UT Austin Villa 2008: Standing On Two Legs

Todd Hester, Michael Quinlan and Peter Stone
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, Texas 78712-1188
{todd,mquinlan,pstone}@cs.utexas.edu
`http://www.cs.utexas.edu/~AustinVilla`

Technical Report UT-AI-TR-08-8

November 3, 2008

Abstract

In 2008, UT Austin Villa entered a team in the first Nao competition of the Standard Platform League of the RoboCup competition. The team had previous experience in RoboCup in the Aibo leagues. Using this past experience, the team developed an entirely new codebase for the Nao. Development took place from December 2007 until the competition in July of 2008. This technical report describes the algorithms and code developed by the team for the 2008 RoboCup competition in Suzhou, China. A major development was a software architecture designed for easy use, extendability, and debugability. On top of this architecture, the team developed modules for vision, localization, motion, and behaviors. These developments provide a strong foundation for our team to compete successfully in the Standard Platform League in future RoboCup competitions.

1 Introduction

RoboCup, or the Robot Soccer World Cup, is an international research initiative designed to advance the fields of robotics and artificial intelligence, using the game of soccer as a substrate challenge domain. The long-term goal of RoboCup is, by the year 2050, to build a team of 11 humanoid robot soccer players that can beat the best human soccer team on a real soccer field [4].

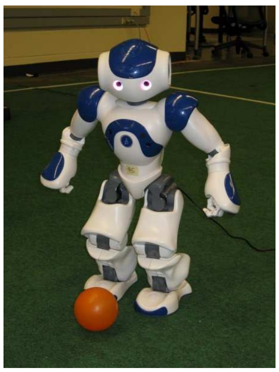

Figure 1: The Aldebaran Nao robot.

RoboCup is organized into several leagues, including both simulation leagues and leagues that compete with physical robots. This report describes our team's entry in the Nao division of the Standard Platform League (SPL)[1]. In the SPL, all the teams compete with identical robots, making it essentially a software competition. All the teams used identical humanoid robots from Aldebaran called the Nao[2], shown in Figure 1.

Our team is UT Austin Villa[3], from the Department of Computer Sciences at the University of Texas at Austin. Our team is made up of Professor Peter Stone, PhD student Todd Hester, and postdoc Michael Quinlan, all veterans of past RoboCup competitions. We started the codebase for our Nao team from scratch in December of 2007. Our previous work on Aibo teams [12, 13, 14] provided us with a good background for the development of our Nao team. We developed the architecture of the code in the early months of development, then worked on the robots in simulation, and finally developed code on the physical robots starting in March of 2008. Our team competed in the RoboCup competition in Suzhou, China in July of 2008.

This report describes all facets of our development of the Nao team codebase. Section 2 describes our software architecture that allows for easy extendability and debugability. Our approaches to vision and localization are similar to what we have done in the past [14] and are described in sections 3 and 4 respectively. One of the biggest changes from the Aibo league was moving from a four-legged robot to a two-legged robot. We developed a motion engine for the Nao to address this challenge, which is described in section 5. Section 6 briefly describes the behaviors we developed on the robot. Section 7 presents our results from the competition and section 8 concludes the report.

2 Software Architecture

The introduction of the Nao allowed us to redesign the software architecture without having to support legacy code. Previous RoboCup efforts had taught us that the software should be flexible to allow quick changes but most importantly it needs to be debugable.

[1]http://www.tzi.de/spl/
[2]http://www.aldebaran.com/
[3]http://www.cs.utexas.edu/~AustinVilla

The key element of our design was to enforce that the environment *interface*, the agent's *memory* and its *logic* were kept distinct (Figure 2). In this case *logic* encompasses the expected vision, localization, behavior and motion modules. Figure 3 provides a more in-depth view of how data from those modules interact with the system.

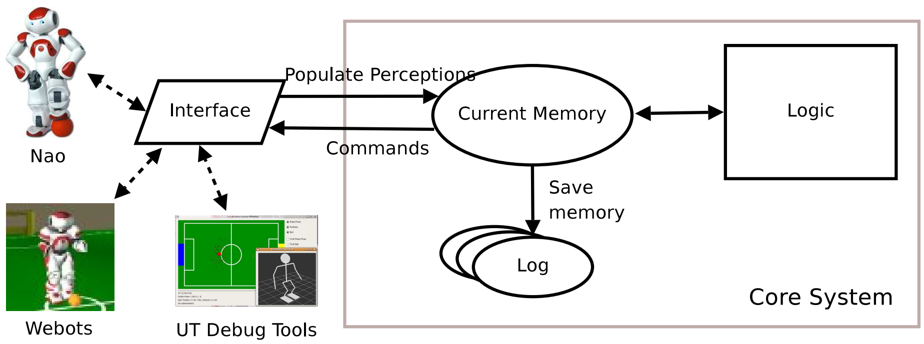

Figure 2: Overview of the 2008 UT Austin Villa software architecture.

The design advantages of our architecture are:

Consistency The *core system* remains identical irrespective of whether the code is run on the robot, in the simulator or inside our debug tool. As a result, we can test and debug code in any of the 3 environments without fear of code discrepancies. The robot, simulator and tools each have their own *interface* class which is responsible for populating *memory*.

The robot interface talks to NaoQi (and related modules) to populate the perceptions and then reads from memory to give commands to ALMotion. Since July the simulation interface also communicates with NaoQi; previously it communicated via the older Webots/Nao API. The tool interface can populate memory from either a saved log file or over a network stream.

Flexibility The internal *memory* design is show in Figure 3. We can easily plug & play modules into our system by allowing each module to maintain its own local memory and communicate to other modules using the common memory area. By forcing communication through these defined channels we prevent 'spaghetti code' that often couples modules together. For example, a Kalman Filter localization module would read the output of vision from common memory, work in its own local memory and then write object locations back to common memory. The memory module will take care of the saving and loading of the new local memory, so the developer of a new module does not have to be concerned with the low level saving/loading details associated with debugging the code.

Debugability At every time step only the contents of current *memory* is required to make the logic decisions. We can therefore save a "snapshot" of the current memory to a log file (or send it over the network) and then examine the log in our debug tool and discover any problems. The debug tool not only has the ability to read and display the logs, it also has the ability to take logs and process them through the logic modules. As a result we can modify code and watch the full impact of that change in our debug tool before testing it on the robot or in the simulator. The log file can contain any subset of the saved modules, for example saving only percepts (i.e. the image and sensor readings) is enough for us to regenerate the rest of the log file by passing through all the logic modules (assuming no changes have been made to the logic code).

It would be remiss not to the mention the main disadvantage of this design. We implicitly have to "trust" other modules to not corrupt data stored in memory. There is no hard constraint blocking one module writing data into a location it shouldn't, for example localization could overwrite a part of common memory that should only be written to by vision. We could overcome this drawback by introducing read/write permissions on memory, but this would come with performance overheads that we deem unnecessary.

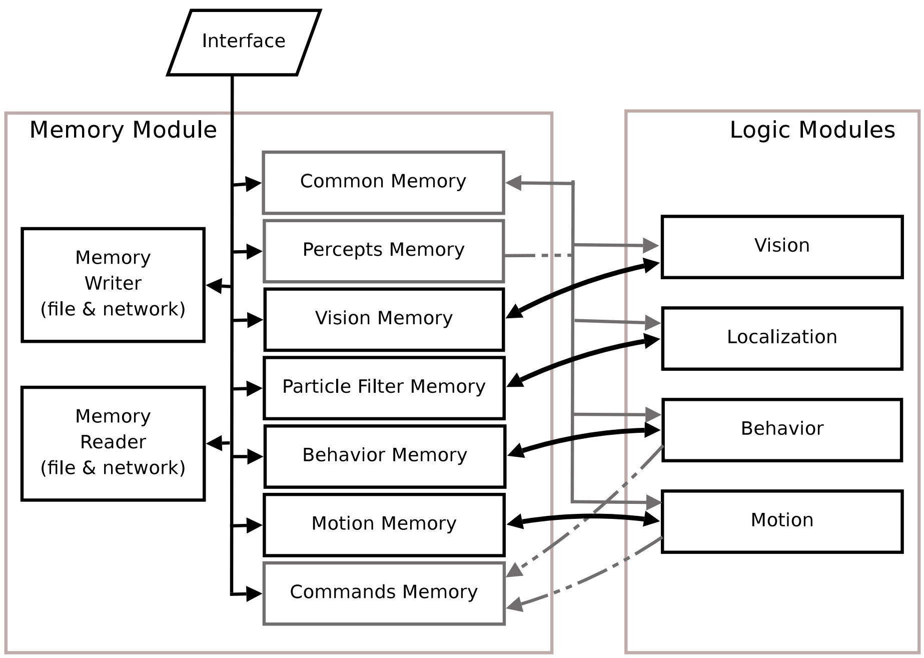

Figure 3: Design of the Memory module. The gray boxes indicated memory blocks that are accessed by multiple logic modules. Dotted lines show connections that are either read or write only (percepts are read only, commands are write only).

,

2.1 Languages

The code incorporates both C++ and Lua. Lua is a scripting language that interfaces nicely with C++ and it allows us to change code without having to recompile (and restart the binary on the robot). In general most of vision, localization and motion are written in C++ for performance reasons, but all control decisions are made in Lua as this gives us the ability to turn on/off sections at runtime. Additionally the Lua area of the code has access to all the objects and variable stored in C++ and therefore can be used to prototype code in any module.

3 Vision

In 2008 we used the 160×120 pixel image primarily because the early version of the simulator and robot only provided this image size. Since the number of pixels was relatively low we were able to use same image processing techniques that were applied by most AIBO teams [14, 9].

First, the YUV image is segmented into known colors. Second, blobs of continuous colors are formed and finally, these blobs are examined to see if they contain an object (ball, goal, goal post). Additionally a line detection algorithm is run over the segmented image to detected field lines and intersections (L's or T's).

Figure 4 gives an example of a typical vision frame. From left-to-right we see the raw YUV image, the segmented image and the objects detected. In this image the robot identified an *unknown blue post* (blue rectangle), an *unknown L intersection* (yellow circle), an *unknown T intersection* (blue circle), and three *unknown lines*.

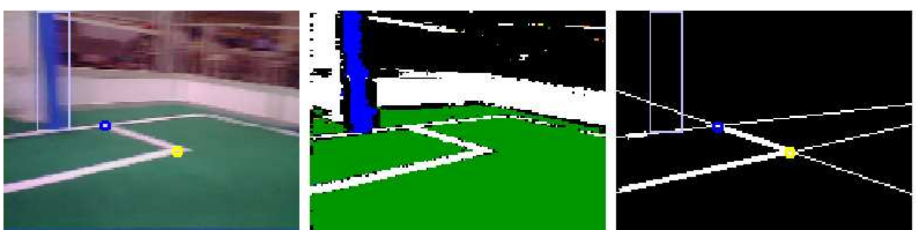

Figure 4: Example of the vision system. Left-to-Right: The raw YUV image, the segmented image and the observed objects

A key element to gaining more accurate distances to the ball is circle fitting. We apply a least squared circle fit based upon the work of Seysener et al [10]. Figure 5 presents example images of the ball. In the first case the robot is "bending over" to see a ball at its own feet. The orange rectangle indicates the bounding box of the observed blob, while the pink circle is the output of the circle fit. For reference we show a similar image gathered from the simulator. The final image shows a ball that is occluded, in this case it extends off the edge of the image. In this situation a circle fit is the only method of accurately determining the distance to the ball.

3.1 Field Lines and Intersections

The line detection system returns lines in the form of $ax + by + c = 0$ in the image plane. These lines are constructed by taking the observed line segments (the bold white lines in the lower right image of Figure 6) and forming the general equation that describes the infinite extension of this line (shown by the thin white lines). However, localization currently requires a distance and angle to the closest point on the line after it has been projected to the ground plane.

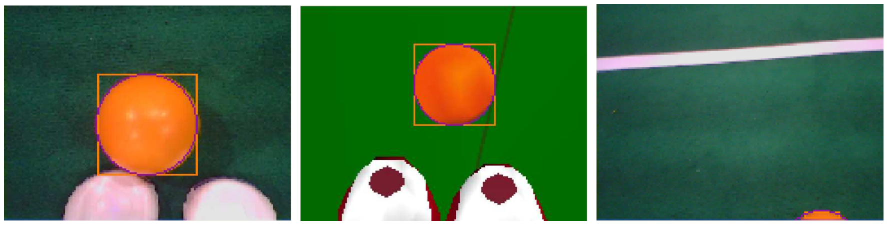

Figure 5: Example of circle fitting applied to the ball. The first two images a examples of a ball when the robot is preparing to kick (an image is presented from the robot and from the simulator). The third image shows a ball where circle fitting is required to get an accurate estimate of distance.

To find the closest point on the line, we take two arbitrary points on the image line (in our case we use the ends of the observed segments) and translate each point to the ground plane using method described in Section 3.1.1. We can then obtain the equation of the line in the ground plane by forming the line that connects to these two points. Since this projected line is relative to the robot (i.e. the robot is positioned at the origin), the distance and bearing to the closest point can be obtained by simple geometry. The output of the translations can be seen in Figure 6.

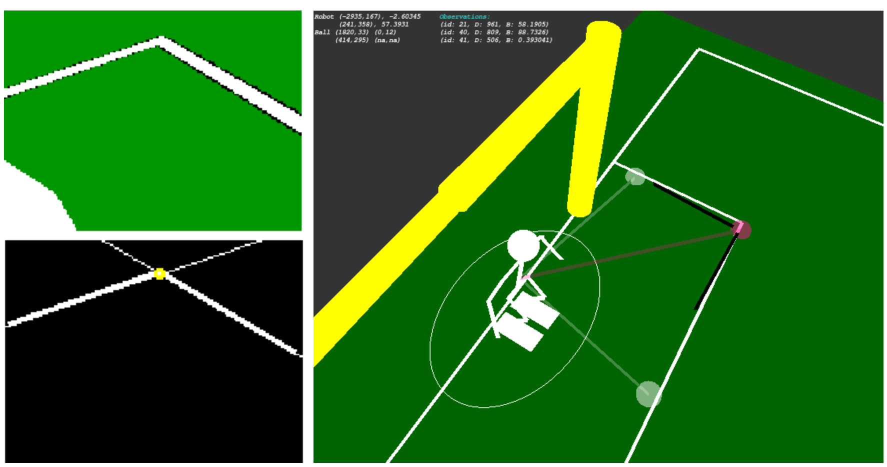

Figure 6: Example of transforming lines for use in localization. The images on the left show the segmented image and the results of object recognition, in this case two lines and an L intersection. The image on the right shows these objects after they have been transformed to the ground plane. The two black lines indicate the observed line segments, the white circles show the closest points to the infinite extension of each line and the pink circle indicates the observed location of the L intersection.

3.1.1 Calculating distance, bearing and elevation to a point in an image

Given a pixel (x_p, y_p) in the image we can calculate the bearing (θ_b) and elevation (θ_e) relative to the camera by:

$$\theta_b = tan^{-1}\left(\frac{w/2 - x_p}{w/(2 \cdot tan(FOV_x/x_p))}\right), \theta_e = tan^{-1}\left(\frac{h/2 - y_p}{h/(2 \cdot tan(FOV_y/y_p))}\right)$$

where w and h are the image width and height and FOV is the field of view of the camera.

Typically for an object (goal, ball, etc.) we would have an estimated distance (d) based on blob size. We can now use the pose of the robot to translate d, θ_b, θ_e into that object's location (x, y, z), where (x, y, z) is relative to a fixed point on the robot. In our case this location is between the hips. We define $(x, y, z) = transformPoint(d, \theta_b, \theta_e)$ as:

$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cos\theta_b & 0 & sin\theta_b & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta_b & 0 & cos\theta_b & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta_e & sin\theta_e & 0 \\ 0 & -sin\theta_e & cos\theta_e & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 70 \\ 0 & 0 & 1 & 50 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\phi_p & -sin\phi_p & 0 \\ 0 & sin\phi_p & cos\phi_p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cos\phi_y & 0 & sin\phi_y & 0 \\ 0 & 1 & 0 & 0 \\ -sin\phi_y & 0 & cos\phi_y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 211.5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta_p & -sin\theta_p & 0 \\ 0 & sin\theta_p & cos\theta_p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where ϕ_p and ϕ_y are angle of the head pitch and head yaw joints. θ_p is the body pitch as calculated in Section 5.3 and the constants are the camera offset from the neck joint (70mm and 50mm) and the distance from the neck joint to our body origin (211.5mm).

For a point on a line we do not have this initial distance estimate and instead need an alternate method for calculating the relative position. The approach we use is to solve the above translation for two distances, one very short and one very long. We then form a line (ℓ) between the two possible locations in 3D space. Since a field line must lie on the ground, the relative location can be found at the point where ℓ crosses the ground plane (Eq 1). We can then calculate the true distance, bearing and elevation to the point (Eq 2).

$$(x_1, y_1, z_1) = transformPoint(200, \theta_b, \theta_e)$$
$$(x_2, y_2, z_2) = transformPoint(20000, \theta_b, \theta_e)$$

$$y = -height\,,\ x = x_1 + (x_2 - x_1) \cdot \left(\frac{y - y_1}{y_2 - y1}\right),\ z = z_1 + (z_2 - z_1) \cdot \left(\frac{y - y_1}{y_2 - y1}\right) \quad (1)$$

$$d_{true} = \sqrt{x^2 + y^2 + z^2}\,,\ \theta_{b_{true}} = tan^{-1}\left(\frac{x}{z}\right),\ \theta_{e_{true}} = tan^{-1}\left(\frac{y}{\sqrt{x^2 + z^2}}\right) \quad (2)$$

where *height* is the calculated height of the hip from the ground.

4 Localization

We used Monte Carlo localization (MCL) to localize the robot. Our approach is described in [11, 1]. In MCL, the robot's belief of its current pose is represented by a set of particles, each of which is a hypothesis of a possible pose of the robot. Each particle is represented by $\langle h, p\rangle$ where $h = (x, y, \theta)$ is the particle's pose and p represents the probability that the particle's pose is the actual pose of the robot. The weighted distribution of the particle poses represents the overall belief of the robot's pose.

At each time step, the particles are updated based on the robot's actions and perceptions. The pose of each particle is moved according to odometry estimates of how far the robot has moved since the last update. The odometry updates take the form of $m = (x', y', \theta')$, where x' and y' are the distances the robot moved in the x and y directions in its own frame of reference and θ' is the angle that the robot has turned since the last time step.

After the odometry update, the probability of each particle is updated using the robot's perceptions. The probability of the particle is set to be $p(O|h)$, which is the likelihood of the robot obtaining the observations that it did if it were in the pose represented by that particle. The robot's observations at each time step are defined as a set O of observations $o = (l, d, \theta)$ to different landmarks, where l is the landmark that was seen, and d and θ are the the observed distance and angle to the landmark. For each observation o that the robot makes, the likelihood of the observation based on the particle's pose is calculated based on its similarity to the expected observation $\hat{o} = (\hat{l}, \hat{d}, \hat{\theta})$, where $\hat{d}$ and $\hat{\theta}$ are the the expected distance and angle to the landmark based on the particle's pose. The likelihood $p(O|h)$ is calculated as the product of the similarities of the observed and expected measurements using the following equations:

$$r_d = d - \hat{d} \tag{3}$$
$$s_d = e^{-r_d^2/\sigma_d^2} \tag{4}$$
$$r_\theta = \theta - \hat{\theta} \tag{5}$$
$$s_\theta = e^{-r_\theta^2/\sigma_\theta^2} \tag{6}$$
$$p(O|h) = s_d \cdot s_\theta \tag{7}$$

Here s_d is the similarity of the measured and observed distances and s_θ is the similarity of the measured and observed angles. The likelihood $p(O|h)$ is defined as the product of s_d and s_θ. Measurements are assumed to have Gaussian error and σ^2 represents the standard deviation of the measurement. The measurement variance affects how similar the observed and expected measurement must be to produce a high likelihood. For example, σ^2 is higher for distance measurements than angle measurements when using vision-based observations, which results in angles needing to be more similar than distances to achieve a similar likelihood. The measurement variance also differs depending on the type of landmark observed.

For observations of ambiguous landmarks, the specific landmark being seen must be determined to calculate the expected observation $\hat{o} = (\hat{l}, \hat{d}, \hat{\theta})$ for its likelihood calculations. With a set of ambiguous landmarks, the likelihood of each possible landmark is calculated and the landmark with the highest likelihood is assumed to be the seen landmark. The particle probability is then updated using this assumption.

Next the algorithm re-samples the particles. Re-sampling replaces lower probability particles with copies of particles with higher probabilities. The expected number of copies that a particle i will have after re-sampling is

$$n \times \frac{p_i}{\sum_{j=1}^{n} p_j} \tag{8}$$

where n is the number of particles and p_i is the probability of particle i. This step changes the distribution of the particles to increase the number of particles at the likely pose of the robot.

After re-sampling, new particles are injected into the algorithm through the use of re-seeding. Histories of landmark observations are kept and averaged over the last three seconds. When two or more landmarks observations exist in the history, likely poses of the robot are calculated using triangulation. Lower probability particles are replaced by new particles that are created with these poses [7].

The pose of each particle is then updated using a random walk where the magnitude of the particle's adjustment is inversely proportional to its probability. Each particle's pose h is updated by adding $w = (i, j, k)$ where (i, j, k) are defined as:

$$i = \textsc{max-distance} \cdot (1 - p) \cdot random(1) \tag{9}$$

$$j = \textsc{max-distance} \cdot (1 - p) \cdot random(1) \tag{10}$$

$$k = \textsc{max-angle} \cdot (1 - p) \cdot random(1) \tag{11}$$

The MAX-DISTANCE and MAX-ANGLE are parameters that are used to set the maximum distance and angle that the particle can be moved during a random walk and $random(1)$ is a random real number between 0 and 1. This process provides another way for particles to converge to the correct pose without re-sampling.

Finally, the localization algorithm returns an estimate of the pose of the robot based on an average of the particle poses weighted by their probability. Figure 7 shows an example of the robot pose and the particle locations. The algorithm also returns the standard deviation of the particle poses. The robot may take actions to improve its localization estimate when the standard deviation of the particle poses is high.

In addition to the standard use of MCL, in [1], we introduced enhancements incorporating negative information and line information. Negative information is used when an observation is expected but does not occur. If the robot is not seeing something that it expects to, then it is likely not where it thinks it is. When starting from a situation where particles are scattered widely, many particles can be eliminated even when no observations are seen because they expect to see a landmark. For each observation that is expected but not seen, the particle is updated based on the probability of not seeing a landmark that is within the robot's field of view. It is important to note that the robot can also miss observations that are within its view for reasons such as image blurring or occlusions, and these situations need to be considered when updating a particle's probability based on negative information. Our work on negative information is based on work by Hoffmann et al [2, 3]. We update particles from line observations by finding the nearest point on the observed line and the expected line. Then we do a normal observation update on the distance and heading to these points.

We used a 4 state Kalman filter to track the location of the ball. The Kalman filter tracked the location and velocity of the ball relative to the robot. Using our localization estimate we could then translate the ball's relative coordinates back to global coordinates. Our Kalman Filter was based on the 7 state Kalman filter tracking both the ball and the robot's pose used in [9].

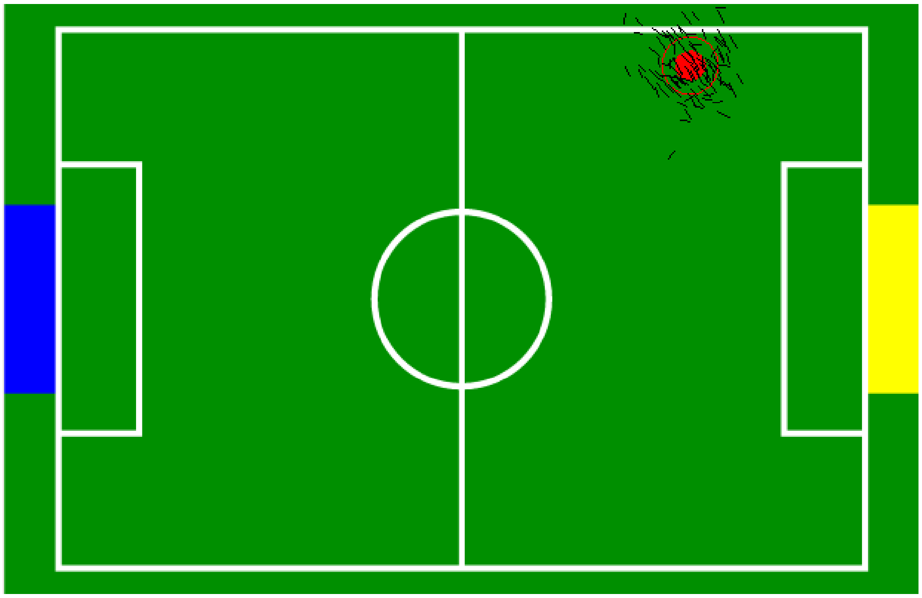

Figure 7: Example of Robot Pose Estimate and Particles

Our combined system of Monte Carlo localization for the robot and a Kalman filter to track the ball worked well and was robust to bad observations from vision. A video showing the performance of the localization algorithm while the keeper was standing in its goal is available at:
`http://www.cs.utexas.edu/users/AustinVilla/legged/teamReport08/naoParticleFilter.avi`.

5 Motion

We developed our own motion engine for the Nao, based on the state-based method of Yin et al [15]. In their approach, they create a simple finite state machine. Each state represents some stage of the motion with a set of target angles for the joints. Between states, motors are driven with proportional-derivative (PD) controllers to drive the motors to the desired angles. Transitions between states can occur based on time or foot contact with the ground. In addition, they dynamically balance the robot during the walk by leaving the swing hip angle free. The desired target for this angle is calculated based on the current and desired torso and opposite hip angles. This enables the walk engine to change the swing hip angle to dynamically balance the robot. We used their approach as the basis for both our walking and kicking motions.

5.1 Walking

While we eventually decided on using the Aldebaran motion engine for the RoboCup competition, we developed our own walk engine for the Nao, which is described below. Our walk engine was a four state finite state machine, demonstrated in Figure 8. In the first state, the robot shifts its weight to its stance leg and lifts its swing leg up and forward. In the next state, it brings the

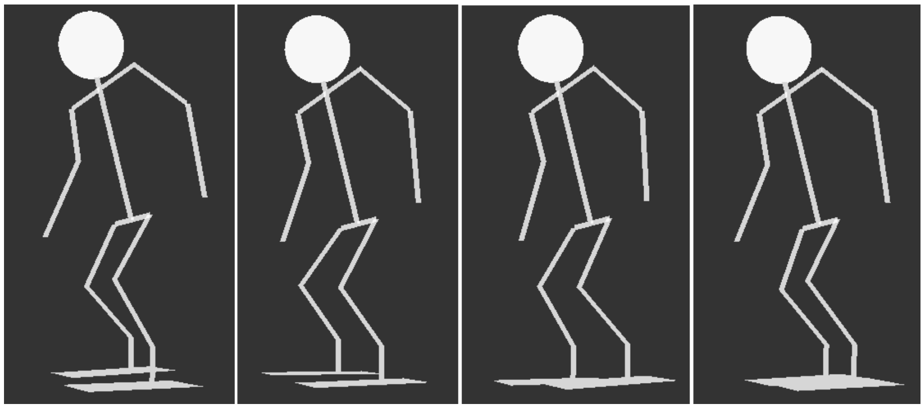

Figure 8: This figures shows the states in our walk engine. In the first state, the robot shifts its weight to its left leg and lifts its right leg. In the second state, it rebalances its weight and brings the right leg forward and down. In states 3 and 4, the robot repeats this motion with the legs reversed.

Target	Type
Swing Leg Forward	mm
Swing Leg Side	mm
Swing Leg Up	mm
Stance Leg Forward	mm
Stance Leg Side	mm
Stance Leg Up	mm
Swing Shoulder	Rad
Stance Shoulder	Rad

Table 1: State Targets

swing leg forward and down. These two states are then repeated with the legs switched. Each state is characterized by the targets listed in Table 1. The first six targets are used to calculate desired angles for all of the leg joints, while the last two targets are used directly as the targets for the arm joints.

In our walk engine, the target parameters for the swing and stance legs are described by the relative distances of each foot from the center of the hips. The distances down, forward, and sideways from the hip center to each foot are set in millimeters. We then use inverse kinematics to calculate the desired hip pitch, hip roll, and knee pitch angles from these targets. The ankle roll and ankle pitch angles are calculated to keep the ankles parallel to the ground at all times. The ankle pitch angles are also used to maintain the robot's torso at a desired angle for dynamic balancing. When turning, the hip yaw pitch angle is set to turn the robot's legs. In addition, the robot's shoulder joints are moved to help balance the robot during walking.

A walk in our walk engine is characterized by the parameters in Table 2, which also shows the best parameters that we found for our walk engine through machine learning in simulation and trial and error on the physical robot. Table 3 shows how these 10 parameters are used to determine the target parameters for each state.

Parameter	Type	Value
Step Time	Seconds	1.5
Foot Forward	mm	-10
Step Length	mm	65
Walk Height	mm	160
Step Height	mm	15
Shift	mm	40
Torso Angle	Rad	15
Leg Lengthen Ratio	%	15
Shoulder Forward	Rad	60
Shoulder Back	Rad	110

Table 2: Walk Parameters

Target	State 1	State 2
SwingLegForward	FootForward	FootForward - StepLength / 2.0
SwingLegUp	WalkHeight - (1 - LegLengthenRatio) * StepHeight	WalkHeight
SwingLegSide	Shift	Shift
StanceLegForward	FootForward	FootForward + StepLength / 2.0
StanceLegUp	WalkHeight + LegLengthenRatio * StepHeight	WalkHeight
StanceLegSide	Shift	Shift
SwingShoulder	(ShoulderForward + ShoulderBack) / 2.0	ShoulderBack
StanceShoulder	(ShoulderForward + ShoulderBack) / 2.0	ShoulderForward

Table 3: Walk Equations

At standing, the feet will be $FootForward$ in front of the hips and $WalkHeight$ below the hips. During the walk, the feet will slide $Shift$ to the side from the hips to move the weight onto the stance leg. The robot takes steps of length *StepLength* with the hips exactly halfway between the forward and back feet. When stepping, the swing foot will be *StepHeight* millimeters above the stance foot. The step height is created both by straightening the stance and shortening the swing leg. The stance leg is lengthened by *LegLengthenRatio* of the step height and the swing leg is shortened by the remaining amount.

We performed machine learning on the simulated robot in the Webots simulator[4] to learn the best parameters for our walk engine, based on the walk learning approach used on the Aibo by Kohl and Stone [6, 5]. We then used these learned parameters as a starting point to find a good walk on the physical robot. We used the Downhill Simplex algorithm [8] to learn the best walk parameters. On each trial, the robot's walk engine was initialized with the parameters given to it by the algorithm to be tested. The robot was placed 1.8 meters from the ball in the simulator. It then tried to walk for 1200 frames or until it fell or came within 800 centimeters of the ball (based on vision estimates). The evaluated walk was given a value of $2000 - NumFrames$ if the robot fell and $NumFrames/(1800 - mmTraveled)$ otherwise. This evaluation function gave lower values for walks that went farther and high values for trials were the robot fell. The Downhill Simplex algorithm tried to find the parameters that minimized this function.

Our walk engine was preferable in many respects to the walk provided by Aldebaran. The direction and speed of the walk could be modified dynamically, while the Aldebaran walk engine required the user to provide a target distance or

[4] http://www.cyberbotics.com/

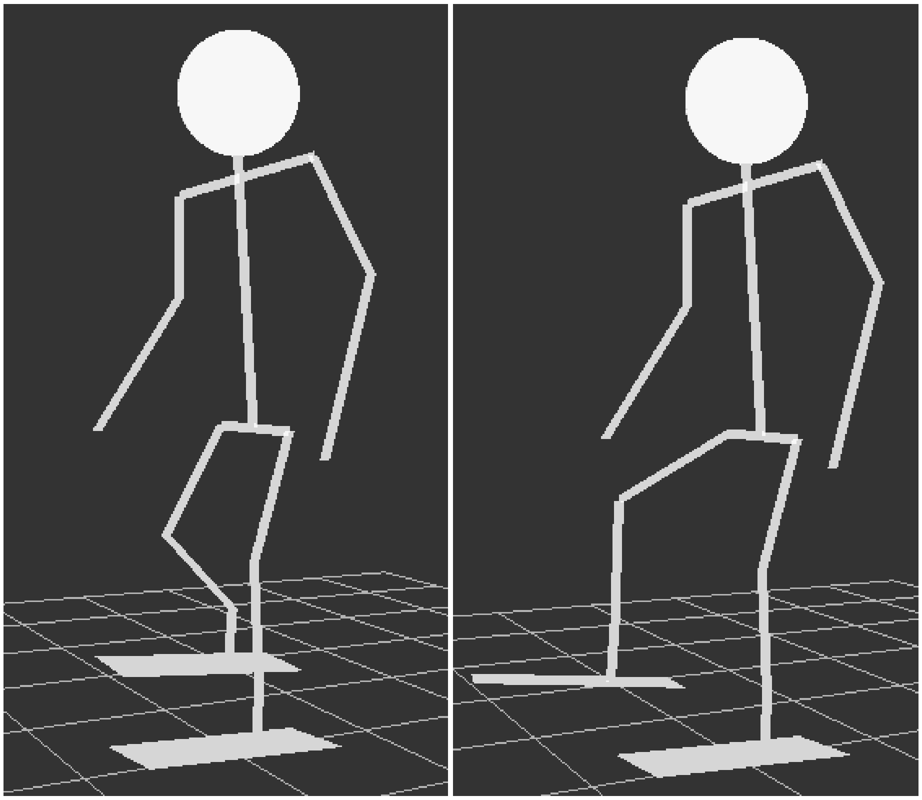

Figure 9: This figure shows a model of a robot in Webots kicking the ball. State 2, where the robot brings its leg back, is shown on the left. State 3, where the robot swings its leg forward, is on the right.

turn angle and wait for the robot to complete it. Our walk engine's parameters could also be adjusted on the fly. One useful example of this is its ability to increase the angle of the torso as the robot approached the ball so that the robot could continue to see it. In spite of these benefits, we chose to use the Aldebaran walk engine because it was much more stable than our own walk engine. Our walk was less stable than the Aldebaran walk because of inherent jerkiness in the robot's motion caused by using the ALMotion interface instead of DCM. We believe with a few modifications our walk engine will be more stable and we expect to use it in the 2009 RoboCup competition.

5.2 Kicking

We approached the problem of kicking the ball in a similar manner to our walk. Once again, we created a state machine for the robot to kick. In the first state, the robot shifts it weight to the stance leg. Figure 9 shows the second state and third states of the kick engine. In the second state, the robot lifts the kicking leg and brings it back. In the third state, the robot kicks the ball by swinging the kicking leg forward through the ball. In the final state, the robot brings the legs even again and balances its weight back onto them. This kick was characterized

Parameter	Type	Value
Body Height	mm	190
Feet Forward	mm	19
Shift	mm	50
Kick Height	mm	35
Swing Back	mm	32
Swing Fwd	mm	85
Torso Angle	Rad	8

Table 4: Kick Parameters

by the parameters shown in Table 4. Once again, the target x, y, z coordinates of the feet relative of the hips were used to calculate the desired joint angles for the legs. In each state, the leg angles were driven to these target angles using PD controllers.

We used the Downhill Simplex algorithm [8] to learn the best kick parameters in simulation, similar to our approach for learning the walk parameters. In each trial, the robot's kick engine was initialized with the parameters to be tested from the algorithm. The robot was started with the ball near its right foot. Then the robot executed its kick based on the given parameters. The trial continued until the robot fell or 300 frames had passed. The kick was given a value of $2000 - NumFrames$ if the robot fell, 1000 if it could not see the ball, and $(6000 - BallDistance)/6 + BallBearing$ otherwise. This function gives the lowest value for a ball that was kicked very far and very straight. The parameters that were learned are shown in Table 4. These parameters appeared to be the best parameters on the physical robot as well, as the robot was able to kick the ball several meters.

5.3 Kinematics

To determine the pose of the robot we used the accelerometers to estimate the roll (ϕ), pitch (θ) and yaw (ψ) of the torso. As expected the accelerometers were fairly noisy; to overcome this we used a Kalman Filter to produce a smoother set of values (ϕ_f, θ_f, ψ_f). The forward kinematics were then calculated by using the modified Denavit and Hartenberg parameters with the XYZ axes rotated to reflect ϕ_f,θ_f and ψ_f. Figure 10 shows the filtered versus unfiltered estimates of the robots pose during a typical walking cycle. A video can also found at:
`http://www.cs.utexas.edu/users/AustinVilla/legged/teamReport08/naoKinematics.avi.`

6 Behavior

Our behavior module is made up of a hierarchy of task calls. We call a PlaySoccer task which then calls a task based on the mode of the robot {ready, set, playing, penalized, finished}. These tasks then call sub-tasks such as ChaseBall or GoToPosition.

Our task hierarchy is designed for easy use and debugging. Each task maintains a set of state variables, including the sub-task that it called in the previous frame. In each frame, a task can call a new sub-task or continue running its

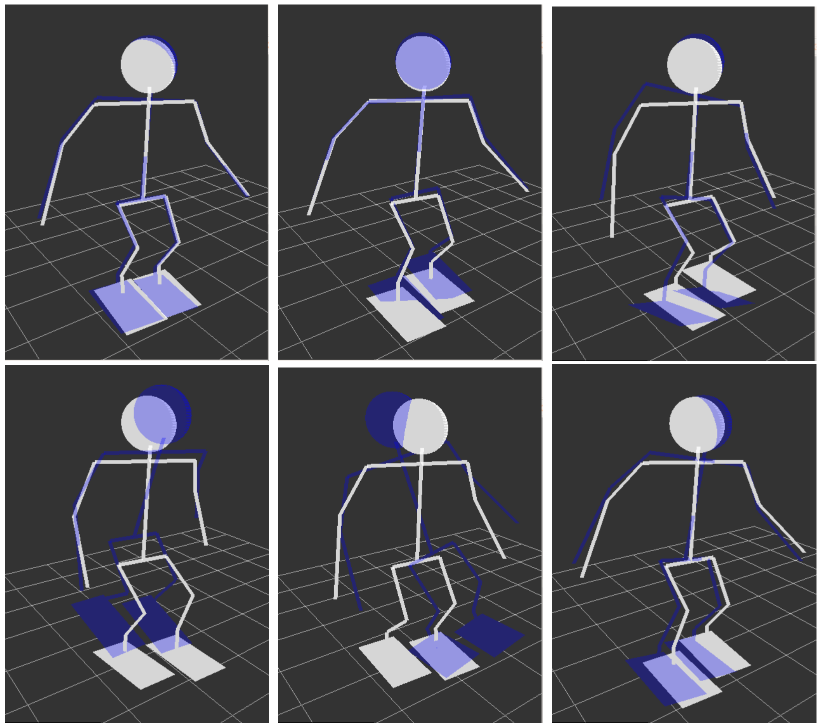

Figure 10: Estimations of the robots pose while walking forwards. The blue robot is the unfiltered pose while the white robot is the filtered pose.

previous one. If it does not explicitly return a new sub-task, its previous sub-task will be run by default. Tasks at the bottom of the hierarchy are typically the ones that send motor commands to the robot; for example telling it to walk, kick, or move its head.

Tasks in our system can also call multiple tasks in parallel. This ability is used mainly to allow separate tasks to run for vision and motion on the robot. While the robot is running a vision behavior such as a head scan or looking at an object, the body can be controlled by a separate behavior such as kicking or walking towards the ball.

One of the benefits of our task hierarchy is its debugability. In addition to the logs of memory that the robot creates, it also creates a text log that displays the entire tree of tasks that were called each frame along with all their state variables. Figure 11 shows an example of one frame of output of the behavior log in the tool. The information provided in the text log is enough to determine why the robot chose each particular behavior, making it easy to debug. In addition, this text log is synchronized with the memory log in the tool, allowing us to correlate the robot's behaviors with information from vision, localization, and motion.

The behaviors that were used in the RoboCup competition were fairly simple. In its most basic form, the robot tried to walk to the ball and kick it towards the goal. Kicking the ball was somewhat complicated, as the robot could not see the ball near its feet due to the limited field of view of its camera. Our

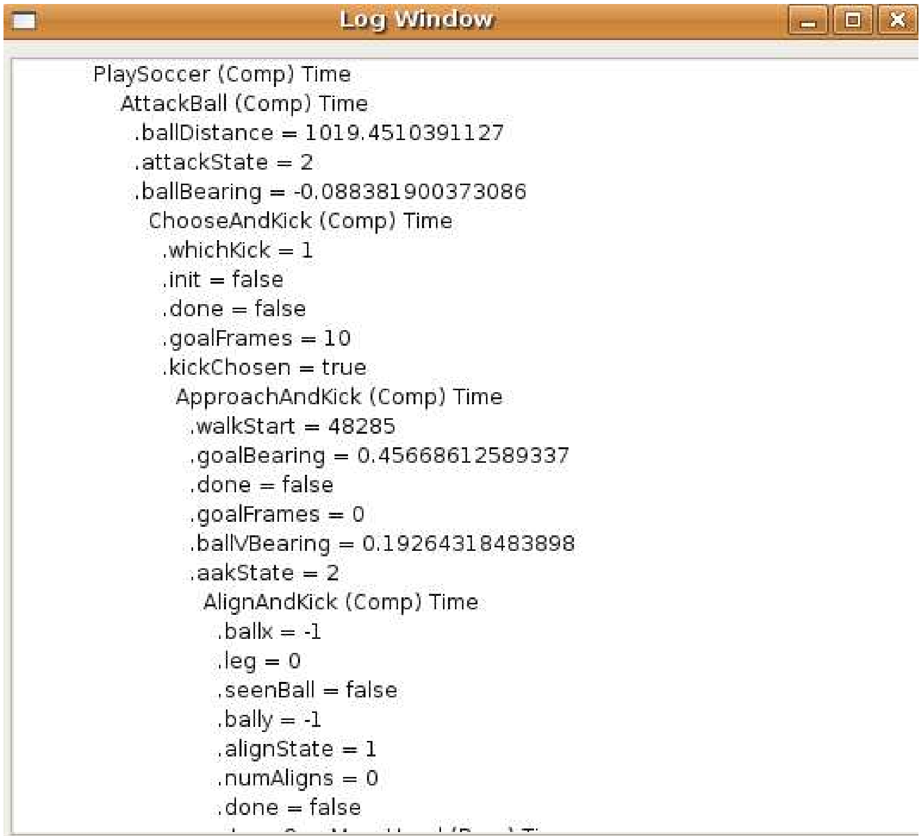

Figure 11: Example Behavior Log, showing the trace of task calls and their state variables.

kicking behavior consisted of walking up to the ball, adjusting to face the goal, leaning over to see if the ball was there, aligning to the ball, and then kicking. A video of this behavior is available online at:

`http://www.cs.utexas.edu/users/AustinVilla/legged/teamReport08/naoBehavior.avi`

7 Competition

The 12th International Robot Soccer Competition (RoboCup) was held in July 2008 in Suzhou, China[5]. 15 teams entered the competition. Games were played with two robots on a team. The first round consisted of a round robin with three groups of four teams and one group of three teams. Each team played each of the other teams in its group. The top two teams from each group advanced to the quarterfinals. From the quarterfinals on, the winner of each game advanced to the next round.

UT Austin Villa was in round robin group D with Northern Bites and Zadeat. In the round robin, games that ended on a tie were decided by which team

[5]http://robocup-cn.org/

displayed more skills (ability to walk, ability to kick, ability to play soccer). Based on these criteria, UT Austin Villa was deemed to be the best team in the group and moved on to the quarterfinals. In the quarterfinals, UT Austin Villa played Kouretes. The game ended in a tie, and Kouretes won the game in overtime on penalty kicks. Although we did not perform as well as we hoped, we were happy with the progress that was made in the short time we had the robots before the competition.

8 Conclusion

This report described the technical work done by the UT Austin Villa team for its Nao entry in the Standard Platform League. Our team developed a new codebase, with a new software architecture at its core. The architecture consisted of many modules communicating through a shared memory system. This setup allowed for easy debugability, as the shared memory could be saved to a file and replayed later for debugging purposes. The Nao code included vision and localization modules based on previous work. The team developed new algorithms for motion and kinematics on a two-legged robot. Finally, we developed new behaviors for use on the Nao.

The work presented in this report gives our team a good foundation on which to build better modules and behaviors for future competitions. In particular, our modular software architecture provides us with the ability to easily swap in new modules to replace current ones, while still maintaining easy debugability. While we were disappointed in our result at RoboCup 2008, we believe this work gives us a good start towards competing successfully in future RoboCup competitions.

References

[1] T. Hester and P. Stone. Negative information and line observations for Monte Carlo localization. In *IEEE International Conference on Robotics and Automation (ICRA)*, May 2008.

[2] J. Hoffmann, M. Spranger, D. Göhring, and M. Jüngel. Exploiting the unexpected: Negative evidence modeling and proprioceptive motion modeling for improved markov localization. In *RoboCup*, pages 24–35, 2005.

[3] J. Hoffmann, M. Spranger, D. Göhring, and M. Jüngel. Making use of what you don't see: Negative information in markov localization. In *IEEE/RSJ International Conference of Intelligent Robots and Systems*, 2005.

[4] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. RoboCup: The robot world cup initiative. In *Proceedings of The First International Conference on Autonomous Agents*. ACM Press, 1997.

[5] N. Kohl and P. Stone. Machine learning for fast quadrupedal locomotion. In *The Nineteenth National Conference on Artificial Intelligence*, pages 611–616, July 2004.

[6] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Proceedings of the IEEE International Conference on Robotics and Automation*, May 2004.

[7] S. Lenser and M. Veloso. Sensor resetting localization for poorly modelled mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

[8] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, UK, 2nd edition, 1992.

[9] M. J. Quinlan, S. P. Nicklin, N. Henderson, R. Fisher, F. Knorn, S. K. Chalup, R. H. Middleton, and R. King. The 2006 NUbots Team Report. Technical report, School of Electrical Engineering & Computer Science Technical Report, The University of Newcastle, Australia, 2007.

[10] C. J. Seysener, C. L. Murch, and R. H. Middleton. Extensions to object recognition in the four-legged league. In D. Nardi, M. Riedmiller, and C. Sammut, editors, *Proceedings of the RoboCup 2004 Symposium*, LNCS. Springer, 2004.

[11] M. Sridharan, G. Kuhlmann, and P. Stone. Practical vision-based monte carlo localization on a legged robot. In *IEEE International Conference on Robotics and Automation*, April 2005.

[12] P. Stone, K. Dresner, P. Fidelman, N. K. Jong, N. Kohl, G. Kuhlmann, M. Sridharan, and D. Stronger. The UT Austin Villa 2004 RoboCup four-legged team: Coming of age. Technical Report UT-AI-TR-04-313, The University of Texas at Austin, Department of Computer Sciences, AI Laboratory, October 2004.

[13] P. Stone, K. Dresner, P. Fidelman, N. Kohl, G. Kuhlmann, M. Sridharan, and D. Stronger. The UT Austin Villa 2005 RoboCup four-legged team. Technical Report UT-AI-TR-05-325, The University of Texas at Austin, Department of Computer Sciences, AI Laboratory, November 2005.

[14] P. Stone, P. Fidelman, N. Kohl, G. Kuhlmann, T. Mericli, M. Sridharan, and S. en Yu. The UT Austin Villa 2006 RoboCup four-legged team. Technical Report UT-AI-TR-06-337, The University of Texas at Austin, Department of Computer Sciences, AI Laboratory, December 2006.

[15] K. Yin, K. Loken, and M. van de Panne. Simbicon: Simple biped locomotion control. *ACM Trans. Graph.*, 26(3):Article 105, 2007.