rUNSWift 2012 Inverse Kinematics, the Build System, and the Python Bridge.

Carl Chatfield (z3255311)
Bachelor of Computer Science

Supervisor: Bernhard Hengst
Assessor: Maurice Pagnucco

October 24, 2012

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

Abstract

This report covers three parts of the rUNSWift infrastructure: a solution to the inverse kinematic problem on the Aldebaran Nao, implementation details of our Python bridge used for developing high level behaviours, and instructions for setting up an environment suitable for cross compiling gentoo packages for the Nao. The chapters covering the inverse kinematics and Python bridge are accompanied by some rudimentary benchmarks comparing their execution time against other implementations. The inverse kinematics section also contains an analysis of the accuracy of our solution. The last section is purely technical.

Contents

Chapter 1

Introduction

The 2012 Robocup Standard Platform League competition was the first to incorporate the Aldebaran Nao v4 robot, a model which differed significantly from its predecessors. Whilst the upgrade brought with it many advantages, including increased processing power and better cameras, taking advantage of these changes required significant adaptations of existing rUNSWift infrastructure.

The Nao runs a specialised flavour of GNU/Linux, opennao, distributed by Aldebaran. Versions of opennao distributed for previous generations of the Nao where based on open embedded, a distribution of GNU/Linux specifically designed for embedded systems. However, beginning with the Nao v4, Aldebaran made the decision to move to a Gentoo based version of opennao citing the need to simplify the process for end users to build third party software packages. The rUNSWift team relies heavily on software packages not included in the opennao distribution that we must build ourselves, but process for setting up an environment suitable for building these packages proved complex. This report documents the process in a step by step fashion for others wishing to create a similar environment.

From the experience of past teams we were also aware of some deficiencies in the existing infrastructure, and the reworking of low level code gave us the opportunity to rectify these. One issue noticed by the previous team was the unacceptable performance overhead of our python bridge that allows high level behaviours, written in python, to communicate with the underlying C code. We have since substituted the previous bridge that was automatically was generated by SWIG with a hand maintained bridge written using `boost::python`. We also provide some benchmarks that compare the overheads of the two bridges in this real life use case.

Another awry section of our code was the inverse kinematics computations. We recognise that a robust bipedal walk on the Aldebaran Nao will be key to our performance in future competitions, and inverse kinematics is a requirement for walk development. However, the iterative solution used until now was not well understood by the team. The convergence and speed of this method has now been well analysed, but we have also moved to and analysed a closed form solution that is both faster and guaranteed to be exact.

Keeping these three areas of infrastructure current, whilst not particularly novel, are key to maintaining the competitiveness of our system.

Chapter 2

Closed Form Inverse Kinematics

2.1 Background

In 2012 there was much interest in developing a new walk model based on a linear inverted pendulum. In such a model, the body sways back and forth as would a normal inverted pendulum, except that hip height is held constant throughout the walk gait. This is achieved by modifying the length of the inverted pendulum, i.e. the extension of the leg, according to function of the pendulum's incline. Doing so requires precise inverse kinematics.

The previous iterative method of solving inverse kinematics on the Nao was not well understood by the team and was accessed through an awkward interface. Instead of specifying a target position and orientation, the value of the hip yaw and knee joints were specified, along with a target forward and sideways translation. The inverse kinematics then calculated the remaining joints such that the foot arrived at the target flat to the ground. Keeping the foot positioned according to the inverse pendulum model would have required pre-computing the knee joint such that the extension was correct.

Speed is another shortcoming of the iterative method. The closed form calculations are nearly an order of magnitude faster than the previous implementation, despite the code for the closed form not being nearly as optimised. Analysis shows that the previous implementation may have been over conservative by performing a more than required number of iterations and hence unnecessarily slow, however this analysis was not available to the previous teams.

The replacement closed form implementation aims to be fast whilst providing a simpler interface. The replacement also allows the foot's target roll and pitch to be specified in addition to the yaw.

2.2 Considerations

Any unambiguous definition of a target in 3-dimensional space must contain at least 6 parameters. For example, we use a 3-dimensional position vector plus three Euler angles, roll, pitch and yaw. This means that for any kinematic chain to reach an arbitrary target it must contain at least 6 degrees of freedom.

A degree of freedom is defined as an independent parameter defining the configuration of a system; in a kinematic chain one can think about these in terms of reachable targets. Each joint in the kinematic chain adds a degree of freedom if, assuming all joints are optimally placed, its addition

increases the range of reachable targets. Because the range of reachable targets increases, the introduced joint must have added a parameter to the system that is independent of all others. Optimally placed means that moving the joint somewhere else in the chain whilst maintaining the total chain length cannot increase the range of reachable targets.

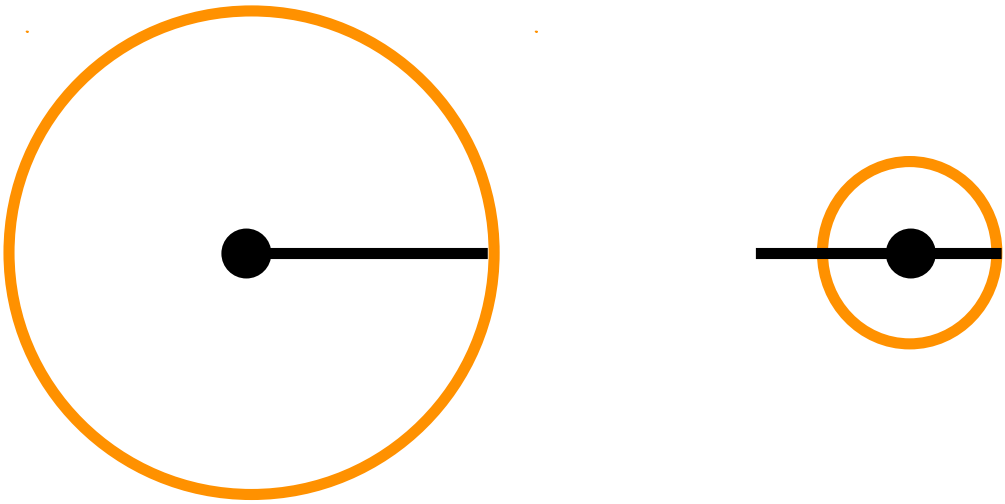


Figure 2.1: An optimally placed joint (left) at the chain origin allows the foot to reach a large circle of targets. A suboptimal placement (right) in the middle of the chain results in a smaller range of possible targets.

For example, if operating in 2 dimensions a single link chain of unit length and no joints has one possible target relative to the origin. Adding a joint at the origin allows a circle of targets at unit distance to be reached from the origin at a single orientation. Adding a joint at the end of the chain will allow the same targets to be reached but at any orientation. Finally, a joint half way along the chain will allow any target within the unit circle to be reached at any orientation. Each joint increased the range of reachable targets and therefore contributed a degree of freedom.

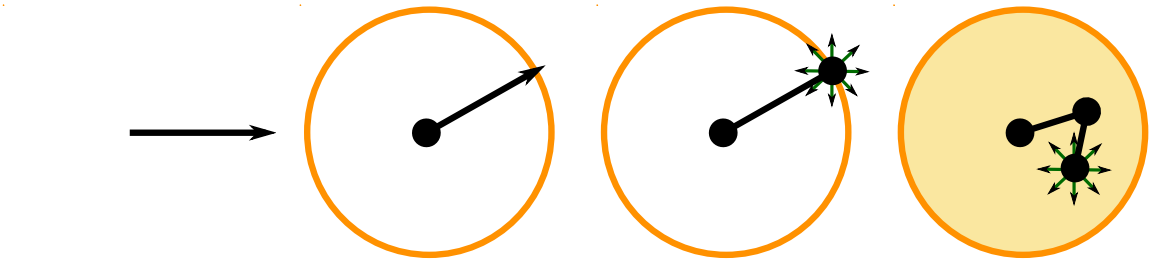


Figure 2.2: Progression of reachable targets as joints are added to the chain.

A target definition in two dimensions requires only 3 parameters, a two dimensional position vector plus an angle. Therefore, adding more joints to the chain cannot increase the number of free parameters.

The leg of the Aldebaran Nao has 6 joints and hence the required number of freedoms to reach an arbitrary target, and the joint placements are almost optimal. The upper and lower leg links are not exactly equal, so the range of possible targets could be slightly increased by moving the knee joint such that they are equal. The ankle joints would also be moved to the tip of the chain eliminating the third foot link entirely.

Assuming optimality however, it is also clear that each joint in the leg chain increases the range of reachable targets. Any two of the hip joints plus the knee joint allow the chain to reach any point in the sphere around the hip of radius equal to the total length of the leg. The remaining three joints each parameterise the orientation of the foot.

The range of targets reachable by the foot of the actual robot is a 6-dimensional space and somewhat difficult to visualise, but if we consider a 3-dimensional slice where all orientations are reachable, the reachable space will be be a hollow sphere of radius the length of the chain minus twice the foot. The radius of the hollow inside the sphere is the length of the foot. The reasoning is that the hollow sphere is obtained by taking the sphere of targets reachable by the ankle, and then subtracting all points within foot length of the sphere's surface and origin. This is not formally derived here. If assumed to be correct however, it follows that for every target within the hollow sphere a solution exists for the leg joints such that chain arrives at the target with an arbitrary orientation. In practice, the range of reachable targets is also restricted by the rotation limits of the joints.

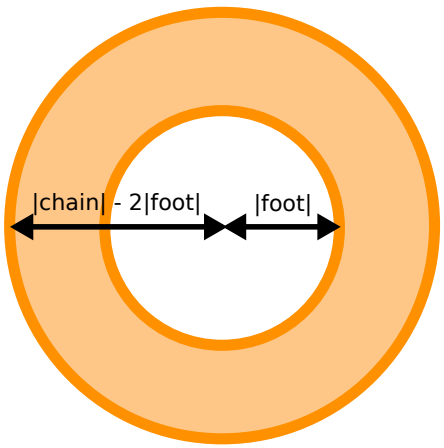


Figure 2.3: A two dimensional cross section of the reachable sphere where all orientations are reachable.

However, the first joint in the chain, the hip yaw joint, is common to both legs. The chain from the left foot to the right therefore effectively has 11 degrees of freedom, making reaching two arbitrarily defined targets for both feet impossible even if both targets fall within the above range. We must therefore allow error in the solution. This error is allowed to manifest in the target yaw as the foot will remain flat to the target regardless of the error.

2.3 Previous Works

The previous iterative implementation used by rUNSWift was inspired by the method described in the 2008 Northern Bites team report [5], which itself is based on a paper from UC San Diego [3]. It works by computing the gradient of steepest descent in the error function and then iteratively moving towards a solution. The analysis in section 2.6 provides detailed graphs showing the convergence of this iterative processes. However, the 2009 Northern Bites [6] report notes that they have since moved to a closed form solution provided by UPennalizers. The UPennalizers team reports make mention of inverse kinematics however do not describe their implementation [2].

BHuman, the victors of the 2009, 2010 and 2011 competition, published a closed form solution in their 2009 team report [7]. This solution computes the leg chain twice for a given target. The first computation exactly computes all joints for each leg chain. However, as mentioned the hip yaw joint is common to both legs and the hip rotation in each leg's solution must be equal. However, when calculating the chains individually there is no guarantee this will be the case. In the second computation, the hip yaw from each solution is averaged together and each chain recalculated with the hip yaw fixed at this average. As noted in the considerations section 2.2, this restriction may introduce error into the target foot yaw. In their report, BHuman refer to this error in terms of a virtual foot yaw joint that, if present, would compensate for the introduced error.

As inverse kinematics is fundamental to any walk gait development, it is likely that other teams implement variations of both closed form and iterative solutions.

2.4 Nao Leg Kinematic Chain Overview

Six joints comprise each leg chain, namely the *hip yaw*, *hip roll*, *hip pitch*, *knee pitch*, *ankle roll* and *ankle pitch*. To compute the position of the foot using rotational and translational matrices, the rotations must be applied in the order they are listed above. The three links in the chain will be referred to *thigh*, *shin* and *foot*.

The hip yaw, roll and pitch joints are situated such that they all rotate about axes that intersect at a single point, greatly simplifying the calculations. The same is true for the ankle roll and pitch joints. Roll joints move the foot along the coronal (yz-plane), and pitch joints move the foot along the sagittal plane (the xz-plane). These two classes of joints operate along orthogonal planes. The hip yaw joint is oriented at 45 degrees on the coronal plane, pointing outwards from the torso. Its rotation affects not only the foot's yaw, but also the foot's roll and pitch.

Customarily, angles about an axis are wound counter clockwise around a positive vector, i.e. use a right hand winding rule. If considering the Nao's left leg and assuming no hip yaw rotation, then the roll and pitch joints rotate about a positive unit vector along the x and y axes respectively. The hip yaw rotates the rest of the chain about a unit vector pointing outwards from the torso at a 45 degree decline. A zero radian rotation about each joint results in the Nao standing directly upright. This zero angle acts as the frame of reference for all other angles.

2.5 Closed Form Solution

The closed form solution is computed in five stages.

Subtraction of foot bone
: Because the target position and orientation of the foot is known in advance, the foot bone can be subtracted from the foot target to produce an intermediate target for the ankle. The chain from the hip to the ankle only contains two bones, making the calculations much simpler.

Coordinate system rotation about the hip yaw joint
: In the absence of hip yaw rotation, all pitch joints translate the foot along the x-axis, and all roll joints translate the foot along the y-axis. If hip yaw is introduced, the movement of the other joints are no longer aligned to these axes. The coordinate system must be rotated such that it is realigned.

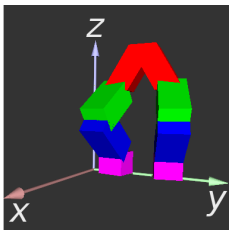


Figure 2.4: Nao leg chains, with Nao standing facing forward.

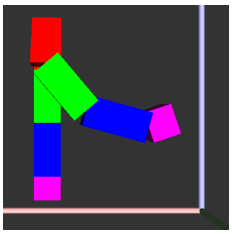


Figure 2.5: Positive pitch values move the leg behind the Nao.

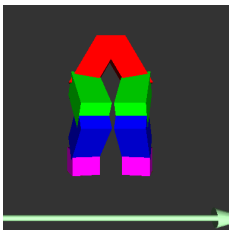


Figure 2.6: Positive hip yaw moves the legs inwards.

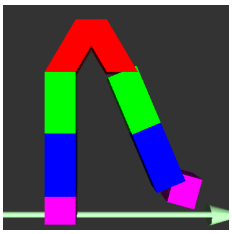


Figure 2.7: Positive roll values move the leg outwards.

Coordinate system rotation about the hip roll joint

The y component of the ankle target is known, and in the remaining chain to the ankle only the hip roll joint can translate the ankle along the y-axis. Therefore it is trivial to calculate the hip roll. Once known, the coordinate system can be again rotated to leave only a two dimensional problem.

Calculation of pitch joints

The remaining hip and knee pitch values can be calculated using trigonometry.

Ankle roll and pitch computation

Once the ankle is known to arrive at its target, the ankle roll and pitch joints must be calculated such that the foot has the correct orientation.

The coordinate system rotations each simplify the problem to one that is easier to solve. For the purpose of explanation, it is easier to first consider cases where hip roll and hip yaw are absent. The steps will therefore be introduced in reverse order.

2.5.1 Subtraction of foot bone

The final orientation of the foot is known at the beginning of the computation, and therefore the foot bone can be subtracted from the kinematic chain to yield an intermediary target for the ankle. First, a vector $\boldsymbol{v}_{\text{foot}}$ is calculated which represents the components of the foot bone in 3-dimensional space. This is then subtracted from foot target $\boldsymbol{t}_{\text{foot}}$ to produce the ankle target $\boldsymbol{t}_{\text{ankle}}$.

$$\boldsymbol{v}_{\text{foot}} = \text{foot} * \begin{pmatrix} -\cos(\text{target}_{\text{roll}}) * \sin(\text{target}_{\text{pitch}}) \\ \sin(\text{target}_{\text{pitch}}) \\ -\cos(\text{target}_{\text{roll}}) * \cos(\text{target}_{\text{pitch}}) \end{pmatrix}$$

$$\boldsymbol{t}_{\text{ankle}} = \boldsymbol{t}_{\text{foot}} - \boldsymbol{v}_{\text{foot}}$$

If the ankle reaches $\boldsymbol{t}_{\text{ankle}}$, and all joints in the kinematic chain are computed such that the foot is in the correct orientation, then the foot too will reach its target, $\boldsymbol{t}_{\text{foot}}$. All future mentions of the target, unless explicitly stated otherwise, will refer to the intermediate ankle target. [1]

2.5.2 Restricted xz-planar Motion

Allowing motion in only a single plane reduces the problem to two dimensions and makes it solvable using only basic trigonometry.

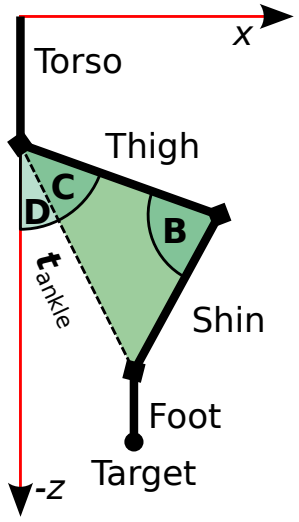


Figure 2.8: Side View of the Leg.

After removing the foot bone, the remaining chain consists of only the thigh and shin bones. Along with the target vector $\boldsymbol{t}_{\text{ankle}}$, these links form a triangle with all lengths known and hence the triangle's angles can also be computed. Angle B corresponds to the required knee pitch, and the required hip pitch is simply the sum of angles C and D. The joints on the robot use the reference system described in section 2.4, and hence the angles require a simple conversion.

$$|\boldsymbol{t}_{\text{ankle}}| = \sqrt{x_{\text{targ}}^2 + z_{\text{targ}}^2}$$

$$B = \text{acos}\left(\frac{\text{shin}^2 - |\boldsymbol{t}_{\text{ankle}}|^2 - \text{thigh}^2}{-2 * \text{thigh} * |\boldsymbol{t}_{\text{ankle}}|}\right)$$

$$C = \text{acos}\left(|\boldsymbol{t}_{\text{ankle}}|^2 - \frac{\text{shin}^2 - \text{thigh}^2}{-2 * \text{thigh} * \text{shin}}\right)$$

$$D = \text{acos}\left(\frac{x_{\text{targ}}}{h}\right)$$

$$\text{hipPitch} = -(C + D)$$

$$\text{kneePitch} = -(B + \pi)$$

The final step is to calculate the ankle pitch such that the foot arrives at the correct orientation. The process to do so will be described later.

[1] The observant reader may recall that our solution allows error into the final foot yaw. Fortunately, target yaw is absent from the equations computing $\boldsymbol{t}_{\text{foot}}$.

2.5.3 Introducing Hip Roll

Hip roll allows the ankle to travel along the y-axis and adds a third dimension to the problem. However, the y-component can be removed by rotating the initial reference frame about the hip roll joint to a new frame where the y axis is orthogonal to the target vector. The extent of the rotation is equal to the required hip roll.

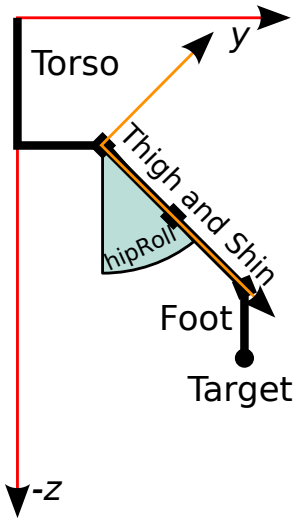


Figure 2.9: Front View of the Leg.

As the y component of the ankle target is known, and in the chain to the ankle only the hip roll can translate the ankle along the y-axis, the hip roll must be the arctangent of the right angle triangle formed by the y and z components of the target.

$$\text{hipRoll} = \text{atan2}(y_{\text{targ}}, -z_{\text{targ}})$$

The target vector is now rotated in the opposite direction, removing the y component.

$$\boldsymbol{t'}_{\text{ankle}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(-\text{hipRoll}) & -\sin(-\text{hipRoll}) \\ 0 & \sin(-\text{hipRoll}) & \cos(-\text{hipRoll}) \end{pmatrix} * \boldsymbol{t}_{\text{ankle}}$$

Instead of using a costly matrix multiplication, $\boldsymbol{t'}_{\text{ankle}}$ can be computed directly using trigonometry.

$$\boldsymbol{t'}_{\text{ankle}} = \begin{pmatrix} \boldsymbol{t}_x \\ 0 \\ \boldsymbol{t}_z / \cos(\text{hipRoll}) \end{pmatrix}$$

Using the new target vector as input to the two dimensional solution, the hip and knee pitch joints can be calculated as before.

2.5.4 Introducing Hip Yaw

The unusual orientation of the hip yaw joint makes calculating the kinematic chain awkward. We therefore want to perform another coordinate system rotation that realigns the pitch and roll joints to the x and y axes, but this requires knowing the final hip yaw value in advance.

BHuman [8] provide an exact solution for calculating the kinematic chain for a single foot to a target, but as mentioned in the considerations section 2.2 the required hip yaw rotation for each leg may differ. In the BHuman solution, the hip yaw rotation from each chain is averaged together, and then a second solution for each leg is computed with the hip yaw fixed at this averaged value. The key is that in the second computation, the final hip yaw rotation is known in advance and therefore can be compensated for by a change of coordinate system.

Computing the chain twice is a substantial overhead. A simpler approach is to assume that the target foot yaw and hip yaw are equal, and the analysis section 2.6 shows that this is a reasonable approximation when target x is small. For larger values of x, the error in the foot yaw becomes substantial. Error does not change substantially with changes to target y or z. We do not currently compensate for this error, but the analysis section discusses this error in detail. Recall that the last step of the inverse kinematics calculation is to compute the ankle joints such that the foot is flat to the target plane; the error only appears in the foot's orientation on the plane.

As with the hip roll, rotating the coordinate frame appropriately will remove the hip yaw rotation from the problem. Because the rotation takes place about a 45 degree vector, there is no concise matrix notation. We instead denote the rotation about the hip yaw axis vector $\boldsymbol{a}$ using angle axis notation.

$$\begin{aligned}
\text{hipYaw} &= \text{target foot yaw} \\
\boldsymbol{a} &= \begin{pmatrix} 0 \\ \sqrt{2}/2 \\ -\sqrt{2}/2 \end{pmatrix} \\
\boldsymbol{t'}_{\text{ankle}} &= \text{AngleAxisRotation(-hipYaw}, \boldsymbol{a}) * \boldsymbol{t}_{\text{ankle}}
\end{aligned}$$

To convert an angle and axis to a familiar rotation matrix, the below expansion of AngleAxisRotation$(\theta, \boldsymbol{a})$ can be used. However, it is worth noting that internally our BLAS library, libEigen, uses quaternions internally to represent the rotations.

$$\begin{pmatrix}
\boldsymbol{a}_x^2(1-\cos\theta)+\cos\theta & \boldsymbol{a}_x\boldsymbol{a}_y(1-\cos\theta)-\boldsymbol{a}_z\sin\theta & \boldsymbol{a}_x\boldsymbol{a}_z(1-\cos\theta)+\boldsymbol{a}_y\sin\theta \\
\boldsymbol{a}_y\boldsymbol{a}_x(1-\cos\theta)+\boldsymbol{a}_z\sin\theta & \boldsymbol{a}_y^2(1-\cos\theta)+\cos\theta & \boldsymbol{a}_y\boldsymbol{a}_z(1-\cos\theta)-\boldsymbol{a}_x\sin\theta \\
\boldsymbol{a}_z\boldsymbol{a}_x(1-\cos\theta)-\boldsymbol{a}_y\sin\theta & \boldsymbol{a}_z\boldsymbol{a}_y(1-\cos\theta)+\boldsymbol{a}_x\sin\theta & \boldsymbol{a}_z^2(1-\cos\theta)+\cos\theta
\end{pmatrix}$$

Once the hip rotation has been compensated for the previous joint values can be solved for as before, this time using $\boldsymbol{t'}_{\text{ankle}}$ as input to the equations.

2.5.5 Calculating Ankle Pitch and Roll

The kinematic chain is now divided into two parts, the first contains the previously calculated hip and knee joints, whilst the second contains the as yet unknown ankle pitch and roll joints. Call these two chains *leg* and *ankle*.

The previous calculations ensure that the ankle at the end of the leg chain arrives at the ankle target. The orientation of the ankle, represented by the matrix L, can be calculated by multiplying together all the rotations of the leg chain. We again use angle axis notation.

$$
\begin{aligned}
L_{\text{hipYaw}} &= \text{AngleAxisRotation}(\text{hipYaw}, (0, \sqrt{2}/2, \sqrt{2}/2)) \\
L_{\text{hipRoll}} &= \text{AngleAxisRotation}(\text{hipRoll}, (1, 0, 0)) \\
L_{\text{hipPitch}} &= \text{AngleAxisRotation}(\text{hipPitch}, (0, 1, 0)) \\
L_{\text{kneePitch}} &= \text{AngleAxisRotation}(\text{kneePitch}, (0, 1, 0)) \\
L &= L_{\text{hipYaw}} * L_{\text{hipRoll}} * L_{\text{hipPitch}} * L_{\text{kneePitch}}
\end{aligned}
$$

Similarly, a matrix T representing the target foot orientation is also calculated by composing the target Euler angles.

$$
\begin{aligned}
T_{\text{pitch}} &= \text{AngleAxisRotation}(\text{target pitch}, (0, 1, 0)) \\
T_{\text{roll}} &= \text{AngleAxisRotation}(\text{target roll}, (1, 0, 0)) \\
T_{\text{yaw}} &= \text{AngleAxisRotation}(\text{target yaw}, (0, 0, -1)) \\
T &= T_{\text{pitch}} * T_{\text{roll}} * T_{\text{yaw}}
\end{aligned}
$$

The ankle rotation matrix A must fill the gap in the chain, rotating L to T. In other words it must satisfy the below matrix equation.

$$T = L * A$$

Matrix A can be solved for, and the roll and pitch components of the rotation can be separated out [4] from the matrix. The caveat is that the third Euler angle, yaw, is unaccounted for. As previously explained, we accept this error in yaw as a consequence of our approximation of the shared hip joint.

$$
\begin{aligned}
A &= L^{-1} * T \\
\text{ankleRoll} &= -\text{asin}(A_{12}) \\
\text{anklePitch} &= \text{atan2}(A_{02}, A_{22}) \\
\text{error} &= \text{atan2}(A_{10}, A_{11})
\end{aligned}
$$

The ankle pitch and roll joints complete the leg chain.

2.5.6 Notes on Source Code Implementation

The source code in appendix B.1 implements our algorithm, however it makes some optimisations.

- The computation of the target rotation matrix does not include the yaw component as it does not affect the final result.

- The foot vector subtracted from the target is derived from the third column of the target rotation matrix instead of computing it using sines and cosines.
- When computing the ankle pitch and roll, the inverse rotation matrix is calculated by applying the leg rotations in the opposite directions and reverse order. This is equivalent to applying the rotations as usual to construct a rotation matrix and then inverting it.

2.6 Analysis

Three inverse kinematics implementations have been analysed for sake of comparison. Our previously described solution and the solution taken directly from the BHuman source code will be compared as is.

The previous rUNSWift iterative implementation required modification to be fairly compared. It calculated the chain in three steps. First the solution was calculated assuming no hip yaw using closed form equations similar to those presented in section 2.5.2. The hip yaw is then set to a value specified by the invoker introducing error which is iteratively removed. Finally, the ankle joints are computed such that the foot is flat to the ground.

The closed form equations from the first stage take as input a desired knee rotation instead of the 3-dimensional target vector used as input to the other two implementations. These equations have been replaced with ones that accept a target vector and computes the required knee pitch joints.

2.6.1 Yaw Error

All three implementations calculate the ankle joints such that the foot is guaranteed to be flat to the target. Therefore, error can only appear in the target yaw. All three implementations also use a predetermined hip yaw and then calculate the rest of the chain. They differ in how they predetermine the value.

BHuman first solve the kinematic chain exactly for each leg, and then average the two solution's hip yaw values together. The rUNSWift implementations, both old and new, require the user to specify the hip yaw in advance. Whilst in the old implementation this was by design, we acknowledge it as an error in the solution.

The invoker of the inverse kinematic computation is probably most concerned with error between the target and actual foot yaw, however to correct for the error in the foot yaw we must first understand the error in the hip yaw. As can be seen from the two dimensional plots, even when x and y translation is absent from the target the hip yaw error function is not a simple sinusoid.

Translation along the y or z axis does not drastically affect the error, however translation along the x axis does. At the extremes of the Nao's range of motion, the difference between the specified and required hip yaw can be up to 90 degrees. This difference actually puts the target outside the Nao's range of motion, it is only the specified hip yaw that is within range. Fortunately, the actual error in the foot yaw is only about 45 degrees at the extremes.

2.6.1.1 Compensation

Compensating exactly for the hip yaw error requires calculating it exactly, which is an expensive overhead. An alternative approach is to try approximate the error. One such empirically found

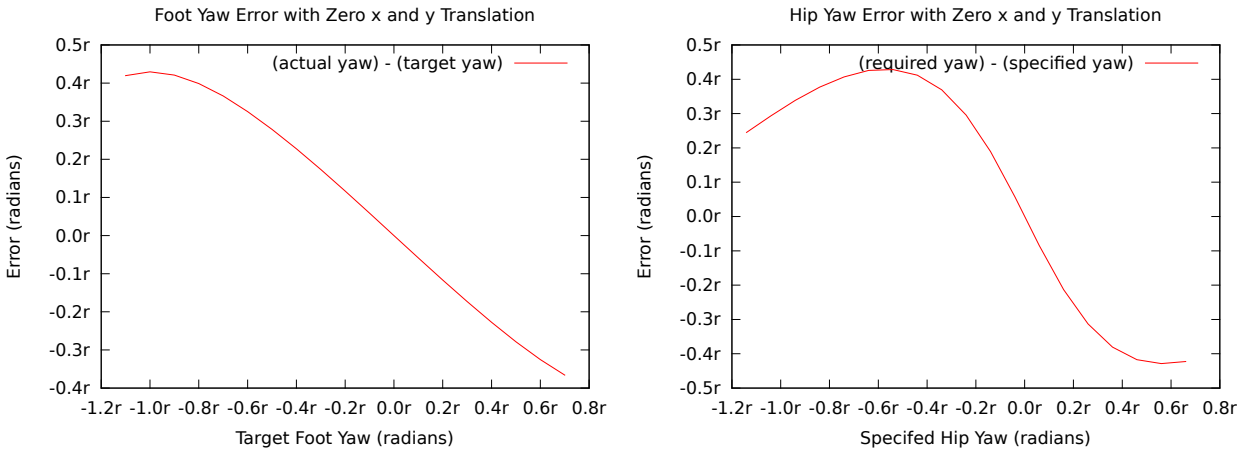


Figure 2.10: Yaw error when the foot is directly below the hip. The left plot shows the error between the target and actual foot yaw. The right plot shows the error between the hip yaw specified by our solution and the correct hip yaw required to reach the users target.

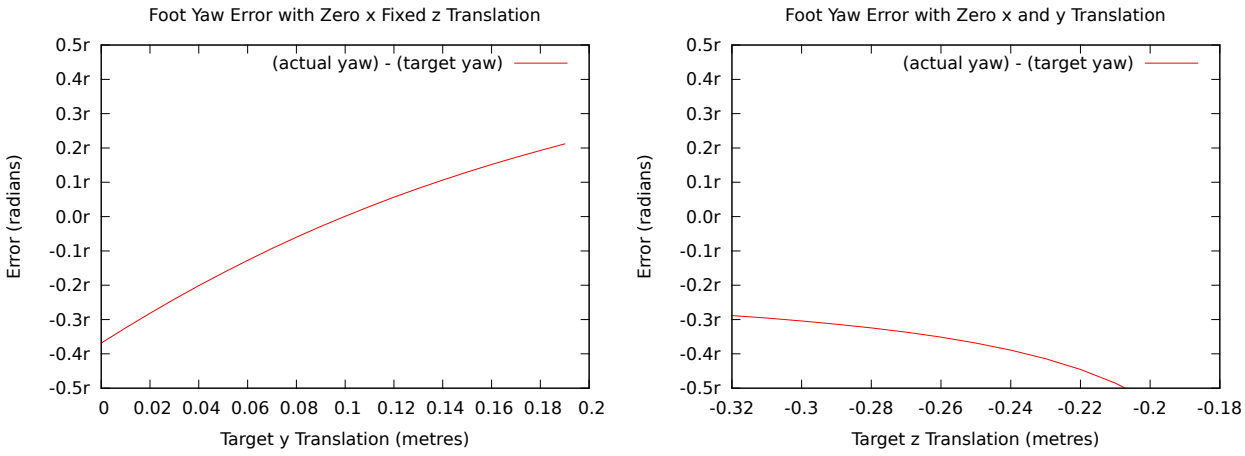


Figure 2.11: Yaw error with respect to y (left) and z (right) translation with target yaw fixed at 45 degrees. 45 degrees is where the error is most prevalent.

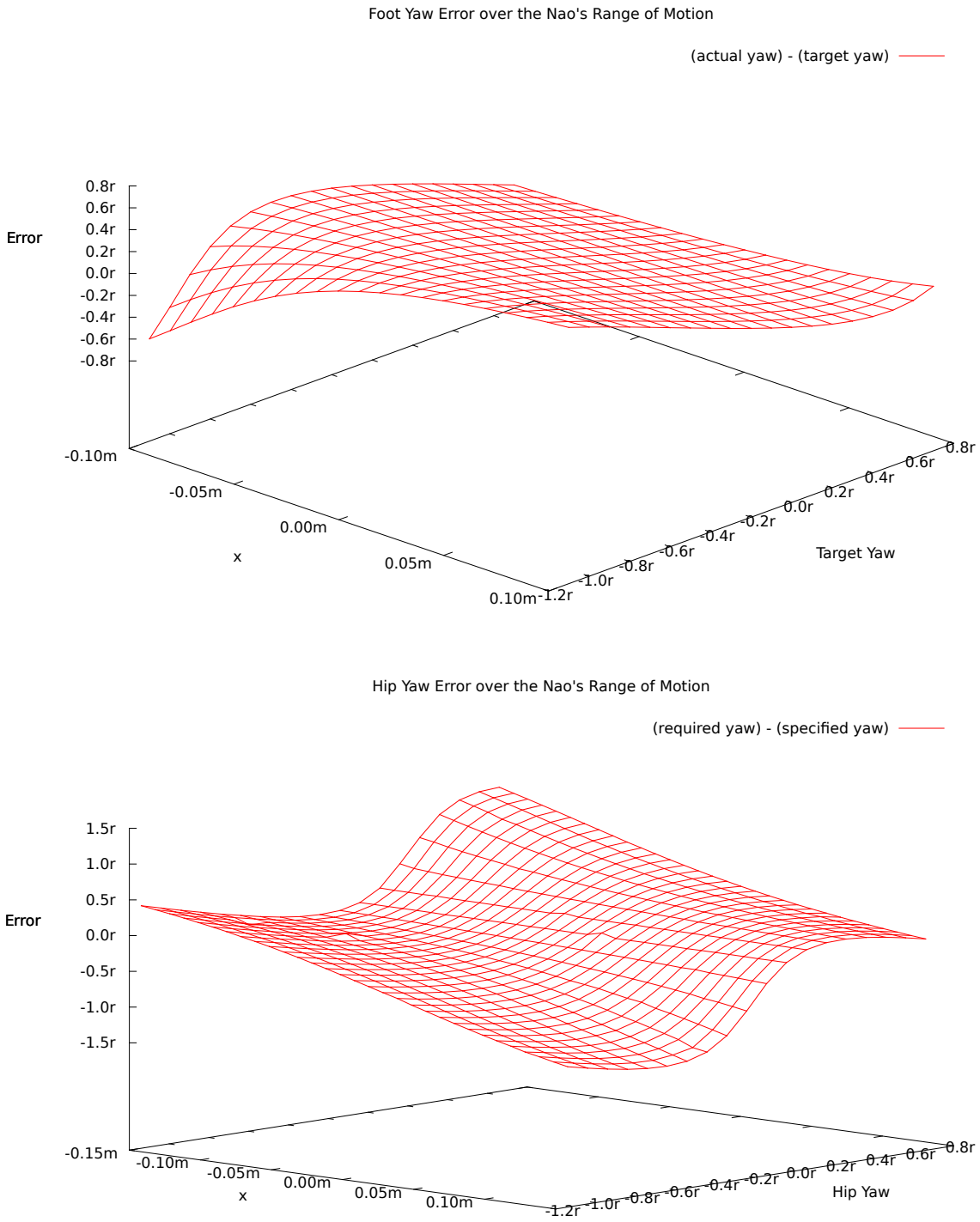
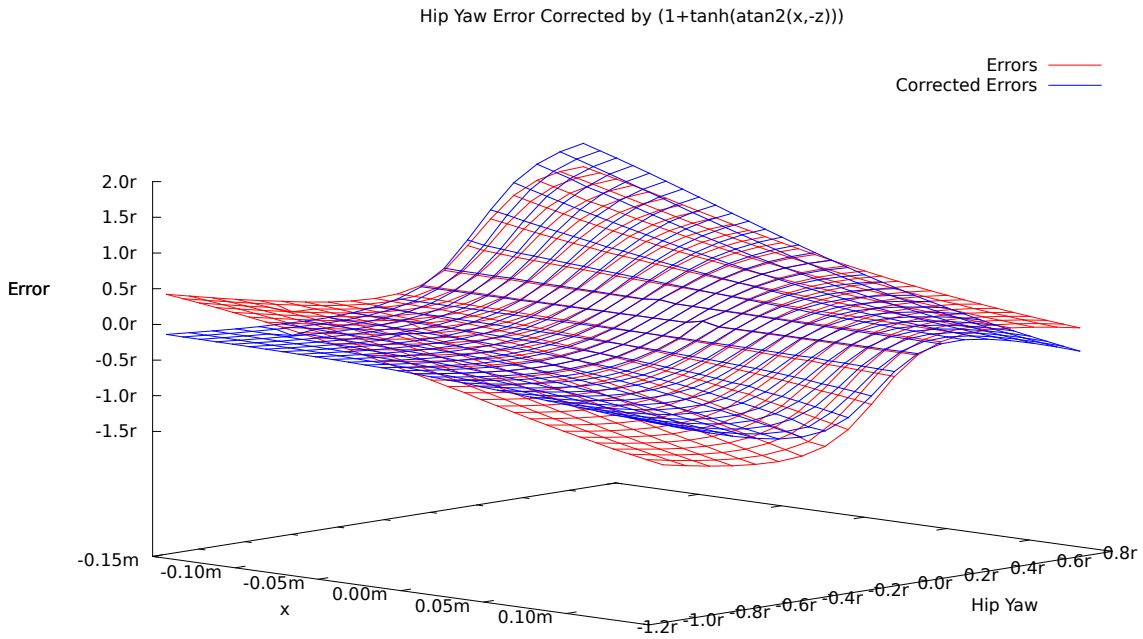


Figure 2.12: Yaw error with respect to target yaw and x translation. The top plot shows the error between the target and actual foot yaw. The bottom plot shows the error between the hip yaw specified by our solution and the correct hip yaw required to reach the users target.

approximation is to multiply the requested hip yaw by $1 + \tanh(\text{atan2}(x, z))$. As the plot shows, this actually works surprising well when the target yaw is less than zero. A negative rotation is an outwards rotation of the leg, which is the more common type of action. When the target hip yaw is greater than zero there is a slight degradation. However, the two trigonometric are in themselves quite expensive to compute.

Another approach is to fit a polynomial surface to the errors and calculate the required correction from the surface's parameters. This is very fast as it requires only a few floating point multiplications which are much faster than the trigonometric function calls. We attempted to fit polynomial curves of several degrees to the hip error surface, however we found the fitted surfaces to be somewhat bumpy. This resulted in a solution where monotonically increasing the target yaw is not guaranteed to monotonically increase the actual foot yaw, which may come as a surprise to the invoker. See the appendix A.2 for fit surfaces of higher degree polynomials.

At present, neither of these compensations have been incorporated into our solution. The old implementation also required the invoker to specify the hip yaw directly, so currently our implementation is compatible. For now there is little motivation to correct the error.



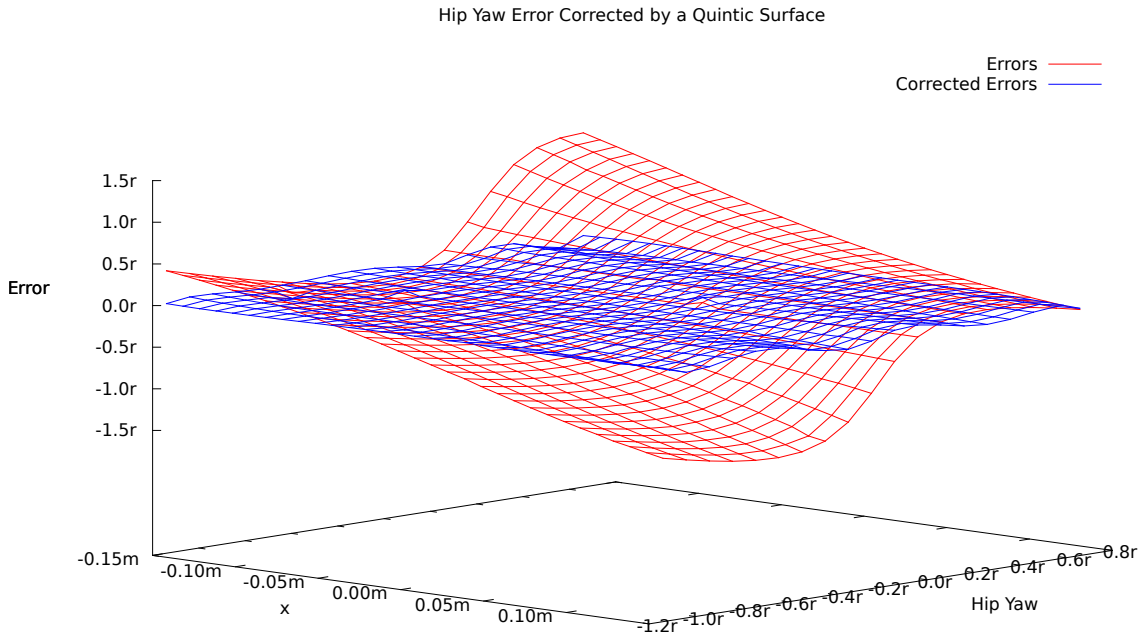


Figure 2.13: Error surface corrected by multiplying the hip yaw by $1 + \tanh(\text{atan2}(x, -z))$ (top) and subtracting a quintic surface(bottom)

2.6.2 Positional Error

Both the BHuman and our closed form kinematics calculates the leg chain such that the foot arrives at the target position vector exactly. The previous iterative method is not exact, but our analysis shows that the iterations converge quickly to an acceptable solution.

2.6.2.1 Convergence of iterative method

The major speed bottleneck of the old implementation is the iterative step. The below graphs show the convergence of the error function, a simple Euclidean distance, over a reasonable portion of the Nao's range of motion.

Each individual graph shows the error surface in metres with the hip yaw fixed at a certain value. The surface is plotted over x and y target translations with $x \in [-0.1\text{m}, 0.1\text{m}]$ and $y \in [-0.05\text{m}, 0.15\text{m}]$. The hip joint is located 0.05m left of the torso, so these ranges represent 0.1m forwards, backwards, left and right from the hip. The z translation is fixed at -0.25m which is a common value. Change in the z translation does not substantially affect the error. Appendix contains graphs over a wider domain of hip yaw values.

It can be seen that in most cases after 4 iterations the error converges. In fact, in most of the graphs the 4 iteration and 5 iteration curves cannot be distinguished. The actual implementation uses 6 iterations which this analysis shows to be very conservative. The 6 iteration curve has been omitted from the below graphs as it only adds to the noisiness.

2.6.3 Speed Comparison

The running time required for each method was calculated by timing one million computations with the per process high precision clock. In theory, the exact number of clock cycles can be computed by multiplying the average run time by the clock frequency of the processor, however it is acknowledged that this process is not exact. All measurements were taken on the Nao v4.

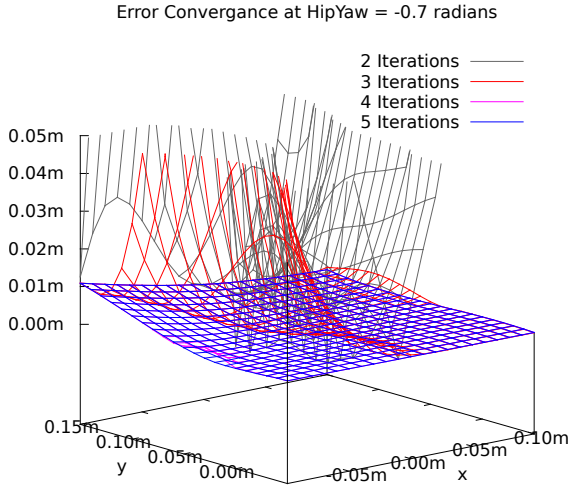
Implementation	μseconds	Clock Cycles	Speed up
Closed Form	7.695	12312	1.00
Iterative 4 iterations	43.462	69539	5.65
Iterative 6 iterations	50.905	81448	6.62
BHuman	13.399	21438	1.78

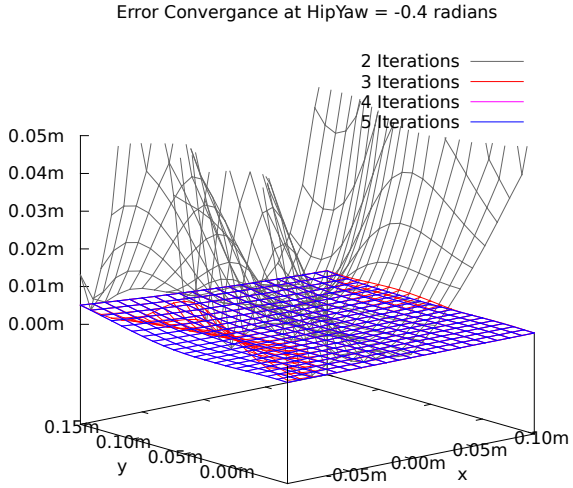
Table 2.1: Average run time comparison.

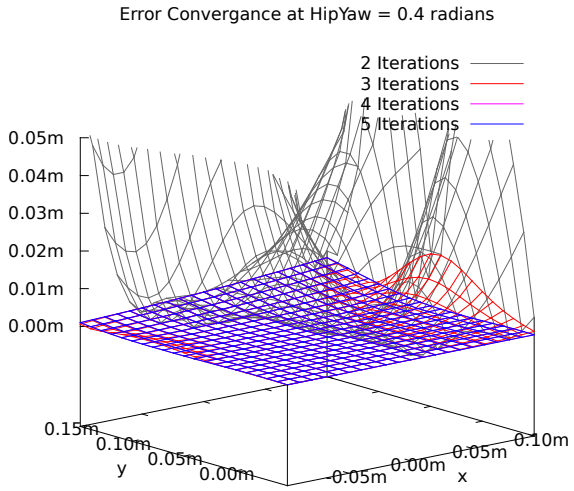
It must be remembered that the BHuman implementation computes each leg chain twice, and this overhead is reflected in the above table. The iterative method, on the other hand, is substantially slower.

The inverse kinematics are computed once per cycle, or 100 times per second. Therefore the iterative solution consumed 0.5% of all CPU time, including CPU time consumed by the operating system. This is by no means prohibitive, however it does rule out search based motion planners. These work by searching the leg chain space for a series of valid intermediate states between the foot's current and target position, and then committing to one of the intermediate targets. Each intermediate chain considered by the search must be evaluated, and even 8 evaluations would cause the iterative method to consume 4% of all CPU time. This is unacceptable. The faster closed form

inverse kinematics can make these 8 evaluations in roughly the same time as the iterative solution makes 1 evaluation, opening the door for such motion planners.







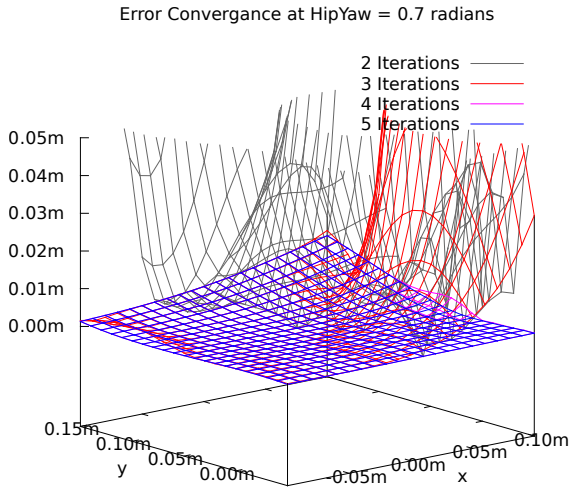


Table 2.2: Convergence of the error over successive iterations. Each graph shows the convergence at a fixed hip yaw value.

Chapter 3

Python Bridge

High level rUNSWift behaviours that run above the core infrastructure are implemented using python 2. The 2010 python bridge was manually implemented using the CPython API; however, the need to reference count python objects and manually write wrappers for each new piece of exposed data caused this approach to become finicky and unsustainable. The following 2011 team automated the wrapping process using SWIG, but unfortunately the generated code introduced intolerable performance overheads. In 2012 we have moved to `boost::python` as it provides a compromise between performance and the verbosity of interface definitions, which are themselves written in plain C++.

3.1 Organisation

The bridge is divided into two components: `wrappers` and `converters`. `boost::python` is capable of automatically generating wrappers for all elementary C++ types, and automatic generation can be extended to non-templated classes using the `boost::python` API. All wrappers that are automatically generated in this manner are stored in the `wrappers` directory.

Certain data types cannot be automatically converted to python due to limitations of either C++ or `boost::python` itself. Instead, `boost::python` exposes an API for adding user defined converters alongside those automatically generated. These converters are stored in the `conversions` directory.

3.2 Converters

Converters are static functions that convert C++ types to python objects using the CPython API. We provide converters for several types defined by libeigen, the library we use for matrix, vector, and BLAS operations. The matrix and vector types are templated and the internal data arrays are not exposed, making wrapping difficult. Instead, our converters produce python tuples containing the original floating point values.

```
struct Point_to_python
{
   static PyObject *convert(const Point &p)
   {
      return
         boost::python::incref(
            boost::python::make_tuple(p.x(), p.y()).ptr()
         );
   }
};
```

Listing 3.1: A simple wrapper around our Point class

We also require a way to convert C style arrays into a python accessible objects. The `boost::python` documentation recommends using STL vectors in place of C style arrays, however in our case the data is statically allocated on a memory mapped shared page. Although possible, controlling where STL vectors allocate memory is difficult. We also have C style arrays in other parts of the code that need to be exposed, so we chose to implement our own converter.

The C type system defines an array in terms of both its element type and length, i.e. a `int[2]`, `float[2]` and `float[3]` all require a different converter. C++ templates are used to generate all array converters from a single piece of code. The resulting `PyObject` satisfies the python mapping interface, meaning the underlying elements of the array can be accessed using python's `array[i]` indexing syntax.

3.3 Wrappers

The process of defining wrappers that result in automatically generated interfaces is relatively straightforward and well documented in the `boost::python` documentation. The only caveat is that if an exposed class member is fetched via a converter, an explicit getter function is needed. Various helper methods for creating these getters exist.

```
class_<BallInfo>("BallInfo")
   .def_readonly("rr"          , &BallInfo::rr          )
   .def_readonly("radius"      , &BallInfo::radius      )
   .add_property("imageCoords" , make_getter(&BallInfo::imageCoords,
       return_value_policy<return_by_value>()))
   .def_readonly("visionVar"   , &BallInfo::visionVar   );
```

Listing 3.2: A simple wrapper. Note that `imageCoords` is fetched via a converter and hence requires an explicit getter.

3.4 The `robot` Module

Python accesses the underlying infrastructure via the `robot` module. This is a C++ module automatically generated by `boost::python` and exported into the interpreter's namespace from the `PythonSkillAdapter`. The module itself is defined using the `BOOST_PYTHON_MODULE` macro which generates the special initialisation function `initrobot` called by the python runtime. During the initialisation, all converters and wrappers must be registered to `boost::python`. Because the wrappers rely on the converters, the converters are registered first.

```
BOOST_PYTHON_MODULE(robot)
{
    register_python_converters();

    #include "wrappers/AbsCoord_wrap.cpp"
    #include "wrappers/ActionCommand_wrap.cpp"
    /* ... */
```

Listing 3.3: The beginning of the robot module definition.

3.5 Interpreter initialisation

The `PythonSkill::startPython` function is responsible for initialising the interpreter. After the interpreter is started through the CPython `PyInitialize` API call, the `robot` module must be loaded into context. In most documentation, python modules are compiled into shared objects and loaded via python `import` statements. Because we require direct access to the module from the C++ code, we chose to compile the module into the main executable and must call the `initrobot` function explicitly. Finally, we append the directories containing our skills and helpers to python's `sys.path` list. This allows the interpreter to find our code.

3.6 Behaviour Invocation

All information about the current state of the robot is passed to the python behaviours in a single object called the `Blackboard`. The behaviour responsible for deciding the robot's next action is then invoked from the C++ code. Rather than allowing the python code to directly control the robot, the C++ side receives back another python object called a `BehaviourRequest` that describes the desired action. The C++ code interprets the request and complies accordingly.

3.7 Error handling

All python errors are captured and handled by the C++ code. During our development cycle, the error handler puts the robot into a safe stand state and waits for the python code on the robot to be modified. After detecting a modification, the interpreter is restored to a clean state and the behaviours are restarted. During competition, the handler immediately restarts the behaviours on exceptions.

3.8 Performance

The experience of the 2011 rUNSWift team was that SWIG was simply too slow on the old Nao v3 robots. Complex behaviours could take up to 15 milliseconds to run, which is half of the target tick execution time. With the upgraded processor on the Nao v4, it is possible that the old SWIG bridge may have been satisfactory; however, as the histogram of behaviour tick execution times shows `boost::python` is substantially faster.

The histogram was obtained by porting the 2012 Python skills to be compatible with the automatically generated SWIG interface. The otherwise identical skills were both instrumented for about

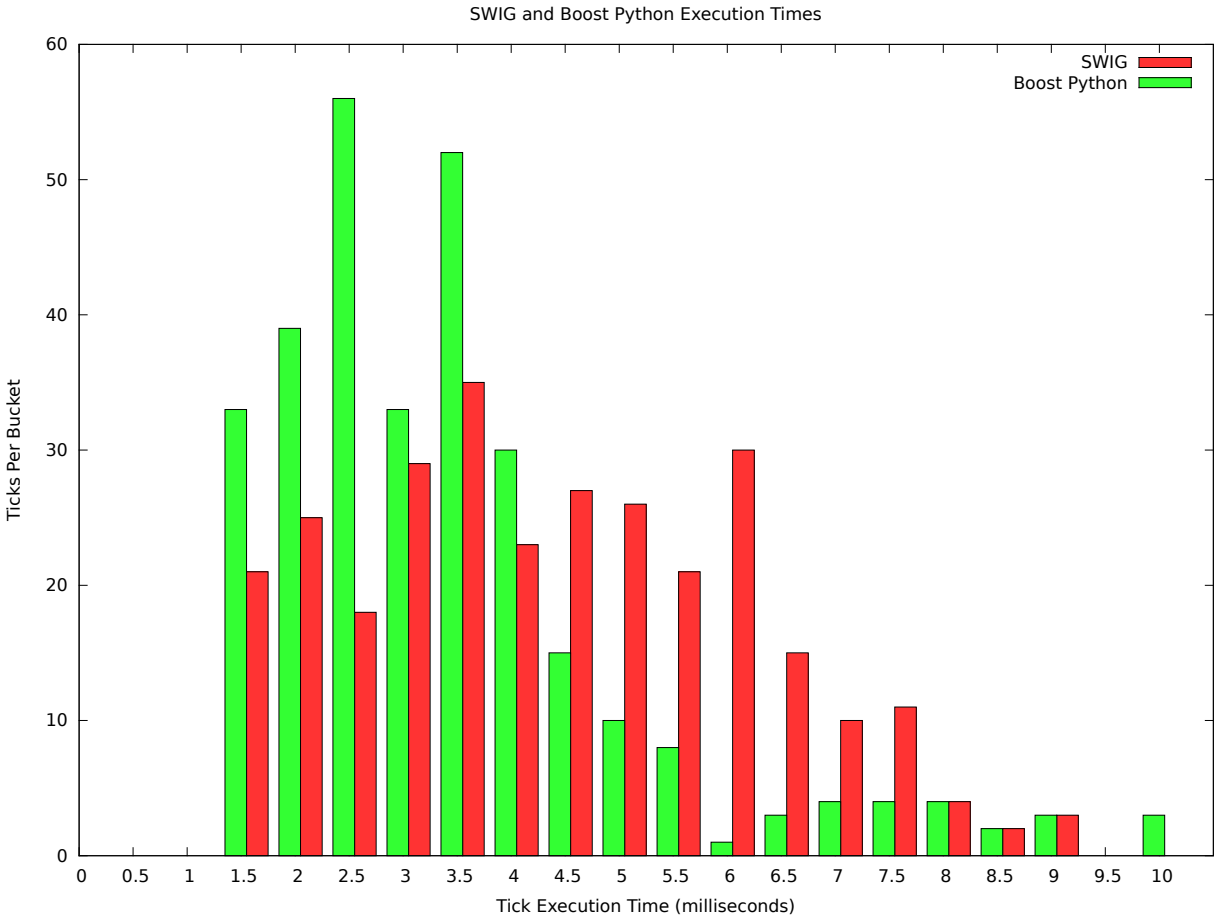


Figure 3.1: Histogram of 300 behaviour execution times using both the SWIG and `boost::python` python bridges. Results are grouped into 0.5 millisecond buckets; faster execution times will result in a shift towards the left of the graph.

10 seconds with the robot walking up to the ball, locating the goal, and then shooting. From each set of measurements 300 have been placed into 0.5 millisecond buckets and plotted next to each other for comparison.

Execution time of each tick is far from a random variable and is highly dependent on the executed code path. The behaviours are represented as a tree of state machines, and often a state machine will simply emit a precomputed action. Transitions between states are more costly and involve some form of pre-computation for later ticks. As an example, on transition into the head scan skill a series of way points are precomputed. Later ticks simply move the head between the way points.

Because of this variance in execution time, there is no distinct bell curve present in the data sets. It is clear however that the SWIG execution time is considerably longer. Beyond the 6 millisecond mark the `boost::python` tick times level off at about 3 per bucket. These are likely the result of long running operating system interrupts servicing the hardware.

Our target perception tick time is 30 milliseconds, and with our current utilisation of the new Nao v4 processor even the longer running SWIG ticks would be unproblematic. Even so future teams will probably appreciate the extra elbow room.

Chapter 4

Cross Compiling Packages

During development rUNSWswift relies on the ability to load the robot's core functionality into our debug tool as a shared library. We are thus forced to use the same tool-chain for compiling code that runs both on and off the robot. However, the standard opennao distribution does not include all the libraries we require, and therefore we build our own. For debugging memory issues, we also compile a valgrind binary compatible with the version of libc on the robot.

4.1 Installation

Our build environment resides in a chroot that mirrors the structure of the opennao image installed on the robots. Anything built inside the chroot is binary compatible with what runs on the robot.

4.1.1 Base Image

Aldebaran provides a modified Gentoo Stage 3 archive that matches what is distributed in the opennao images. It can be found under the `Software/All Downloads/VERSION/NAO OS/NAO OS Sources/` directory of the password protected `http://users.aldebaran-robotics.com/` site. The archive released March 1, 2011 erroneously includes system directories, however this may have been fixed. The first step is to extract the archive to a new directory that will become the chroot.

```
> mkdir gentoo && cd gentoo
> tar -xjvf $IMAGE --exclude ./sys --exclude ./proc --exclude ./dev
```

The next step is to acquire a portage tree snapshot. Unlike previous iterations of opennao that where based on an Open Embedded distribution, the Nao v4 is based on Gentoo which uses a rolling release system managed by a tool called portage. The robots, however, do not update themselves in accordance to Gentoo's release cycle and therefore require a static snapshot of the distribution from the time the image was created. Meta information about the distribution is stored in the portage tree, and Aldeberan now provides the necessary snapshot under the same downloads directory.

```
> cd gentoo/usr
> tar -xjvf $PORTAGE
```

Once this snapshot has been obtained, it should not be updated via `emerge --sync` and friends. Doing so would cause the portage tree to become desynchronised from the software included in the opennao image.

4.1.2 Entering the chroot

Before the chroot is ready to be entered, the system directories must mounted inside the chroot. resolve.conf is also needed.

```
> cd gentoo && mkdir -p proc dev/pts sys
> mount -t proc none proc
> mount -o bind /dev dev
> mount -o bind /dev/pts dev/pts
> mount -o bind /sys sys
> cp /etc/resolve.conf etc/resolve.conf
```

The above procedure needs to be run after every reboot, but can be automated via a shell script.

Finally, enter the chroot.

```
> chroot gentoo /bin/bash
```

4.2 Building Packages

The chroot environment differs substantially from a vanilla gentoo install, and there are a few common hurdles that are frequently stumbled upon when trying to build packages. The gentoo package management is well documented in other places [1] so only problems specific to the Nao are covered here.

4.2.1 Finding Outdated Source Code

Because the portage snapshot is now over a year old, many of the servers that previously hosted source code required to build opennao have ceased to do so. One solution is to tell portage to look in as many places as possible, and this is done by first installing and then using the `mirrorselect` tool.

```
> emerge app-portage/mirrorselect
> mirrorselect -i -o >> /mnt/gentoo/etc/make.conf
```

You will be prompted to choose from a list of mirrors, and you should select as many as possible.

Even so, it is possible that the source is present on none of the mirrors. In this case, the packages can often be found via a google search and manually placed into `/usr/portage/distfiles`. If the checksums of the files match those stored in the portage tree, portage will not try to re-download the package. Otherwise, the package `Manifest` file will need to be updated.

4.2.2 Modifying Portage Packages

Sometimes it is necessary to modify packages before building them. One such example is when the exact source archive cannot be found, but a similar one will suffice. As a security feature, portage stores multiple checksums of each package file and verifies that the files fetched match those that the Gentoo maintainers expected. However, this means that if we modify a package ourselves we must fix the checksums. Other possible reasons for modifying packages include changing ebuild scripts and making source modifications.

Checksums are stored in the `Manifest` file, and it will need to be rebuilt. If only the ebuild script is modified, simply rebuilding the manifest suffices.

```
> cd /usr/portage/$CATEGORY/$PACKAGE
> ebuild $PACKAGE-$VERSION.ebuild manifest
```

If a source modification is required, it is best done by making a patch, then placing that patch into the `files` directory of the package. The ebuild script will also need to be updated to apply the patch.

```
> cd /usr/portage/$CATEGORY/$PACKAGE
> cp my.patch files/
> ebuild $PACKAGE-$VERSION.ebuild manifest
```

If the distributed source archive must be changed, first delete the line in the `Manifest` file corresponding with the old entry. Extract the sources, modify them, and then repackage them using the original archive name. Place the modified archive into the `/usr/portage/distfiles` directory and finally update the manifest as above.

4.2.3 OpenGL

We rely on OpenGL for three dimensional rendering in our debug tool. OpenGL is actually just a specification, and there exist implementations provided by the major graphics hardware vendors as well as one produced by the Mesa project. Mesa is able to fall back to indirect rendering as defined by the GLX specification if it cannot initialise a direct rendering context. Indirect rendering is highly portable and will work with any X server, therefore we use it in our modified opennao image.

Mesa is an incredibly complicated piece of software supporting a wide range of hardware, and is unfortunately inherently difficult to configure. In addition, we uncovered an undocumented bug that will not be fixed due to the age of the Mesa release. The bug pertains to rendering with array buffers, but can be worked around by disabling the newer gallium driver and falling back to the old one.

The exact options we pass to the configure script are as follows:

```
> ./configure
> --disable-debug --prefix=/usr --build=i686-pc-linux-gnu
> --host=i686-pc-linux-gnu --mandir=/usr/share/man
> --infodir=/usr/share/info --datadir=/usr/share
> --sysconfdir=/etc --localstatedir=/var/lib
> --disable-option-checking --with-driver=xlib --disable-glut
> --without-demos --disable-glw --disable-motif --enable-glx-tls
> --disable-gallium --with-state-trackers=glx --disable-gallium-llvm
> --disable-gles2 --disable-gles2 --disable-gles-overlay
> --disable-glx-rts --disable-asm --disable-openvg
```

The `x11-libs/mesa` ebuild script will need to be updated to supply these options to configure. Once mesa has been emerged, if the `libGL` library installed to `/usr/lib/opengl/xorg-x11/lib/` has a name of the form `libGL.so.1.5.xxxxx` then the library has been correctly configured. Otherwise, the name will be of the form `libGL.so.1.x`.

4.2.4 Valgrind

Valgrind is unusually picky about its build environment, however if configured on a i686 machine with a 2.6 kernel it should build without problem. Otherwise, the configure script must be modified in two locations.

```
 case "${host_cpu}" in
-      i?86)
+      *)

 case "${kernel}" in
-      2.6.*)
+      *)
```

We use a patch file to achieve the modification.

4.3 Packaging

The robots do not come with the Gentoo package manager portage installed, nor do they have any notion of packages. We therefore chose to do our package management within the chroot, and then repackage the portage packages as simple archives.

4.3.1 Dependency Management

We use a custom shell script to recursively generate package dependencies and then archive and place the packages into a `/packages` directory. If a dependency is already present in the packages directory, the recursion stops. To avoid building packages that are already part of opennao, we also test if any file from a package is already present in the opennao image. If present, as before the recursion stops. The below script invocation builds the libmesa package and all its dependencies. The script itself is listed under B.2.

```
> bash make_package.sh media-libs/mesa-7.10-r1
```

4.3.2 Installation and Uninstallation

Installation is done by simply extracting the archives to the desired location, probably either to the root of the robot or to the cross tool chain's `sysroot` directory. If a package needs to be uninstalled, this can be achieved using `xargs` to remove all files contained in the archive. The below is an example of installing and uninstalling a package.

```
> cd $SRC/robocup2012/nao-atom-cross-toolchain-1.12.3/sysroot
> tar -xjf package.tar.bz2
> tar -tf package.tar.bz2 | xargs -L1 rm
```

4.3.3 Issues with eselect

Gentoo uses `eselect` to allow multiple versions of packages to be installed simultaneously and swapped between when needed. A good example is `python2` and `python3`. It works by installing each package into a private directory and then creating symlinks to the currently selected package. These symlinks are not included by the packaging mechanism and must be set up manually after installation. Fortunately, few packages make use of the eselect system and this is generally not a problem.

4.3.4 OpenGL

OpenGL is the only library in our modified opennao image that uses the eselect system. `/usr/lib/libGL.so` and `/usr/lib/libGL.so.1` must link to `/usr/lib/opengl/xorg-x11/lib/libGL.so.1.5.xxxxx`.

```
> cd $INSTALL_DIR/usr/lib/
> ln -s /usr/lib/opengl/xorg-x11/lib/libGL.so.1.5.xxxxx libGL.so
> ln -s /usr/lib/opengl/xorg-x11/lib/libGL.so.1.5.xxxxx libGL.so.1
```

4.3.5 Generating Qt headers

The Qt package does not correctly generate a gentoo specific header file, namely `gentoo-qconfig.h`. We generate this manually using the below commands.

```
> cd $INSTALL_DIR/usr/include/qt4/Gentoo/
> rm -f gentoo-qconfig.h
> ls *-*-*-*.h -1 | awk '{print "#include <Gentoo/" $0 ">"}' > gentoo-qconfig.h
```

4.3.6 Python 2

A python package should not be generated by the packaging script, however it is worth noting that the python headers installed on the robot conflict with those in our chroot. Python is already present on both the robot and in the cross tool chain, and therefore there is no need to install it.

Chapter 5

Conclusions

A well designed and implemented foundation is essential for any platform that allows for the rapid development and testing of experimental ideas. One observation about our foundation is that lower level systems seldom change whilst higher level code built atop the foundation tends to change often. It is therefore important to understand and document the foundations of our system, as these are the parts that are unlikely to be touched again until a time when the original expertise responsible for the part has long since left the team. This is perhaps an area for improvement, though rUNSWift generally does a good job documenting our more experimental features.

Improvements to the infrastructure documented in this report were motivated both by previous teams' experiences and changes made by Aldebaran to the Nao v4 robot. In particular, through conversations with other teams at previous competitions and also through their team reports, we now know our solution to the inverse kinematic problem and the implementation of our python bridge to be on par with that of other teams. We have also documented and compared these to our previous implementations for future teams' reference.

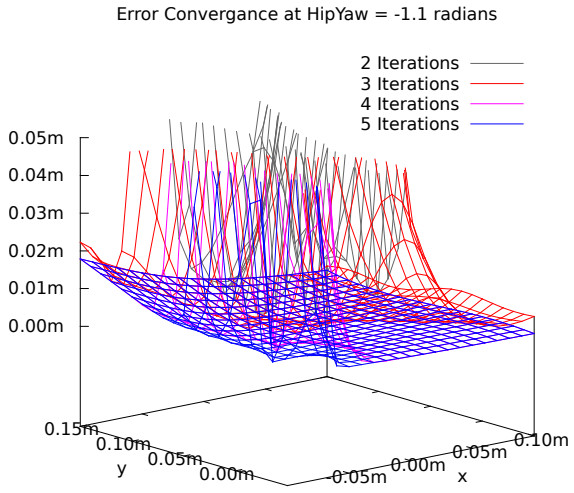
Whilst these improvements are not necessarily novel, having a well performing implementation of the foundations is simply a necessity for future development. The documentation and analysis that now accompanies these areas will be of great assistance if and when a future team must revisit them.

Appendix A

Graphs

A.1 Iterative Method Error Convergence

These graphs show the convergence of the iterative inverse kinematics method at different rotations about the hip yaw joint. The iterative solution first calculates an initial solution assuming that the hip yaw rotation is zero, and then iteratively finds a solution with the correct hip yaw. Therefore, larger hip yaw values will result in more error in the initial solution. To give an indication of how the method converges across the leg chain's range of motion, the error surfaces at each iteration have been plotted as functions of the foot's target x and y translation. It can be seen that each successive iteration's error surface approaches zero as the number of iterations increases.



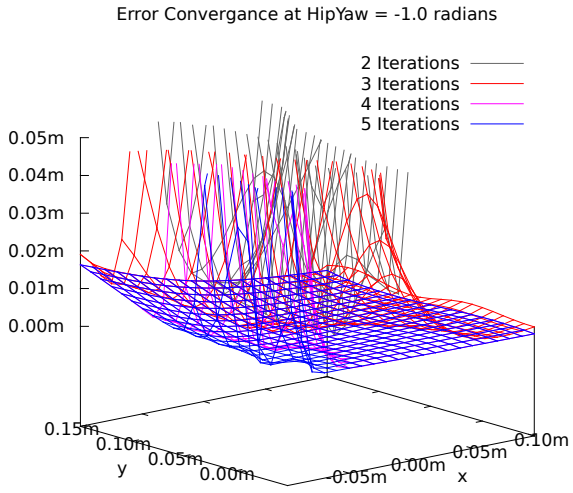
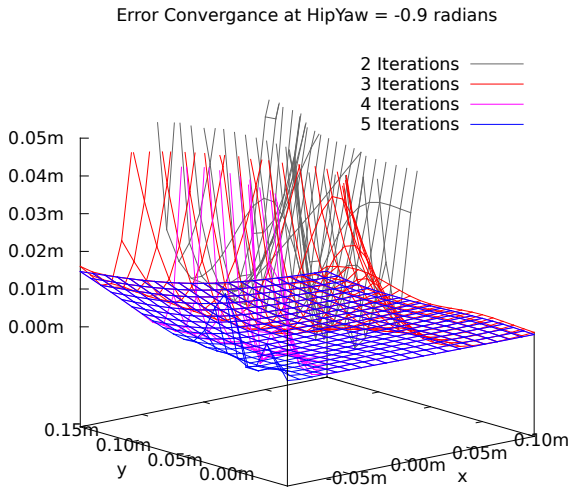
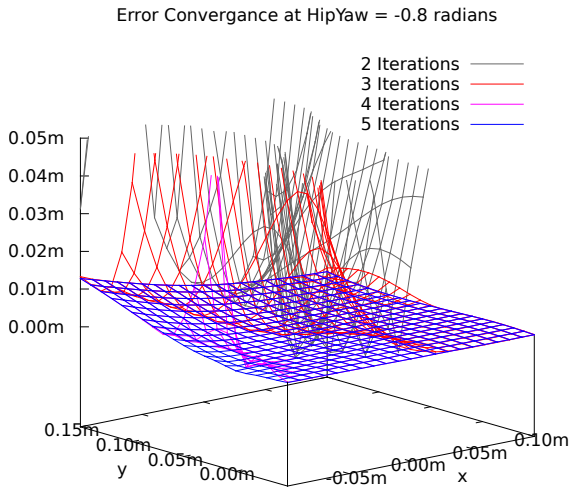
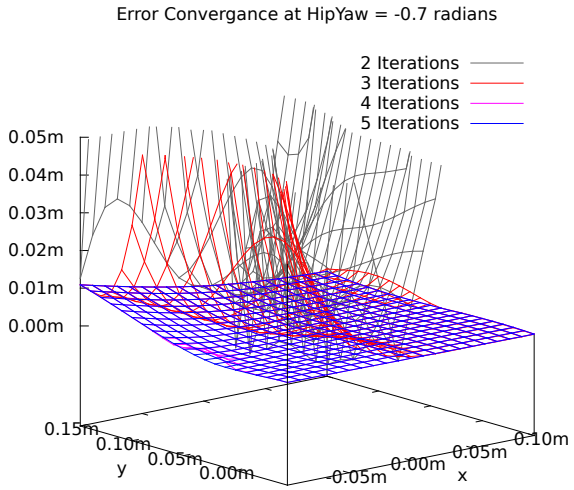


Table A.1: Iterative Error Convergence

Continued on next page







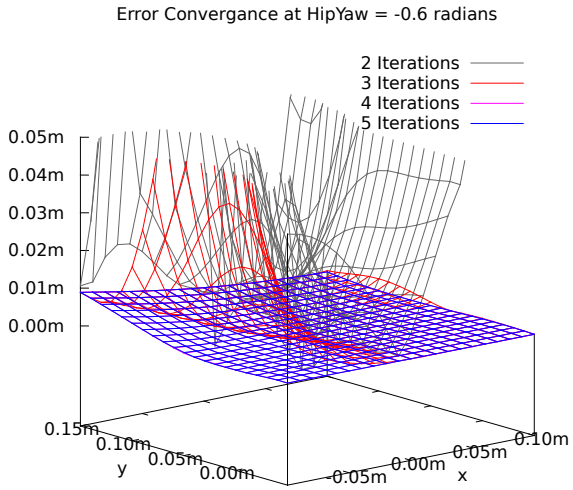
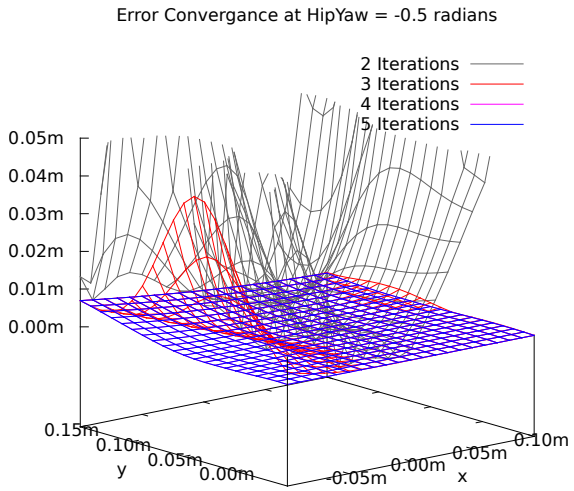
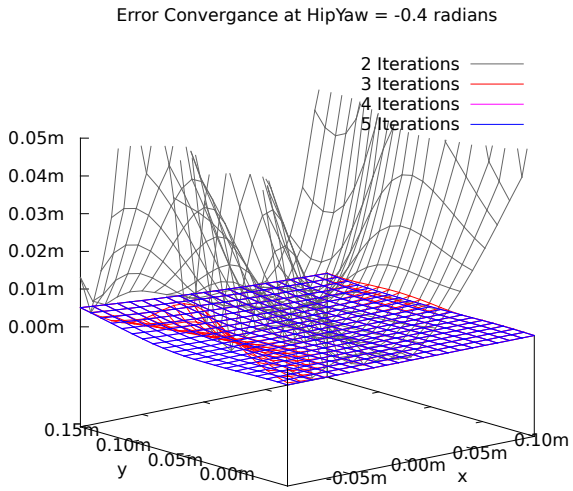
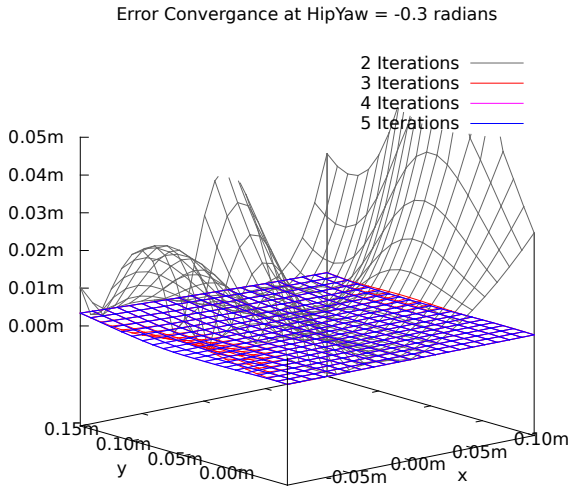


Table A.1: Iterative Error Convergence

Continued on next page







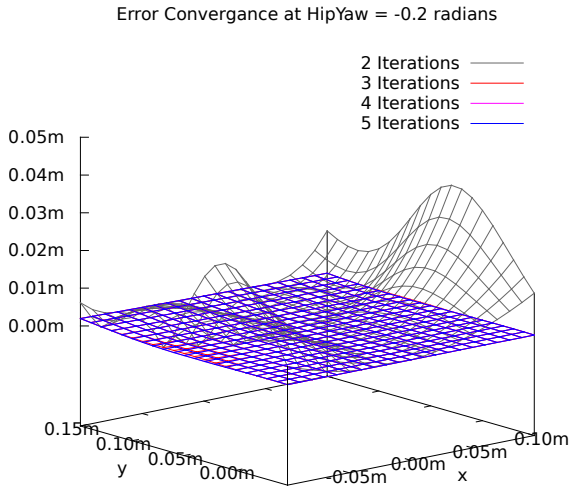
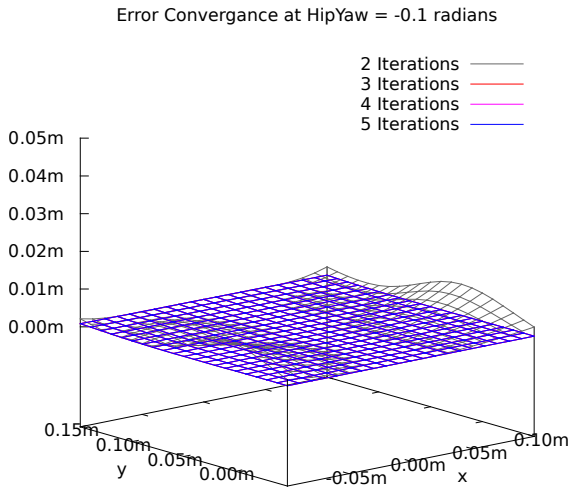
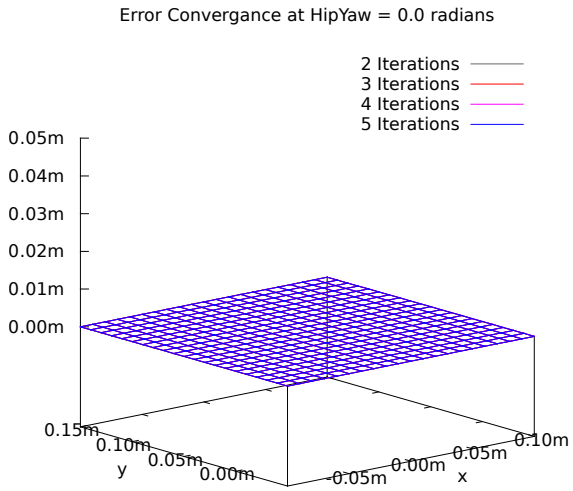
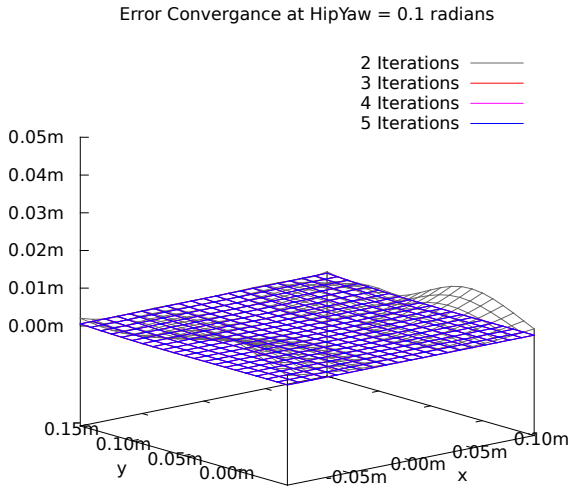


Table A.1: Iterative Error Convergence

Continued on next page







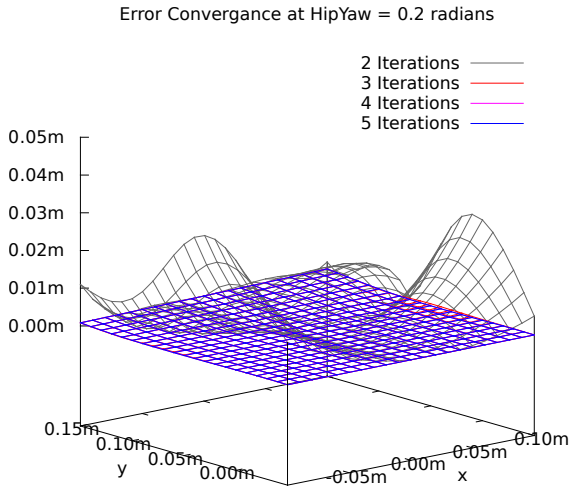
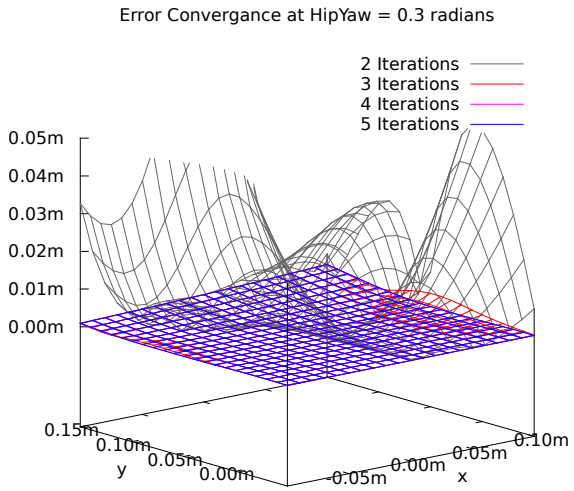
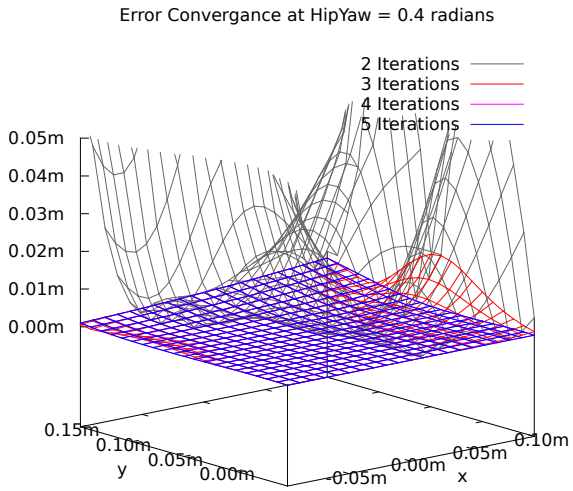
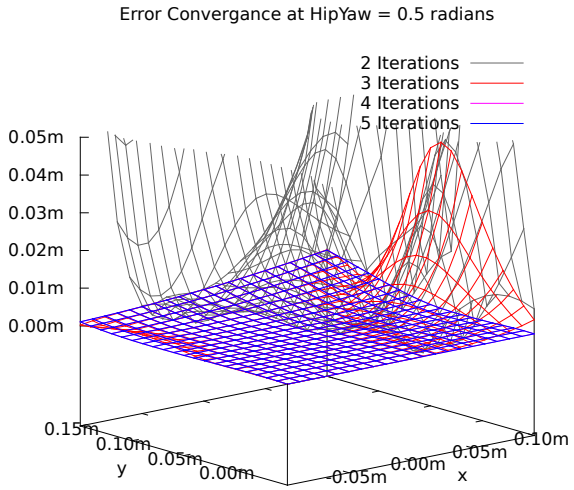


Table A.1: Iterative Error Convergence

Continued on next page







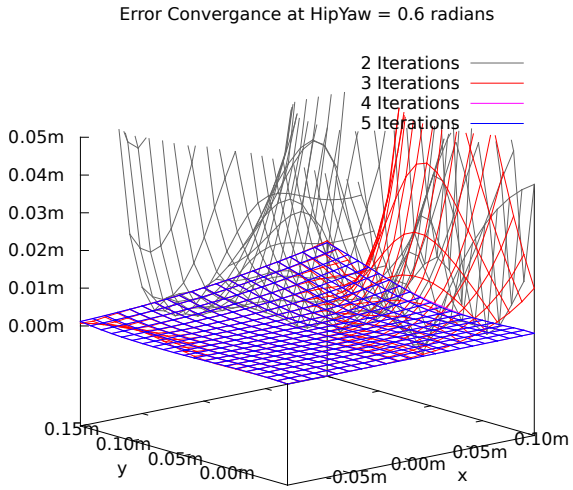
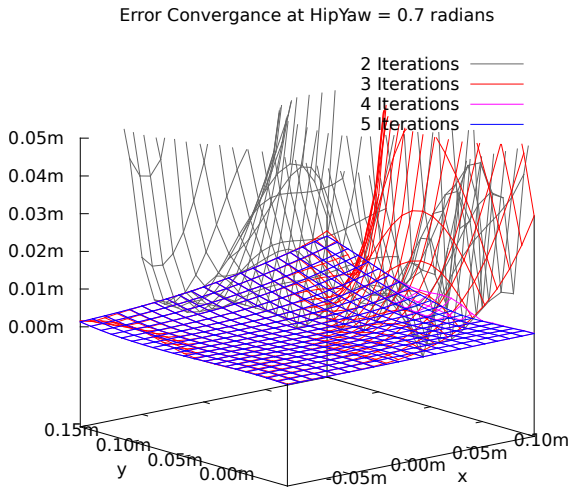


Table A.1: Iterative Error Convergence

Continued on next page



A.2 Yaw Error Fitted Surfaces

In our solution there is an error between the foot yaw specified and the actual resulting foot yaw. This error is plotted in the red on the below graphs. The blue surfaces show the result of fitting a polynomial to the error and then subtracting it to obtain a corrected surface close to zero. Higher order polynomials result in corrected surfaces closer to zero.

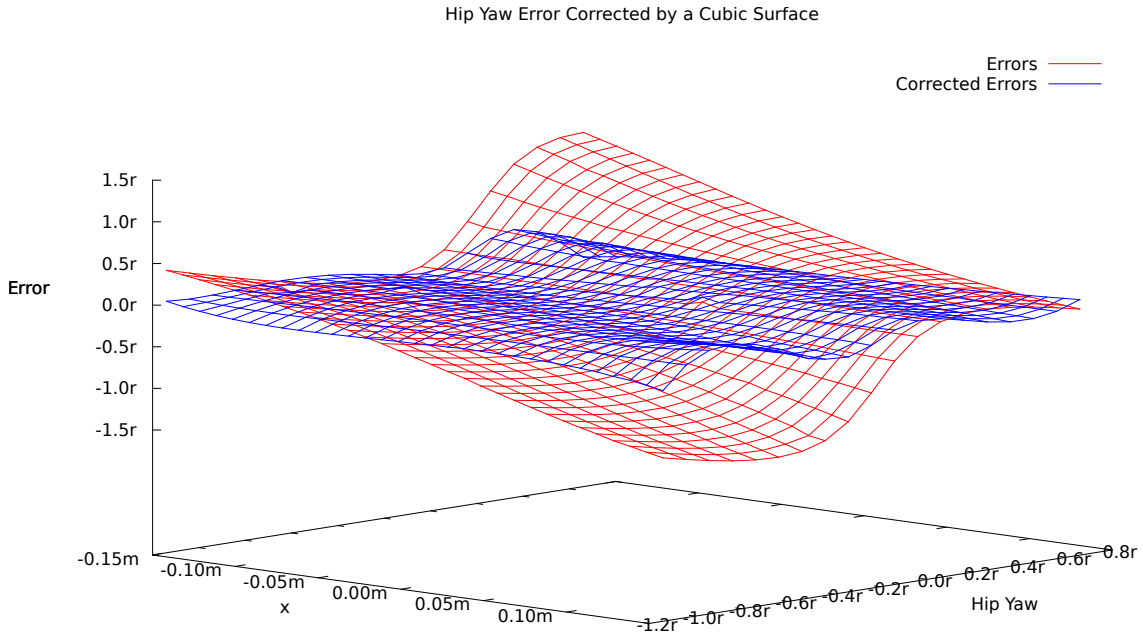


Figure A.1: Yaw error corrected by a cubic surface.

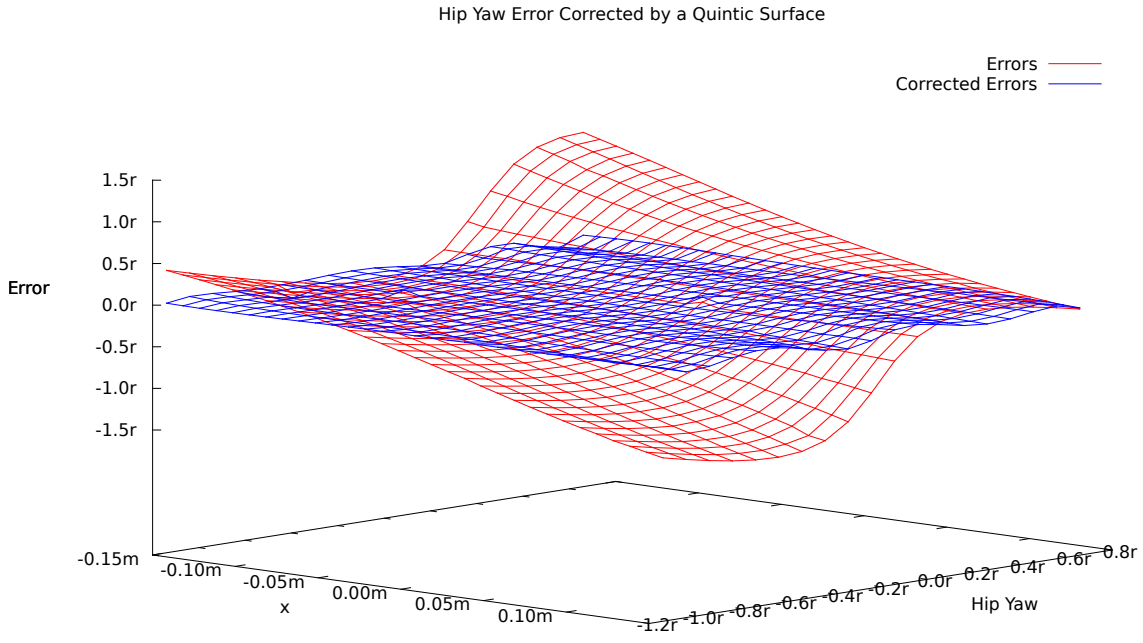


Figure A.2: Yaw error corrected by a quintic surface.

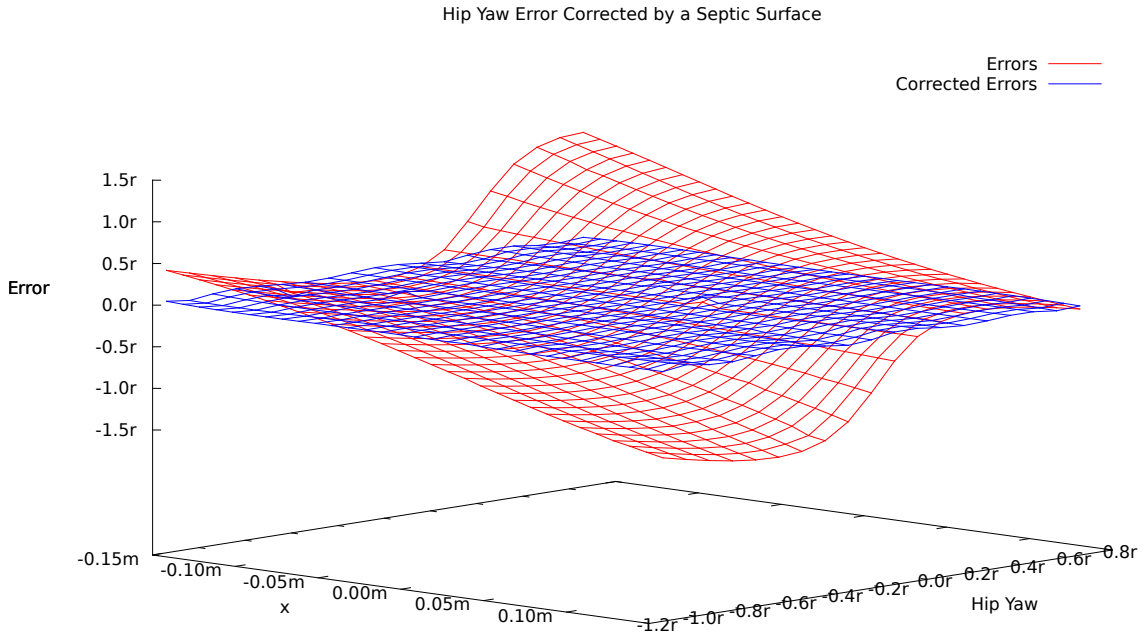


Figure A.3: Yaw error corrected by a septic surface.

Appendix B

Source Listings

B.1 IKinematics.cpp

C code for computing the inverse kinematic chain on the Aldebaran Nao.

```
#include <cmath>
#include <Eigen/Eigen>

using namespace Eigen;

static float crop(float angle)
{
   while (angle <= -M_PI) angle += 2*M_PI;
   while (angle >   M_PI) angle -= 2*M_PI;

   return angle;
}

NaoLegChain NaoSolve(const NaoFootTarget &target, HipYawLock lock)
{
   const float torsoL = 0.085;
   const float torsoW = 0.05;
   const float thighL = 0.10000;
   const float shinL  = 0.10290;
   const float footL  = 0.04519;

   NaoLegChain r;
   float x, y, z;

   /* Remove translation from torso centre to hip joint. */
   x = target.x;
   y = target.y - torsoW;
   z = target.z + torsoL;

   /* Assume hip yaw is equal to target foot yaw. */
   r.hipYaw = target.yaw;

   const float sx = sinf(target.roll  ), cx = cosf(target.roll );
   const float sy = sinf(-target.pitch), cy = cosf(target.pitch);
   const float sz = sinf(target.yaw   ), cz = cosf(target.yaw  );
   const float cycz = cy*cz, sxsy = sx*sy, cysz = cy*sz;

   Matrix3f targetM;
```

```
    targetM <<  cycz + sxsy*sz, cz*sxsy - cysz , cx*sy,
                          cx*sz,           cx*cz,   -sx,
               -cz*sy + cysz*sx, cycz*sx + sy*sz, cx*cy;

    /* Subtract the foot link from the target. The foot bone is (0, 0, -footL). */
    x += footL * targetM(0,2);
    y += footL * targetM(1,2);
    z += footL * targetM(2,2);

    if (r.hipYaw != 0.f) {
        Vector3f v(x, y, z);
        AngleAxis<float> a(-r.hipYaw, Vector3f(0, sqrtf(2)/2, -sqrtf(2)/2));
        v = a * v;
        x = v.x();
        y = v.y();
        z = v.z();
    }

    r.hipRoll = atan2f(y, -z);

    const float z_prime = z / cosf(r.hipRoll);
    const float h = hypotf(z_prime, x);

    const float A = acosf((shinL*shinL - h*h - thighL*thighL) / (-2*thighL*h));
    const float B = acosf((h*h - thighL*thighL - shinL*shinL) / (-2*thighL*shinL));
    r.hipPitch    = -(A + asin(x/h));
    r.kneePitch   = -(B + M_PI);

    const float legPitch = r.hipPitch + r.kneePitch;
    Matrix3f toFootInvM = (
        AngleAxis<float>(-legPitch , Vector3f(0, 1, 0)) *
        AngleAxis<float>(-r.hipRoll, Vector3f(1, 0, 0)) *
        AngleAxis<float>(-r.hipYaw , Vector3f(0, sqrtf(2)/2, -sqrtf(2)/2))
    ).toRotationMatrix();

    Matrix3f footM = toFootInvM * targetM;

    r.ankleRoll  = -asinf(footM(1,2));
    r.anklePitch = atan2f(footM(0,2), footM(2,2));

    /* Finally make sure all angles are bound by (-pi, pi] */
    r.hipYaw     = crop(r.hipYaw    );
    r.hipRoll    = crop(r.hipRoll   );
    r.hipPitch   = crop(r.hipPitch  );
    r.kneePitch  = crop(r.kneePitch );
    r.ankleRoll  = crop(r.ankleRoll );
    r.anklePitch = crop(r.anklePitch);

    return r;
}
```

B.2 make_package

Shell script for packaging portage packages into an archive for deployment on the robots.

```
#!/bin/bash
in_robot()
{
        local f
        for f in `epm -ql $1 | grep '\.\(so\|a\)\(\.\|$\)'`; do
                if test -f "/robot$f"; then
                        continue
                fi
                return 1
        done
        return 0
}
in_packages()
{
        test -f "/packages/$1.tbz2"
}
package()
{
        umask 022
        mkdir -p /packages/`echo $1 | sed 's@/.*$@@'`
        epm -ql $1 | cut -c2- | xargs tar -cjf /packages/$1.tbz2
        get_deps $1
}
get_deps()
{
        echo "Fetching deps of $1."
        local f=`mktemp`
        equery -q depgraph -l -U --depth=1 $1 | cut -f1 -d' ' > $f
        if [ `cat $f | grep ':$' | wc -l` != 1 ]; then
                echo "$f matched more than one package:"
                cat $f | grep ':$'
                echo "specify version number."
                rm $f
                return 1
        fi
        for i in `cat $f | tail -n+3`; do
                if in_packages $i; then
                        echo "$i has previously been packaged."
                elif in_robot $i; then
                        echo "Robot has package $i."
                else
                        package $i
                fi
        done
        rm $f
        return 0
}
if test $# -ne 1; then
      echo "Usage: $0 category/package-version"
elif test -z "$(ls /robot 2> /dev/null)"; then
      echo "/robot not mounted. mount -o bind /path/to/robot/image/root root"
else
      pushd /; package $1; popd
fi
```

Bibliography

[1] Gentoo handbook. http://www.gentoo.org/doc/en/handbook/.

[2] Jordan Brindzaand Alexandra Leeand Anirudhaand Majumdarand Barry Scharfmanand Anne. Robocup standard platform league team report 2009, 2009.

[3] Samuel R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods, 2009.

[4] David Eberly. Euler angle formulas, 1999.

[5] Prof. Eric Chownand Jeremy Fishmanand Johannes Stromand George Slavovand Tucker Hermansand Nicholas Dunnand Andrew Lawrenceand John Morrisonand Elise Krob. The northern bites 2008 standard platform robot team, 2008. http://robocup.bowdoin.edu.

[6] Prof. Eric Chownand Tucker Hermansand Johannes Stromand George Slavovand Jack Morrisonand Andrew Lawrenceand Elise Krob. Northern bites 2009 team report, 2009. http://robocup.bowdoin.edu.

[7] Thomas Röfer, Tim Laue, Judith Müller, Oliver Bösche, Armin Burchardt, Erik Damrose, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Andrik Rieskamp, André Schreck, Ingo Sieverdingbeck, and Jan-Hendrik Worch. B-human team report and code release 2009, 2009. Only available online: http://www.b-human.de/downloads/bhuman09_coderelease.pdf.

[8] Thomas Röfer, Tim Laue, Judith Müller, Armin Burchardt, Erik Damrose, Alexander Fabisch, Fynn Feldpausch, Katharina Gillmann, Colin Graf, Thijs Jeffry de Haas, Alexander Härtl, Daniel Honsel, Philipp Kastner, Tobias Kastner, Benjamin Markowsky, Michael Mester, Jonas Peter, Ole Jan Lars Riemann, Martin Ring, Wiebke Sauerland, André Schreck, Ingo Sieverdingbeck, Felix Wenk, and Jan-Hendrik Worch. B-human team report and code release 2010, 2010. Only available online: http://www.b-human.de/downloads/bhuman10_coderelease.pdf.