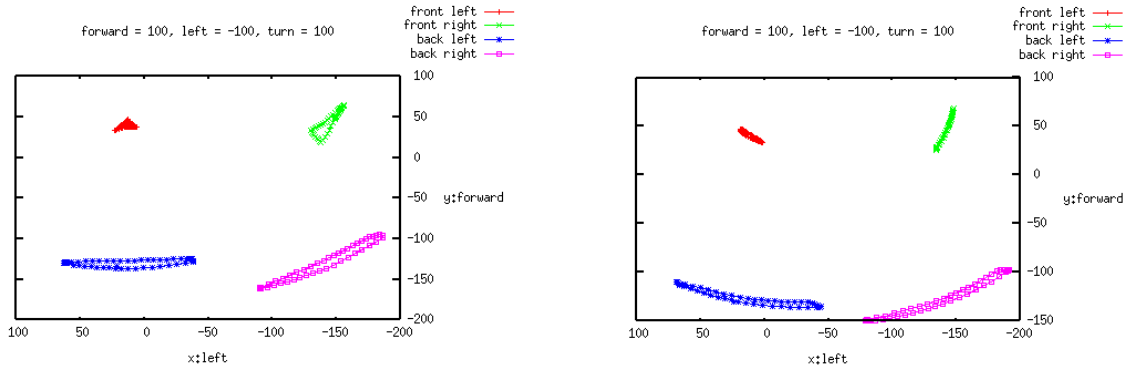
$$C_x = -r \cos \gamma \tag{3.1}$$

$$C_y = r \sin \gamma \tag{3.2}$$

[1] namely forward, left and turn, here written as f, l and t respectively

An implementation can be found in [A.1]. (3.1) and (3.2) give the required values of the turn centre C when used to map the walking locus onto a cylinder. Moving the turn centre like this produces a more fluent and continuous walk. The poor performance associated with large values and the sum of components method is mostly eliminated with this approach, allowing for less clipping and keeping more options for behaviours. Unfortunately the extended flexibility also makes for a less predictable walk, and odometry becomes an issue.

Comparison



(a) Sum of Components.

(b) Change of turn centre.

Figure 3.3: A comparison of sum of components (on the left) and changing turn centre (on the right) with the same walk command.

Figure 3.3 compares the aerial view of the loci of each leg when given a command with a large positive forward, large negative left and large positive turn. This combination moves the turn centre to the front of the robot, so it can circle a point while facing it, for example the ball. Notice how the legs move constructively when changing the turn centre, whilst they work in different directions with the sum of components model.

3.1.4 Problems

Part of the design of the new walk was to introduce a lower body tilt. This has many advantages. For example, the chest IR sensor can be useful over a larger range, as it is not pointing into

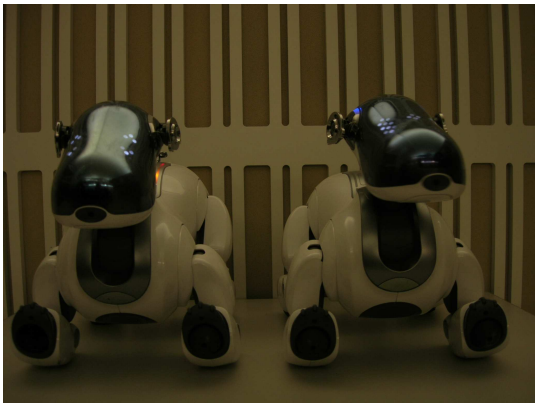
the ground as much. [8] and [5] both discuss the various infra red sensors on the robot in more detail. Also, locating landmarks while grabbed is easier, as the robot's head is flatter.

A comparison of the two stances can be seen in Figure 3.4 and Figure 3.5.

(a) SkellipticalWalk stance.

(b) PolygonWalk stance.

Figure 3.4: Walking stances as seen from the side.



(a) Front view.

(b) Back view.

Figure 3.5: Walking stances as seen from the front and back.

Regrettably it seems that other parts of the system implicitly assumed the SkellipticalWalk stance, which resulted in various other skills performing poorly. A team of SkellipticalWalk robots verses PolygonWalk robots would have TeamPolygonWalk getting to the ball faster than TeamSkellipticalWalk the majority of the time, but then completely lose the advantage by missing the grab or having some other high level skill fail. Overall, this impacted more heavily on game play performance, and there was insufficient time to fix each skill to comply with the new walk.

3.2 Learning

The learning system comes in 4 parts. A behaviour that runs on the dog, a learning engine, a base station to facilitate communication between them, and the interface to the walking engine itself. They are each listed here in bottom up order, although it should be understood that the flow of control moves bottom to top, then back to the bottom again, as per Figure 3.6.

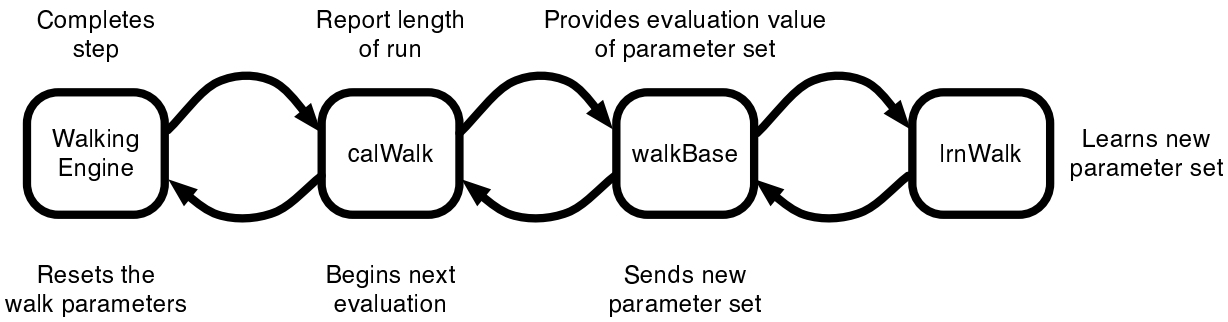


Figure 3.6: Flowchart describing the control flow in the learning system.

3.2.1 Walking Engine

The module that describes the actual walking process. Each walk type must implement a function "read", that takes an io stream containing white space delimited parameters, and sets the appropriate values. They also implement a function "isCompleted", that is used to determine if it is an appropriate time to change step parameters. Together these provide the required interface for learning.

3.2.2 calWalk

This is the behaviour used when walk learning. It is so named as calibration can be performed in a similar method to the learning. The behaviour can be given one or two landmarks depending what type of learning is planned. For straight line learning, either forwards or sidewards, two landmarks are given and the behaviour walks between them, recording time travelled. In the case of turn learning, only one landmark is given and the behaviour spins looking for the landmark,

timing the length of a revolution. In both cases, a number of trials can be taken to constitute a completed run. It is then the total time of the run that is transmitted to the base station.

The main loop for the behaviour is given in A.2.

calWalk is also responsible for instructing the walking engine to change it's current set of walking parameters. It does this at the end of an evaluation, in order to test another point in the search space. calWalk must also ensure the step is complete before changing commands, to promote reproducibility amongst trials.

3.2.3 walkBase

The walkBase runs as a base station, communicating with the robot wirelessly. For every run that calWalk finishes, walkBase records the time and instructs calWalk to continue as appropriate. If walkBase has sufficiently many runs, it sends the median of the run times to the learning algorithm as a fitness for that evaluation. The learning algorithm will then provide a new parameter set to try, If the walkBase has not recorded enough runs, then it sends the same set of walk commands to calWalk, which starts another run of the same evaluation.

3.2.4 lrnWalk

This module is the actual learning algorithm. It uses a simplex downhill based approach. This and other learning algorithms that were considered in previous years, along with their descriptions are discussed in [12]. This year however, to better cope with the massive increase in the size of the parameter set [2], subspace searching was introduced. This approach means that not all dimensions of the full parameter set are searched at once, only a subset are chosen for each "roll downhill". After a given number of evaluations, a new subset is found and the process continues. In the current implementation, dimensions are discouraged from being chosen repeatedly, to broaden the search. Also, dimensions can be forced to be chosen, if they are deemed important to learn in every subspace. See A.3 for the method of choosing a subspace.

[2] SkellipticalWalk has 22 extra parameters, PolygonWalk has 42

3.2.5 Summary

A brief overview of what each module is responsible for:

Walking Engine	Executing Trials
calWalk	Sum several Trials into a Run
walkBase	Take the median of many Runs to return an Evaluation
lrnWalk	Use the Evaluation scores to simplex down hill in different subspaces

3.3 Calibration

Walk calibration, a.k.a. odometry is where we map requested motion to actual motion. We then find the inverse and apply this to all our requested motions so as to better fit reality.

The real goal of calibration is to improve the accuracy of the motion model in the types of motion that are actually used, even if this is at the expense of other types of motion.

3.3.1 Testing an Existing Calibration

The accuracy of a calibration can be tested by disabling the vision updates in the gps module. Without the vision updates, the robot's world model is entirely determined by it's motion model. Starting in a known position on the field and attempting to move until having moved at least some known distance. The distance actually moved can then be measured to determine how close the internal motion model is to reality. The same idea can be applied to turn calibration by turning about a known angle.

3.3.2 Calibrating In the Laboratory

In preparation for the calibration in Bremen, a method was developed in the laboratory.

1. Robot makes a constant number of steps at a particular desired step length.

2. Measure actual length of total distance travelled.

3. Divide total length by number of steps to give each step length.

4. Knowing PG, we can determine speed, since $2PG =$ "number of motion frames per step", so $2 * PG * 8 =$ "number of ms per step", so the speed is $steplength/(2 * PG * 8) mm/ms$.

A script to better automate this process was written, merging the data from several runs to give an amalgamated calibration for any tested speed. Testing several speeds that are actually used, we can fit a piece wise linear function fitting all the test data.

3.3.3 Calibrating at the Competition

After arriving at the venue, the situation was not what I expected. There was a large amount of disorder on the field that meant repeated trials were not easy. Fortunately after testing forward and left movements, the existing calibration was close enough that errors were within noise.

The turn calibration was off however, consistently under turning significantly. Testing [3] multiple dogs and different speeds, they were all under turning by a constant multiple. Correcting this only involved multiplying the given turn amount by the inverse of this under turn amount.

3.4 Conclusion

PolygonWalk has the potential to be a very useful walk. Watching it glide towards the ball while SkellipticalWalk thumps it's way around the field makes it seem promising as a successful replacement. Despite this, when used are a substitute for SkellipticalWalk in an otherwise unmodified team, the result is left to be desired. Without time left to chase and fix the many skills PolygonWalk appeared to break, it was cast aside to perhaps help another day.

[3] As described in 3.3.1

Chapter 4

Tools

There is more to the Robocup codebase than robot code. Many online and offline tools exist to improve productivity. Described in this chapter are a number of tools I either wrote entirely myself or improved upon from existing tools.

4.1 Python Uploader

The behaviours were ported from C++ to python in 2004. The primary reason for switching to an interpreted language was to reduce the development cycle time. Eliminating the necessity to recompile between tests significantly improves work flow, more readily lending to a prototyping style of development. In general this is a fast way to get results that work.

In addition to these advantages, running in an interpreter means that run time changes are a lot easier to make. This eliminates the need to even change the memory stick that is in the robot to test a different behaviour. For this to work, we would need to be able to transmit the new source to the robot, and have it reload the relevant module.

"dog_upload_python" is a script that uses ftp to transfer the relevant behaviour file to the dog. It then uses "robot_commander", as described in 4.2, to reload the python modules.

4.2 Robot Commander

When developing behaviours with the robot's, it is often useful to be able to execute arbitrary code at runtime. This allows for variable inspection, which is very useful for debugging. It also provides the ability to switch behaviours in real time. This is helpful if there are multiple methods to be compared and it is desirable to test them on the same robot, providing consistency of results. Alternatively, there may only be one robot available for testing. The behaviour can be written such that it switches methods on a trigger, preventing the need to change the code between test evaluations. Robot Commander achieves this by sending TCP messages to the robot. Two types of message can be sent.

- The first reloads the Python interpreter. This is useful in combination with the Python Uploader, described in [4.1]. After new modules are uploaded, the interpreter needs to be restarted for the changes to take effect. Reloading the interpreter also recovers elegantly from python exceptions, so in the event of an error, the code can be fixed, the afflicted module uploaded and reloaded for further testing. All without leaving your seat!

- The second executes arbitrary Python code on the robot. Any valid python code segment can be specified as part of the message, and will be executed on the robot. This coupled with telnet provides convenient debugging options. Using the Python/C API in combination with Robot Commander also allows dynamic changes to C code without recompilation.

4.3 Automatic Walk Calibrator

Our system is capable of logging all inputs given to the gps module. This allows for the module to be run offline. Originally this was designed to eliminate the use of several magic constants [1] littered throughout the code, by machine learning them. This is achieved by running the Kalman filter through the logs and measuring the sum of the logarithms of the weights of each gaussian. This gives a fitness function for that world model. This is then run through a simplex downhill algorithm. For more details, see [9].

Using a similar method, we can pass the components of motion for every motion update through an arbitrary linear function, and learn which function gives the best overall accuracy. The assumption is, that the best motion model will give the most consistent world model. Knowing this, means we can invert the function and use that as our calibration, eliminating any hand calculations or measurements. The process is outlined as follows:

1. Record a variety of logs of gps inputs with different behaviours e.g. the dog moving between the goals, or spinning in the middle of the field, or chasing a ball.

2. Feed logs into the gps learner, to learn a system of linear functions;

$$\begin{pmatrix} f_f & f_l & f_t \\ l_f & l_l & l_t \\ t_f & t_l & t_t \end{pmatrix} \begin{pmatrix} fwd_{old} \\ left_{old} \\ turn_{old} \end{pmatrix} = \begin{pmatrix} fwd_{new} \\ left_{new} \\ turn_{new} \end{pmatrix}$$

[1] the world model

3. Parse the output of the gps learner with the walkCorrection utility, which does the required matrix inversion as seen in A.4

4. Insert the resultant code directly into PolygonWalk.cc.

This process, although completely automatic, when it was eventually used the result did not improve on the existing calibration. Even when taking varied logs and combining them in different ways, no calibration was found that improves on hand calibration, so it was abandoned. Running the actuator control module with memory checking utilities unravelled a large number of memory errors. These were mysterious and did not make sense, and as such were never tracked down. This could be one reason the method did not work as intended. Regardless, it should be noted that the version of PolygonWalk.cc as of the end of 2006 no longer supports the insertion of code outputted by walkCorrection.

4.4 Walk Base and Offactuator

Both of these tools required minor tweaks to work with PolygonWalk. In the case of the Walk Base, robot networking code also had to be expanded to facilitate PolygonWalk's larger parameter set.

Chapter 5

Other Research

The nature of development in Robocup is such that it is not really feasible to dedicate yourself solely to one area. When bug fix or even just improving poor performing code, it is important to be flexible and attack the real underlying problem. This is preferable over patching over the issue in whatever piece of the puzzle you happen to be working on at the moment. Consequently, during my time as a rUNSWift member, I have seen a significant portion of the code base and have added to a reasonable amount of it. Included here is a selection of work that does not fall under my expertise, "Gait Optimisation".

5.1 Velocity Interception

A naive way to chase the ball could be to continually walk towards it's current position. This works well with a stationary ball, but that is not what we have in the robocup domain. A better approach would be to move to intercept the ball where it will be by the time we get there. This is made possible since the GPS module provides a more accurate measure of the velocity of the ball than in previous years, thanks to Oleg Sushkov's work [9].

Assuming a maximum robot velocity of V_r and ball velocity of $V_b = (V_x, V_y)$ this minimal intercept point [1] can be found in closed form. At time t the robot can move to any point within a circle of radius $V_r t$, and the ball will have moved by $V_b t$. Thus solving for the intercept point amounts to intersecting a line with a circle as derived in (5.1).

$$\begin{aligned} x &= V_x t + B_x \\ y &= V_y t + B_y \\ x^2 + y^2 &= (V_r t)^2 \\ (V_x t + B_x)^2 + (V_y t + B_y)^2 &= (V_r t)^2 \end{aligned}$$

$$(V_x V_y - V_r^2)t^2 + (V_x B_y + V_y B_x)t + B_x B_y = 0 \tag{5.1}$$

After deriving 5.1 we only need to find the smallest t to give the closest intercept time, which in turn given the intercepted ball position to aim for. Unfortunately the closed form solution is deficient in a number of ways.

This model does not account for any acceleration in the system. Acceleration can be easily modelled in the closed form, but requires an undesirable higher power of t to solve for.

A more critical failing is the inability to realise the movement restrictions on the robot. Robot's movement is much more difficult to accurately model in closed form. Without much thought,

[1] Named (x, y) here

many problems arise, including, the robot's forwards and leftwards movement are different, clipping works non-trivially for differing combinations of forward and left, and the distance metric is actually closer to a queens distance than Euclidean distance, since forwards and leftwards distances are covered simultaneously. Taking these into consideration, it was more practical and flexible to implement as an iterative process A.5, rather than solving in closed form.

5.1.1 Experiment Set Up

A ramp is positioned on the field, at an incline of 15°, the highest point over a goal line and inclined towards the field. The dog is positioned facing the centre of the field with his hind legs on the side line. The ball starts at the top of the ramp, oriented such that the glue ring is in contact with the ground to obtain more consistent rolls between tests.

The ball is released from the top of the ramp. A behaviour then begins chasing the ball when the ball's heading to the dog is less than 35 degrees for more than 3 concurrent frames. The behaviour times the length between the beginning of the chase and when it believes it is within 20cm for more than 3 concurrent frames. Having the dog time it's own run from reproducible points helps reduce human error in measurement.

Comparing velocity interception and straight line chasing this way showed favourable results as can be seen in Table 5.1.

5.2 Goal Keeper

A Goal Keeper's most important role on the team is of course to keep the ball out of his goal. In order to best achieve this, the player is required to find a balance between defence of the goal and attack on the ball. An unchallenged striker may not find it too difficult to score a goal, whilst a striker against a miss positioned Goal Keeper would find it even easier. Presented in this section is a method for positioning, and comments on when not to attack the ball.

Trial	Direct Approach (ms)	Intercept Approach (ms)
1	6703	3965
2	6382	5744
3	6708	4783
4	5874	4718
5	6049	3857
6	7606	4845
7	6609	5709
8	6976	5101
9	6807	4083
10	6070	5689
11	4473	5159
12	6839	5988
13	7114	5235
14	5936	4910
15	6671	5136
$\overline{x}$	6454.47	4994.80
σ	724.64	654.28

Table 5.1: Ball approach results. Average and standard deviation are given by $\overline{x}$ and σ respectively.

5.2.1 Positioning

The perfect positioning policy would be such that at any time, the opposition has an equal chance of scoring a goal on either side of the Goal Keeper. In general this is very difficult to evaluate. In particular, it requires knowing your opponent's strategy, which is very difficult to compute while playing, especially without human intervention.

Defending An Attack

A first approach to approximating the optimal positioning scheme could be to place the Goal Keeper on the interval between the ball and the midpoint of the two goal posts, like in Figure 5.1.

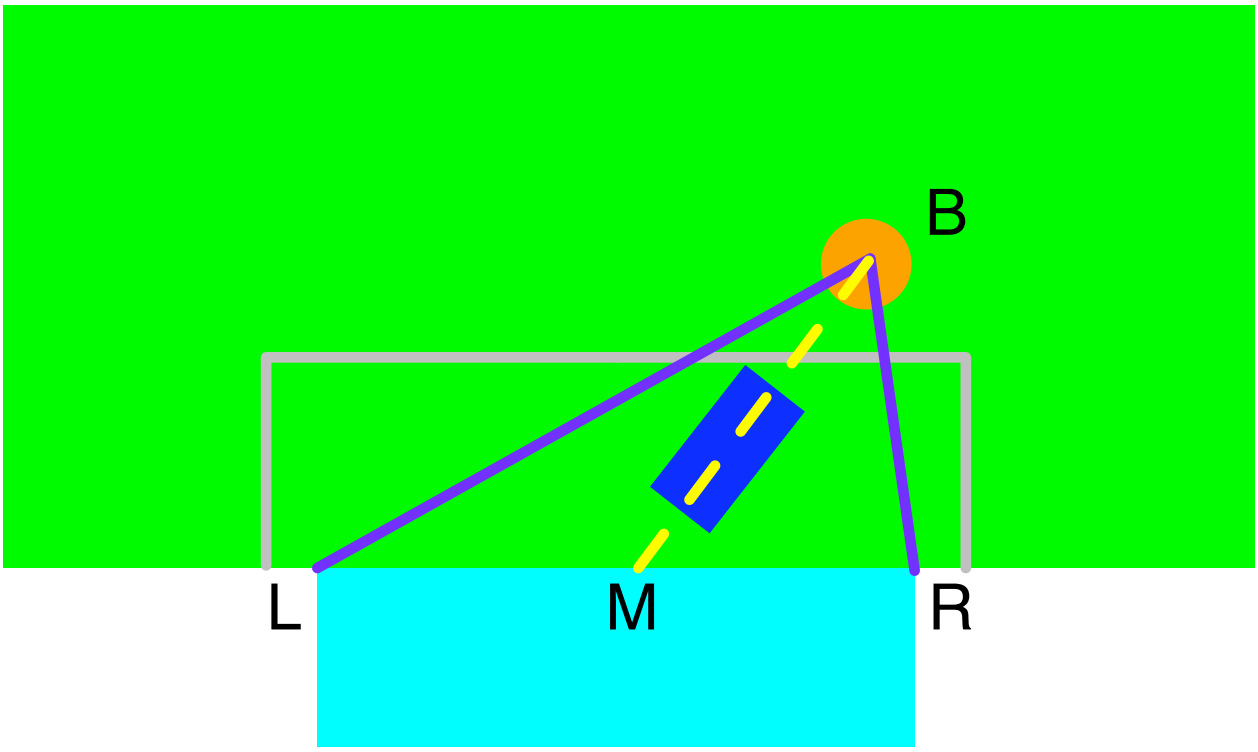


Figure 5.1: Midpoint based Goal Keeper positioning.

Although it seems like this positioning would work well, it does not really capture what it means for there to be an equal chance to score on either side. As you can see in Figure 5.1, the striking player has a much easier shot shooting to the Goal Keeper's right.

A closer approximation would be to place the Goal Keeper on the line that bisects the angle formed by the two goal posts and the ball. This is illustrated in Figure 5.2.

To determine the target position for this method, we require the line PB. We will use the point gradient form of a line, since we already have the point B known and can derive the gradient as $\tan\theta$. Resulting line equation given as equation 5.2.

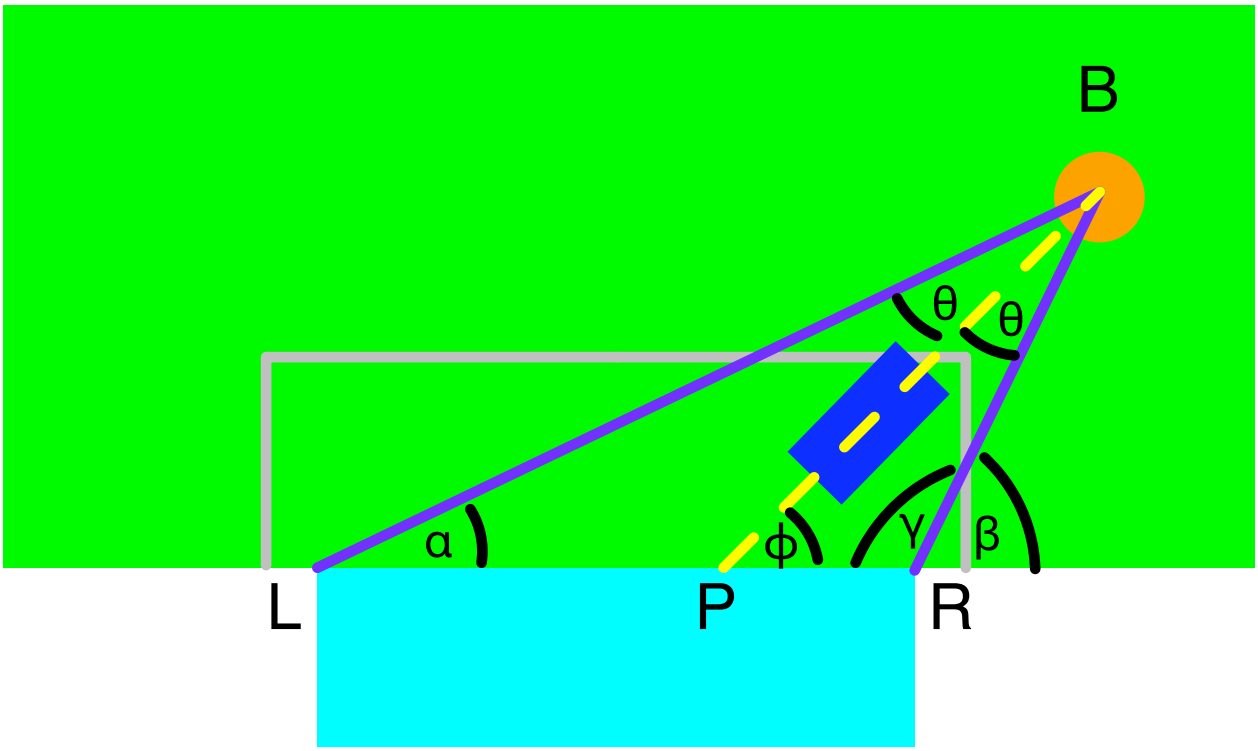


Figure 5.2: Equiangular based Goal Keeper positioning.

$$
\begin{aligned}
\alpha &= \arctan \frac{B_y}{B_x - L_x} \\
\beta &= \arctan \frac{B_y}{B_x - R_x} \\
\gamma &= \pi - \beta \\
\theta &= \frac{\pi - (\alpha + \gamma)}{2} \\
\phi &= \pi - (\theta + \gamma)
\end{aligned}
$$

$$
y - B_y = \tan \phi (x - B_x) \tag{5.2}
$$

Now we can choose an x or a y, and calculate the other using equation 5.2. Usually you would want to control how aggressive the Goal Keeper is, which more directly relates to y than x. So choose a suitable y, such that $0 < y < B_y$, then calculate the corresponding x.

Home position

The previously discussed positioning all assumes you have some idea of where the ball is. If not, as can easily happen when the ball is occluded, there must be a fall back position. The only sensible such position is in line with the centre of the field. Thus the only consideration is where to position in the y direction. The further down field the Goal Keeper is, the more of the goal he can cover. But being closer to the goal means less travelling when the ball moves. Since the Goal Keeper is expected to stay localised very accurately, movement is often discouraged. Any points from the baseline to half way up the goal box are about equally good as home positions. Any further down and your Goal Keeper will spend more time lost than saving goals.

Special Cases

This positioning scheme covers the majority of the time on the field. Certain situations break the continuous positioning however. Explicit conditions for avoiding the goal posts for example, as the Goal Keeper can easily get stuck on them, and never recover. Also, if he finds himself completely lost, there must be a fall back policy that spins looking for a landmark to localise off of.

5.2.2 Attacking

An effective Goal Keeper must not only position himself well, but also clear the ball when it gets too close. Listed here are some guidelines and their respective requirements.

Guideline	Requirement
Do not attack what you can not see.	The Goal Keeper has seen the ball recently.
Do not abandon defence the goal.	The Goal Keeper is close enough to the goal box.
	The ball is close enough to the goal box.
Do not attack your own team member.	No team member has the ball grabbed.

If all such requirements are fulfilled, it is reasonable for the Goal Keeper to approach the ball

in order to clear it from the goal area. It is important to use a kick that will reliably make contact with the ball. The UPenn kick is one of the few rUNSWift kicks that is both reliable and executed outside a grab. This makes it appropriate to use when clearing the ball, as failing a grab too close to the goals would be disastrous. For a more detailed description of the kicks used this year, see [5].

5.3 Python Exception Indicator

When developing in python, it is not uncommon to raise Python exceptions. Before this year, in the event of an error, the excption object would rise up from the cause of the error all the way to the C API level before it is caught. Once an exception occurs at this level, it is blocks further access to the Python interpretter. Instead the exception trace is continuously printed. The behaviours are then in an unstable state. Without further instruction, ActuatorControl usually repeats the last action endlessly. Sometimes python errors could occur at times that immediately crash the robot. In addition to this, the telnet code is buggy, and will lock up and stop printing if it gets flooded. This is exactly the case when the behaviours are printing the python exception once every vision frame. Consequantly, the robot is left acting strangly with no indication of what went wrong.

There are only two python callbacks the C API calls, *processFrame* and *processCommand*. Each of these are now protected, and at the worst, all python exceptions are caught at this level. The python exception is printed once, and a unique indicator fires, alerting the developer that a python error has occured, and that he should check telnet for the nature of the error. See A.6 for reference.

Chapter 6

Evaluation

This chapter provides some numerical justification that PolygonWalk is a smoother, less destructive walk than SkellipticalWalk. Several experiments were carried out to this end. A discussion of the results follows the actual data.

6.1 Data Gathering

It was attempted to show that PolygonWalk is gentler on the joints than SkellipticalWalk is. The joints of the robot can return their "PWM" values. These values are a measure of how worked the joint is. Taking the maximum of the PWM values over all the joints gives an overall indication of how forceful the movement is to the robot.

Data for five types of movement was gathered. Marching on the spot, moving full speed forward, moving full speed left, turning at full speed, and chasing a ball. All tests were performed for both SkellipticalWalk and PolygonWalk on the same robot[1].

The first four tests were conducted using calWalk. For the forward and left results, the goals were used as targets, to give a long run that reduces end effects. Values near a change in walking style were discarded. Ball chasing is far less reproducible, but is presented more as a guide that PWM values in a somewhat more game like situation are also improved.

6.2 Results

Below are line plots for each of the different experiments.

[1] BITS-7ec0

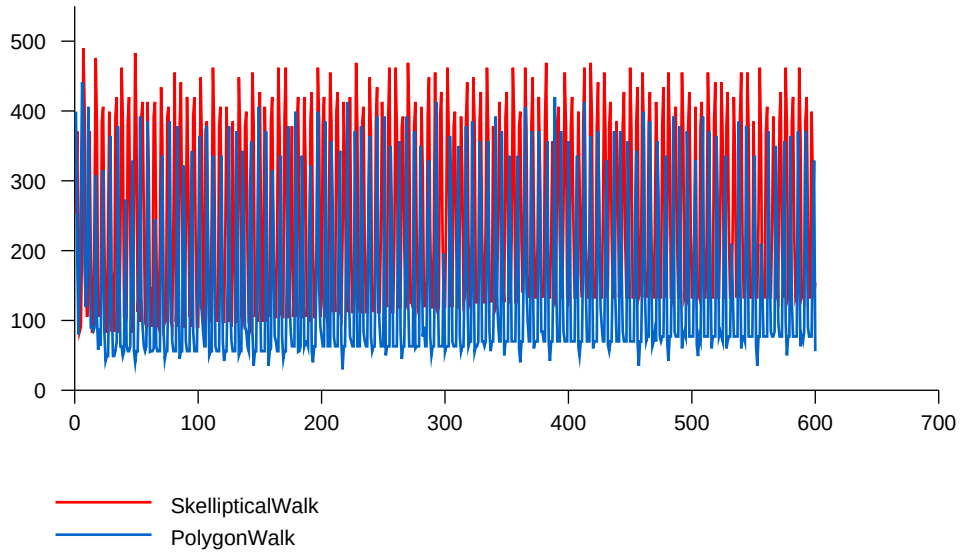


Figure 6.1: Line plot comparing PWM values for SkellipticalWalk and PolygonWalk both marching on the spot.

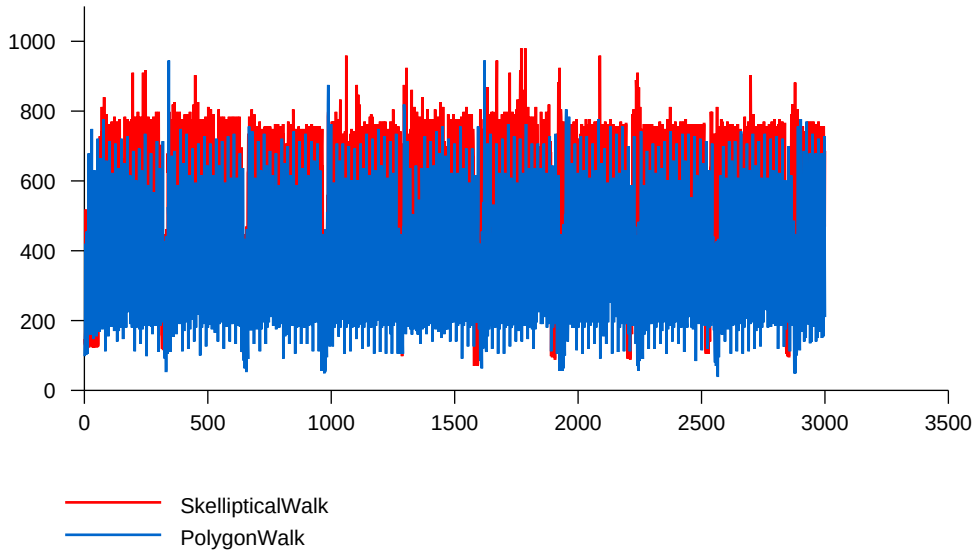


Figure 6.2: Line plot comparing PWM values for SkellipticalWalk and PolygonWalk both walking forwards.

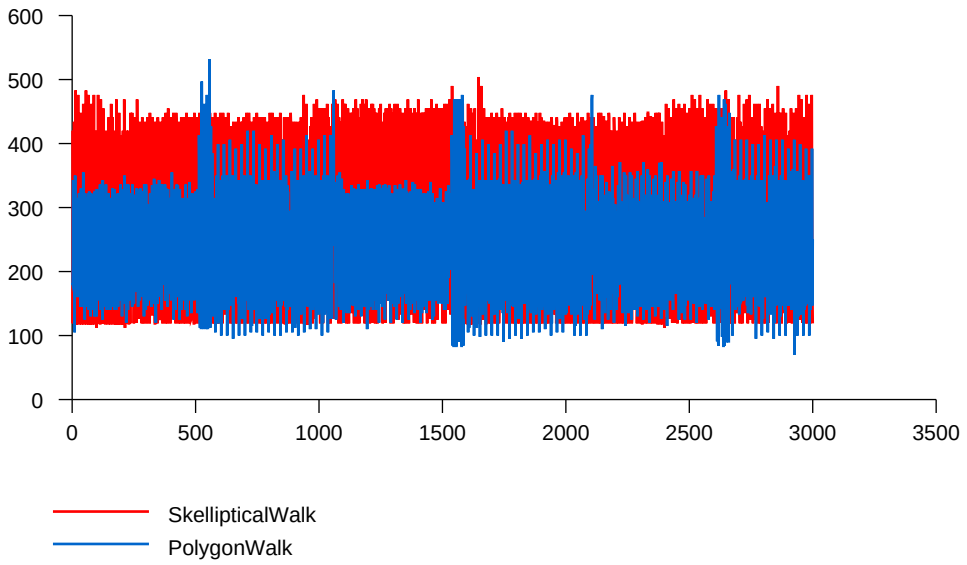


Figure 6.3: Line plot comparing PWM values for SkellipticalWalk and PolygonWalk both walking left.

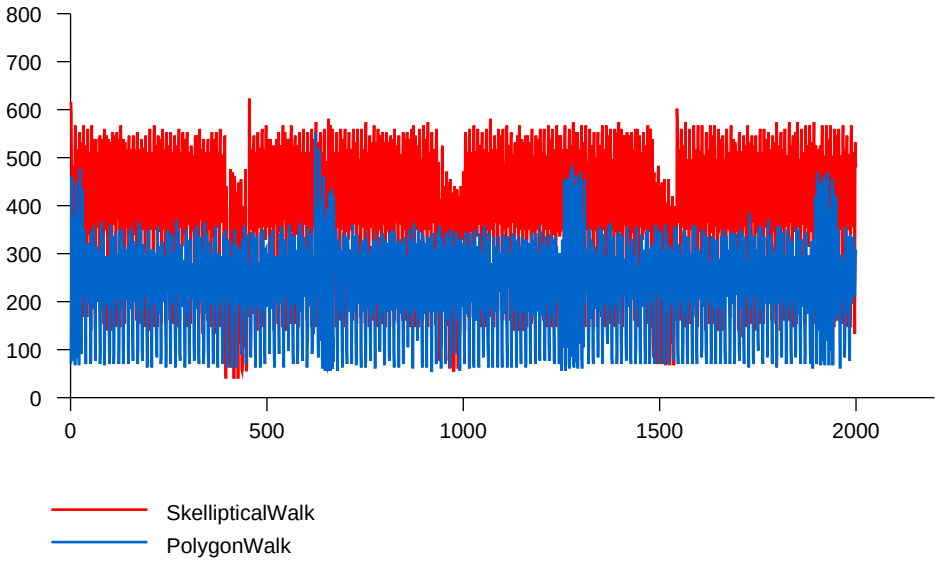


Figure 6.4: Line plot comparing PWM values for SkellipticalWalk and PolygonWalk both turning on the spot.

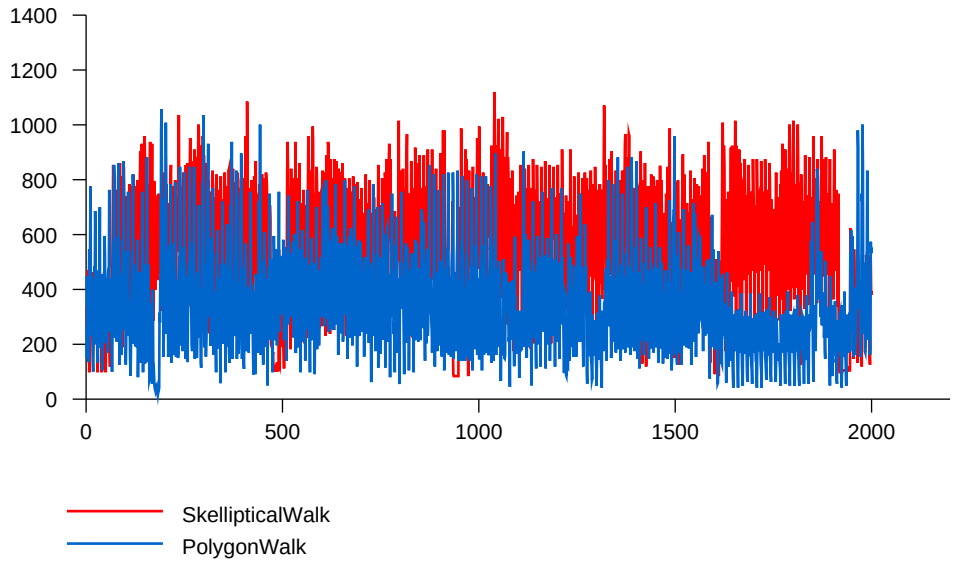


Figure 6.5: Line plot comparing PWM values for SkellipticalWalk and PolygonWalk both chasing a ball.

6.3 Discussion

In each of the tests, PolygonWalk scored lower PWM values than SkellipticalWalk. This shows PolygonWalk has a smaller impact on the joints of the robot, reducing overall wear and tear. This was at no significant loss to straight line speed. PolygonWalk was just as fast as SkellipticalWalk, but quieter and less damaging.

The left results are of particular interest, as the PolygonWalk PWM values consistently alternate between about 300 to about 400. This is due to the nature of the experiment, where at the end of each run, the robot would turn around and walk in the other direction, in this case, walking towards it's right. Since the same locus is used for both directions and the robot is much heavier on one side[2] the resulting walk is quite different in either direction. Despite this, PolygonWalk still beats SkellipticalWalk both ways.

The turn results show equidistant peaks in PolygonWalk and dips in SkellipticalWalk. These represent the part of the trial where the robot is realigning to begin the spin again. The interesting part here is that SkellipticalWalk's spin scores higher PWM when it's spinning compared to the almost marching to realign. The peaks in PolygonWalk are explained because the behaviour falls back onto the SkellipticalWalk to realign. This is a property of the learning, as it must use a known working walk between trials. Notice how the peaks in PolygonWalk equate in value to the dips in SkellipticalWalk.

[2]The battery is quite heavy.

Chapter 7

Conclusion

This report shows that although pushing robots to their breaking point is generally required to get the best performance, the converse is not true. Destructive operation does not imply best use, as we have seen PolygonWalk perform as well as SkellipticalWalk but at a reduced risk.

7.1 Future Work

It is clear that for this walk to be used in a competition, it requires better integration into the existing system. Each skill must be revisited, and individually tuned for the different stance and movement properties.

Further development of the changing turn centre's approach to command interpretation could prove fruitful. In particular, calibrating odometry, and knowing just when it is better to use over the sum of components method.

Code exists to implement joystick control of the robots. The C drivers for the Side Winder joysticks are in the repository, and coupled with the robot_commander infrastructure, developing behaviours that react to the joysticks should be quick and painless.

Bibliography

[1] http://www.smh.com.au/news/technology/aussie-robot-dogs-are-world-champs/2006/06/19/1150569256168.html.

[2] http://www.tzi.de/4legged/pub/Website/Downloads/Rules2006.pdf.

[3] Andrew Owen. The 6th Sense: I see red people (As balls). Technical report, University of New South Wales, September 2006.

[4] Bernhard Hengst and Darren Ibbotson and Son Bao Pham and Claude Sammut. The UNSW United 2000 Sony Legged Robot Software System. Technical report, University of New South Wales, November 2000.

[5] Eric Enyang Huang. The Road to Robocup 2006 - rUNSWift 2006 Behaviors & Vision Optimizations. Technical report, University of New South Wales, September 2006.

[6] Jin Chen and Eric Chung and Ross Edwards and Nathan Wong and Bernhard Hengst and Claude Sammut and William Uther. Rise of the AIBOs III - AIBO Revolutions. Technical report, University of New South Wales, November 2003.

[7] Kim Cuong Pham. COMP3902 Special Project B Report. Technical report, University of New South Wales, September 2004.

[8] Michael Lake. Turning heads at the World Cup & making our mark with field line localisation. Technical report, University of New South Wales, September 2006.

[9] Oleg Sushkov. Robot Localisation Using a Distributed Multi-Modal Kalman Filter, and Friends. Technical report, University of New South Wales, September 2006.

[10] Sony Corporation. *OPEN-R SDK Model Information for ERS-7*, 2004.

[11] Thomas Rofer and Ronnie Brunn and Stefan Czarnetzki and Marc Dassler and Matthias Hebbel and Matthias Joungel and Thorsten Kerkhof and Walter Nistico and Tobias Oberlies and Carsten Rohde and Michael Spranger and Christine Zarges. The German National RoboCup Team. Technical report, 2005.

[12] Weiming Chen. Odometry Calibration and Gait Optimisation. Technical report, University of New South Wales, September 2005.

Appendix A

Code Listings

This appendix contains code segments describing various algorithms referenced in this report. Code as shown here may differ from the versions in the current Robocup repository [rev. 10113] to improve readability. For example debugging statements are removed so as not to distract from the algorithm itself.

A.1 Interpretation of Walk Commands

```
if (absTurnLength > 0.0001 and changeTurnCenter) {
  double turnRadians = turnLength / BODY_RADIUS;
  double dist    = sqrt(absFwdLength * absFwdLength + absLeftLength *
     absLeftLength);
  if (dist > 0.0) {
    double alpha    = (M_PI - turnRadians) / 2;
    double beta     = atan2(leftLength, fwdLength);
    double gamma    = M_PI - alpha - beta;

    double radius   = dist * sin(alpha) / sin(turnRadians);
    double cx       = -radius * cos(gamma);
    double cy       =  radius * sin(gamma);

    setTurnCenter(cx, cy);
  } else {
    setTurnCenter();
  }

  XYZ_Coord fwdPoint  = getLocus(stepFrac, front, FORWARD_LOCUS);
  XYZ_Coord leftPoint = getLocus(stepFrac, front, LEFT_LOCUS);
  XYZ_Coord turnPoint = getLocus(stepFrac, front, TURN_LOCUS);

  XYZ_Coord xyz;
  xyz.x = fwdProportion * fwdPoint.x + leftProportion * leftPoint.x +
     turnProportion * turnPoint.x;
  xyz.y = fwdProportion * fwdPoint.y + leftProportion * leftPoint.y +
     turnProportion * turnPoint.y;
  xyz.z = fwdProportion * fwdPoint.z + leftProportion * leftPoint.z +
     turnProportion * turnPoint.z;
```

```
    // Scale forward to account for turning
    xyz.x *= turnRadians / maxProportions;
    fsh = makeFSHTrn(xyz, left, front);
  } else {
    double fwdProportion_   = fwdLength  / maxFwdLength;
    double leftProportion_  = leftLength / maxLeftLength;
    double turnProportion_  = turnLength / maxTurnLength;
    double totalProportion_ = fwdProportion_ + leftProportion_ +
       turnProportion_;
    if (totalProportion_ == 0.0)
      totalProportion_ = 1;

    // The parameters contributed by the forward locus.
    XYZ_Coord fwdXYZ = getLocus(stepFrac, front, FORWARD_LOCUS);
    fwdXYZ.x *= fwdProportion_;
    fwdXYZ.z *= fwdProportion_;
    FSH_Coord fwdFSH = makeFSHFwd(fwdXYZ, left, front);

    // The parameters contributed by the left locus.
    XYZ_Coord lftXYZ = getLocus(stepFrac, front, LEFT_LOCUS);
    lftXYZ.x *= leftProportion_;
    lftXYZ.z *= leftProportion_;
    FSH_Coord lftFSH = makeFSHLft(lftXYZ, left, front);

    // The parameters contributed by the turn locus.
    XYZ_Coord trnXYZ = getLocus(stepFrac, front, TURN_LOCUS);
    trnXYZ.x *= turnProportion_ / 120; // Converting mm -> radians
    trnXYZ.z *= turnProportion_;
    FSH_Coord trnFSH = makeFSHTrn(trnXYZ, left, front);

    fsh.f = fwdFSH.f + lftFSH.f + trnFSH.f;
    fsh.s = fwdFSH.s + lftFSH.s + trnFSH.s;
```

```
    fsh.h = fwdFSH.h * fwdProportion + lftFSH.h * leftProportion +
        trnFSH.h * turnProportion;
}
```

A.2 Walk Learning Behaviour

```
MAX_LINE_UP_TIME = 2 * Constant.FRAME_RATE
MIN_TRIAL_TIME   = 1 * Constant.FRAME_RATE
lineUpTime       = MAX_LINE_UP_TIME
startTimeStamp   = VisionLink.getCurrentTime()
numStartTrials   = 0
numEndTrials     = 0
def calWalk():
  # 1. initialise
  target = currentMode.target()
  target.frameReset() # update visual info every frame

  # 2. act
  currentMode.look()
  currentMode.walk()

  # 3. time
  global lineUpTime, numStartTrials, numEndTrials
  if lineUpTime > 0:
    # we line up to our target at the end of every run
    lineUpTime -= 1
    currentMode.lineUpToTarget()
  else:
    global isTiming, startTimeStamp
    if isTiming:
      if currentMode.shouldEndTiming():
        endTimeStamp = VisionLink.getCurrentTime()
        elapsed      = hMath.getTimeElapsed(startTimeStamp, endTimeStamp)
        # only end a trial if we haven't already and the last trial lasted
            long enough ago
```

```
      if numEndTrials < numStartTrials and elapsed > MIN_TRIAL_TIME *
          numStartTrials:
        numEndTrials += 1
      else:
        print "time elapsed is too short"
      if numEndTrials >= currentMode.trialsPerRun():
        numStartTrials = 0
        numEndTrials   = 0
        isTiming = False
        print "--------------------------------"
        print currentMode
        print "timestamp:", endTimeStamp
        print "I'm close to", target
        print "time elapsed: ", elapsed, " milli seconds"
        print
        sendWalkInfo(elapsed)
        Action.forceStepComplete()
        currentMode.finishRun()
        lineUpTime = MAX_LINE_UP_TIME
    elif currentMode.shouldStartTiming():
      # record that we ended a trial
      if numStartTrials <= numEndTrials:
        numStartTrials += 1
  else: # if not isTiming:
    if currentMode.shouldStartTiming():
      isTiming = True
      startTimeStamp = VisionLink.getCurrentTime()
```

A.3 Subspace Choice

```
NUM_SAMPLES_PER_SUBSPACE = 50
subspace_samples = 0
current_subspace_flags = []
current_best_parameters = []
global_min_list = []
global_max_list = []

last_flags = [False] * 100
def generateRandomSubpaceFlags(length):
  global last_flags
  subspace_flags = list()

  for i in range(0, length):
    chance = 3
    outOf  = 10
    if last_flags[i]:
      chance = 2 # discourage picking the same flag twice in a row
    subspace_flags.append(random.randint(1, outOf) <= chance)
  # force 0th flag [PG] to be chosen
  subspace_flags[0] = True

  last_flags = subspace_flags[:] # remember this set of flags
  return subspace_flags
```

A.4 Walk Correction

```
#include <iostream>
#include <fstream>
#include "../../robot/share/minimalMatrix.h"

using namespace std;

int main(void) {
  ifstream f("new_inits");
  MVec3 constant;
  MMatrix3 m, n;
  MMatrix34 correction;
  for (unsigned int row = 0; row < 3; row++)
    for (unsigned int col = 0; col < 4; col++)
      if (col < 3)
        f >> m(row, col);
      else
        f >> constant(row, 0);

  n.isInverse(m);

  for (unsigned int row = 0; row < 3; row++)
    for (unsigned int col = 0; col < 4; col++)
      if (col < 3)
        correction(row, col) = n(row, col);
      else
        correction(row, col) = constant(row, 0);

  cout << "double correction[3][4] = {";
  for (unsigned int row = 0; row < 3; row++) {
    if (row > 0)
```

```
      cout << ", ";
    cout << "{";
    for (unsigned int col = 0; col < 4; col++) {
      if (col > 0)
        cout << ", ";
      cout << correction(row, col);
    }
    cout << "}";
  }
  cout << "};" << endl;

  return 0;
}
```

A.5 Ball Interception

```
BALL_FRICTION = 0.97
BALL_FRICTION_4 = BALL_FRICTION ** 4
FRAME_SKIP = 4
FRAME_TO_PREPARE = 10

def walkToMovingBall():
  tx, ty = Global.gpsLocalBall.getPos()
  dx, dy = Global.getOffsetBallVelocity()

  if dx == dy == 0:
    # Velocity was small enough that it was all variance
    return (tx, ty)

  # initialise closest target point to be infinite
  minTarget = (Constant.LARGE_VAL,0,0)

  if ty > 0:
    ySpeed = FWD_SPEED
  else:
    ySpeed = BACK_SPEED

  if tx > 0:
    xSpeed = RIGHT_SPEED
  else:
    xSpeed = LEFT_SPEED

  # try out next 3 seconds
  for i in xrange(0, 3 * Constant.FRAME_RATE, FRAME_SKIP):
    tty = abs(ty) / ySpeed
    ttx = abs(tx) / xSpeed
```

```
    tt = max(tty,ttx)

    td = abs(tt - i)
    # If the robot can reach to the target point comfortably,
    # then go to the target point.
    if tt + FRAME_TO_PREPARE < i:
      return (tx, ty)

    # If this target point has the smallest difference of time for
    # ball and time for robot to reach the point, then consider
    # this target point.
    elif minTarget[0] > td:
      minTarget = (td, tx, ty)

    # updating next possible target point
    tx += dx*FRAME_SKIP
    ty += dy*FRAME_SKIP

    # account for ball friction
    dx *= BALL_FRICTION_4
    dy *= BALL_FRICTION_4

  if minTarget[0] != Constant.LARGE_VAL:
    return (minTarget[1], minTarget[2])

  return Global.gpsLocalBall.getPos()
```

A.6 Python Callbacks

```
# PyError happen, let's let the user know
def handleError():
  hFrameReset.framePreset()
  hFrameReset.frameReset()

  import Indicator
  Indicator.showPythonError()

  Action.standStill()

  hFrameReset.framePostset()


# Protects the function you give it, by catching exceptions and letting
   you know.
def protectFunc(f, *args):
  global hasExceptionHappened
  try:
    # we want exceptions to stick
    if hasExceptionHappened:
      handleError()
    else:
      f(*args)

  except:
    handleError()

    if not hasExceptionHappened:
      hasExceptionHappened = True
      # Let C print the stack trace (But only once!)
```

```
        raise


# This is the top level python that the C layer calls.
def processFrame():
  protectFunc(processFrame_)


def processFrame_():
  VisionLink.startProfile("Behaviour.py")
  profileFunc(hFrameReset.framePreset)
  if not bStopDecideNextAction:

    # If first call to ready state, reset the readyplayer's globals
    kickOffTeam = VisionLink.getKickOffTeam()

    if VisionLink.getTheCurrentMode() == Constant.READYSTATE and
       (Global.state != Constant.READYSTATE or Global.kickOffTeam !=
       kickOffTeam):
      pReady.resetPerform()

    Global.kickOffTeam = kickOffTeam

    profileFunc(hFrameReset.frameReset)

    # Player number detection. The goalie *must* be player one. Freak if
       not.
    if Global.myPlayerNum == 1 and not Global.isGoalieBehaviour:
      raise Exception("This player is NOT a goalie, but you gave it a
         Goalie IP (ie: ##1)")
    elif Global.myPlayerNum != 1 and Global.isGoalieBehaviour:
```

```
    raise Exception("This player is a goalie, you gave it an ip ending
        with \"" + str(Global.myPlayerNum) + "\" which is NOT a Goalie IP
         (ie: ##1)")

  # If it's not playing state, then we should reset grab.
  # Because we don't want to turn off gps vision update.
  if Global.state != Constant.PLAYINGSTATE:
    sGrab.resetPerform()

  # Initial state - just stand there.
  if Global.state == Constant.INITIALSTATE:
    pInitial.DecideNextAction()

  # Ready state - let me move to my correct position.
  elif Global.state == Constant.READYSTATE:
    pReady.DecideNextAction()
    hTeam.sendWirelessInfo()

  # Set state - I can't move my legs, but I can move head to look for
      ball.
  elif Global.state == Constant.SETSTATE:
    pSet.DecideNextAction()
    hTeam.sendWirelessInfo()

  elif Global.state == Constant.FINISHEDSTATE:
    hTeam.sendWirelessInfo()

  else:
    profileFunc(Player.player.DecideNextAction)

debugOnDemand()
Action.hackGain()
```

```
if bStopLegOnly:
  Action.stopLegs()

if bStopMotion:
  Action.standStill()

hFrameReset.framePostset()

VisionLink.stopProfile("Behaviour.py")
```