TT-UT Austin Villa 2009: Naos Across Texas

Todd Hester, Michael Quinlan and Peter Stone
Department of Computer Science
The University of Texas at Austin
1 University Station C0500
Austin, Texas 78712-0233
{todd,mquinlan,pstone}@cs.utexas.edu

Mohan Sridharan
Department of Computer Science
Texas Tech University
P.O. Box 43104
Lubbock, TX 79409-3104
mohan.sridharan@ttu.edu

Technical Report UT-AI-TR-09-08

December 24, 2009

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. In 2009, UT Austin combined forces with Texas Tech University, to form TT-UT Austin Villa. TT-UT Austin Villa won the 2009 US Open and placed fourth in the 2009 RoboCup competition in Graz, Austria. This report describes the algorithms used in these tournaments, including the architecture, vision, motion, localization, and behaviors.

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 [6].

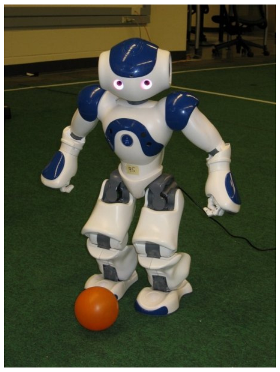


Figure 1: The Aldebaran Nao robot.

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]. All teams in the SPL compete with identical robots, making it essentially a software competition. All teams use identical humanoid robots from Aldebaran called the Nao[2], shown in Figure 1.

Our team is TT-UT Austin Villa[3], from the Department of Computer Science at The University of Texas at Austin and the Department of Computer Science at Texas Tech University. Our team is made up of Professor Peter Stone, PhD student Todd Hester, and postdoc Michael Quinlan from UT Austin, and Professor Mohan Sridharan from TTU. All members are 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 [11, 12, 13] 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. Descriptions of the work for the 2008 tournament can be found in the corresponding technical report [2]. We continued our work after RoboCup 2008, making progress towards the US Open and RoboCup in 2009.

This report describes all facets of our development of the Nao team codebase. For completeness, this report repeats portions of the 2008 team report [2]. The main changes are in vision, localization, motion, and behavior. 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 [13] and are described in sections 3 and 4 respectively. Section 5 describes our motion modules used on the robot. 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.

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

2 Software Architecture

Though based in spirit on our past software architectures used for the Aibos, the introduction of the Nao prompted 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.

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.

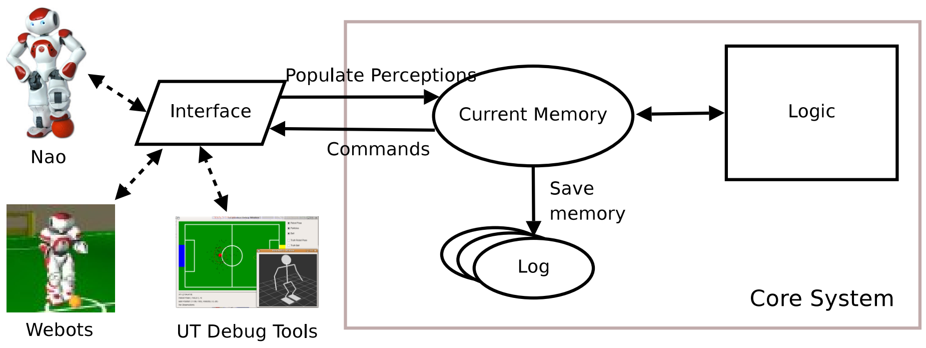


Figure 2: Overview of the TT-UT Austin Villa software architecture.

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 in our debug tool. As a result, we can test and debug code in any of the 3 environments without 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. The simulation interface also communicates with NaoQi. The tool interface can populate memory from 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.

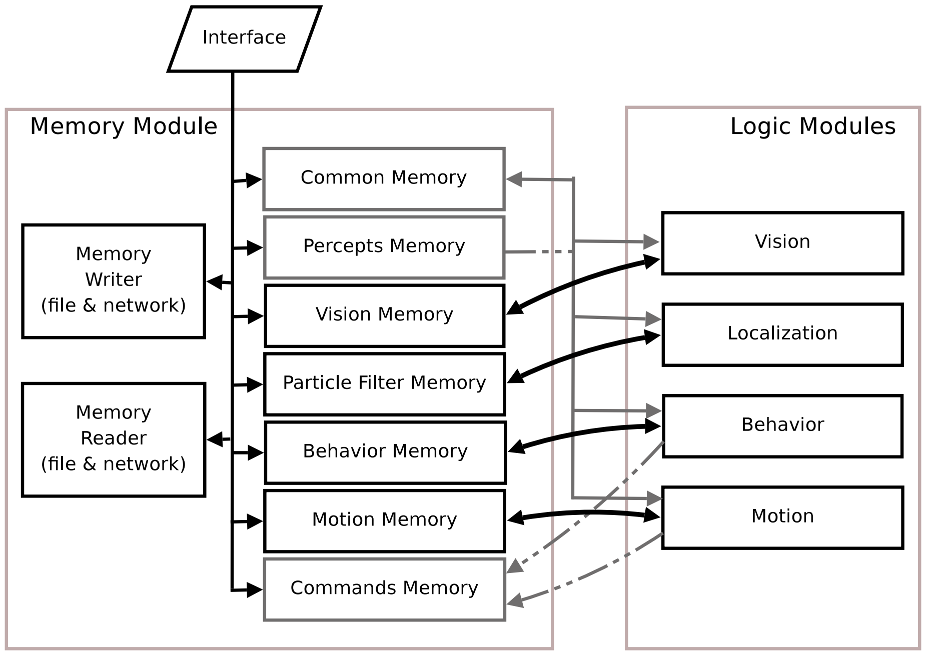


Figure 3: Design of the Memory module. The gray boxes indicated memory blocks that are accessed by multiple logic modules. Dotted lines show connections that are either read or write only (percepts are read only, commands are write only).

,

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

We used the 160×120 pixel image for vision processing. Since the number of pixels was relatively low we were able to use same image processing techniques that were applied by most AIBO teams [13, 8].

First, the YUV image is segmented into known colors. Second, contiguous image regions of the same color are grouped into blobs. Finally, these blobs are examined to see if they contain one of the target objects (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.*

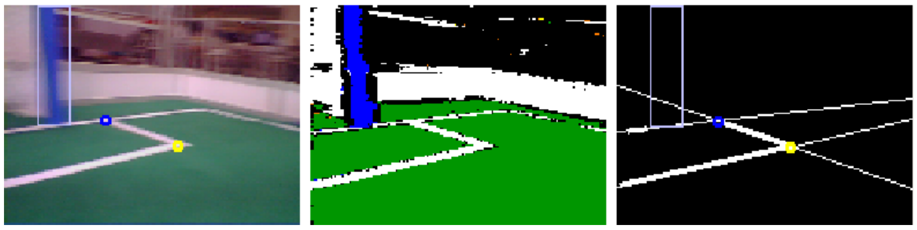


Figure 4: Example of the vision system. Left-to-Right: The raw YUV image, the segmented image and the observed objects

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 [9]. 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.

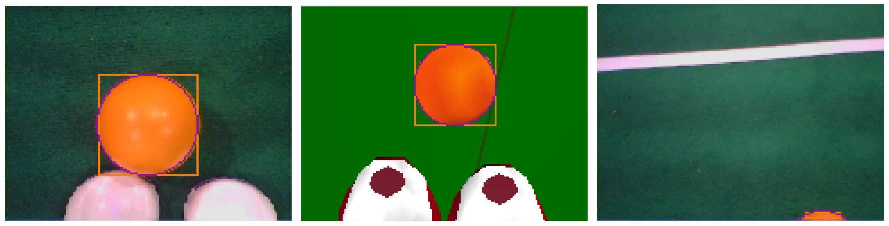


Figure 5: Example of circle fitting applied to the ball. The first two images a examples of a ball when the robot is preparing to kick (an image is presented from the robot and from the simulator). The third image shows a ball where circle fitting is required to get an accurate estimate of distance.

3.1 Goals and Posts

This year we developed new goal and goal post detection code to detect goal posts more reliably and more accurately. First, when viewing posts that were rotated, we used the rotated width of the post to get better distance estimates. Second, when seeing a single post, we used the cross bar and field lines to determine which post it was. Figure 6 shows the detection of left, right, and unknown goal posts.

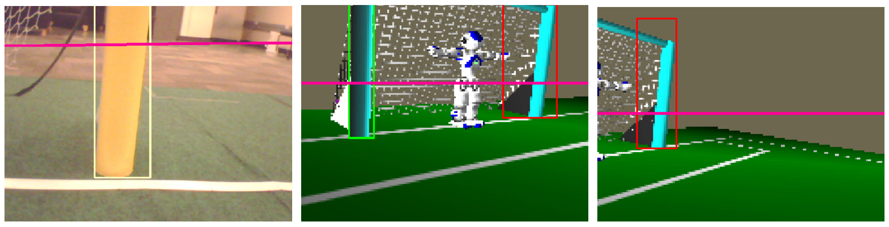
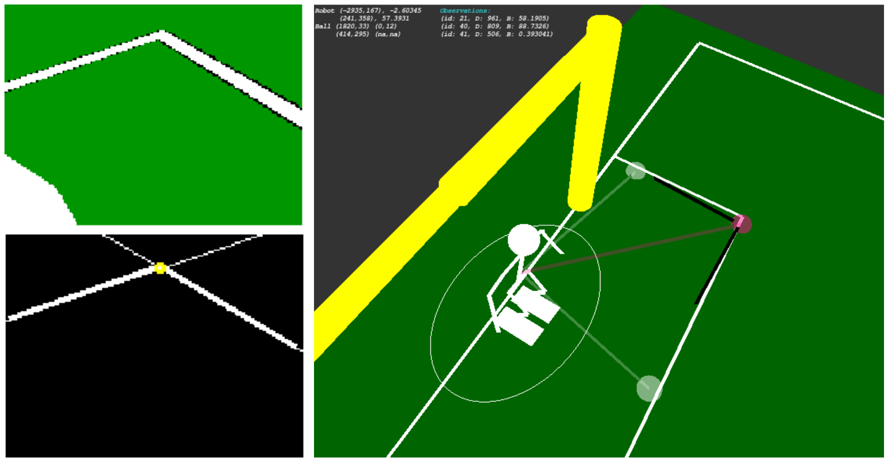


Figure 6: Example of goal post detection. In the first image, there is no information from the lines or crossbar to determine which goal post it is, so it is defined as an unknown yellow post (yellow bounding box). In the second image, both posts can be seen and are labeled as left and right post (green for left post and red for right post). In the third image, part of the crossbar is seen and can be used to define the post as a right goal post.

3.2 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 7) 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.

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 3.2.1. 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 7.



```
Robot (-2935,167), -2.60345      Observations:
      (241,358), 57.3931         (id: 21, D: 961, B: 58.1905)
Ball (1820,33) (0,12)            (id: 40, D: 809, B: 88.7326)
     (414,295) (na,na)           (id: 41, D: 506, B: 0.393041)
```

Figure 7: 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.

3.2.1 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 [10, 3]. 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. When only one landmark has been seen, the re-seeding creates particles in an arc around that landmark. In either case, the probabilities of the particles are calculated and then lower probability particles are replaced by the 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:

$$i = \text{MAX-DISTANCE} \cdot (1 - p) \cdot random(1) \tag{9}$$
$$j = \text{MAX-DISTANCE} \cdot (1 - p) \cdot random(1) \tag{10}$$
$$k = \text{MAX-ANGLE} \cdot (1 - p) \cdot random(1) \tag{11}$$

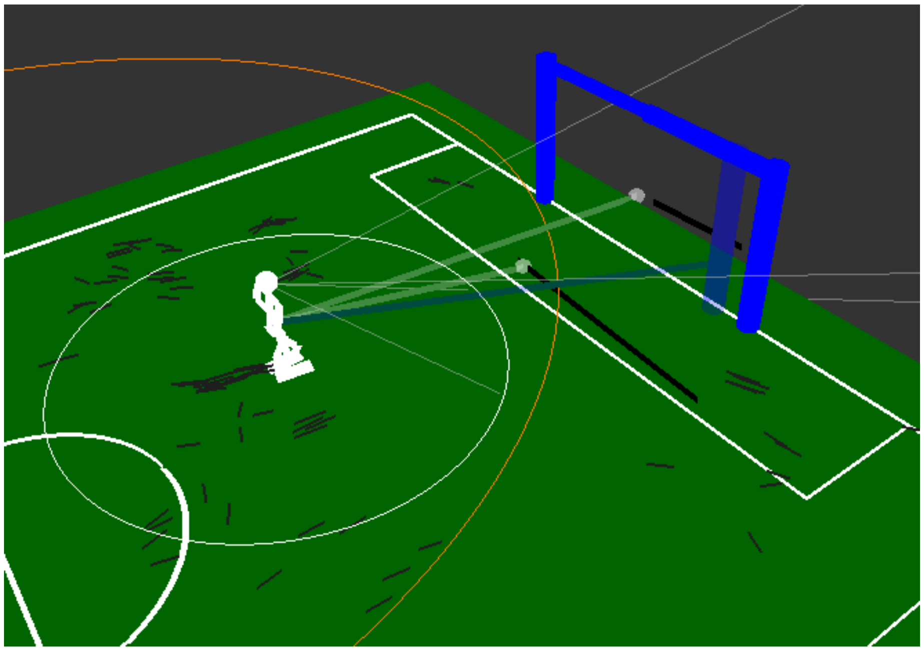


Figure 8: Example of Robot Pose Estimate and Particles. Here the robot has seen a blue goal post and two field lines. The particles are the small black lines that are scattered around the robot, with their length representing their probability. The robot's pose estimate is at the weighted mean of the particle locations, with the standard deviation of the particles shown by the white circle around the robot.

The MAX-DISTANCE and MAX-ANGLE parameters 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 8 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 [3], 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

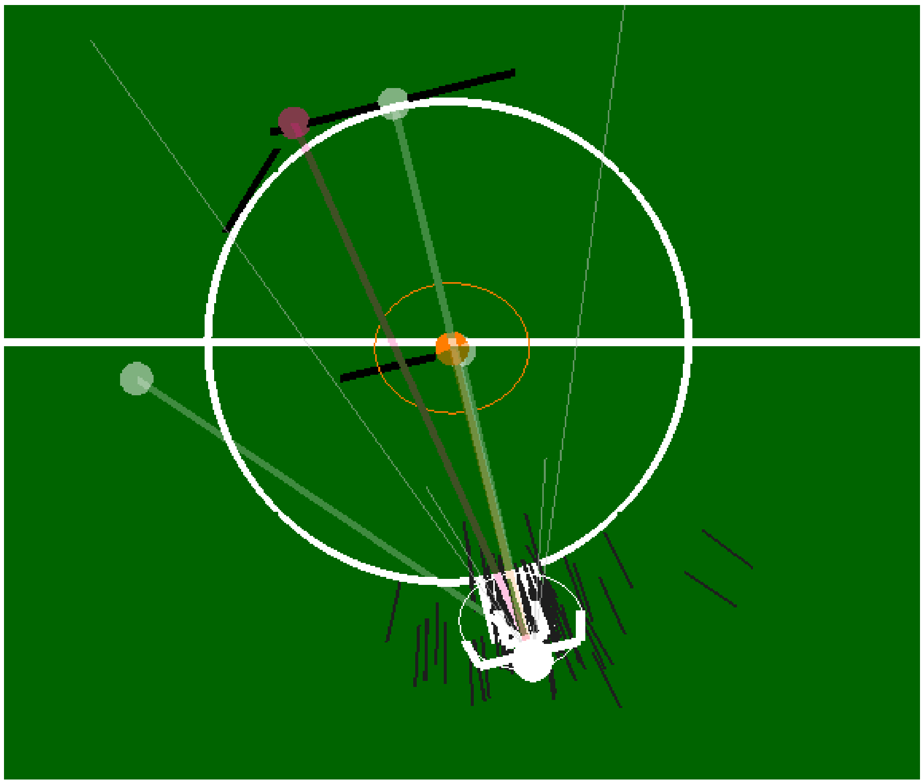


Figure 9: Example of center circle tangent lines. Vision has detected two tangent lines to the circle, as well as the center line. The particle filter is able to match these tangent lines to the circle to provide a good localization estimate.

probability based on negative information. Our work on negative information is based on work by Hoffmann et al [4, 5].

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 bearing to these points. We also use this method when seeing the center circle on the field. Vision reports the center circle as a set of tangent lines to the circle. For each particle and observed line, localization finds the two tangent lines at the angle of the observed line. It then does localization updates comparing the observed line with these two lines as well as all the other field lines on the field. It uses the nearest points on these lines and the observed lines as it does for normal lines. The matching of tangent lines to the field circle is shown in Figure 9.

We used a four-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 seven-state Kalman filter tracking both the ball and the robot's pose used in [8].

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

Parameter	Straight	Turning	Sideways
Step Length	0.06 m	0.05 m	0.05 m
Step Height	0.018 m	0.02 m	0.02 m
Step Side	0.04 m	0.02 m	0.05 m
Max Turn	0.19 deg	0.3 deg	0.3 deg
Zmp Offset X	0.02 m	0.015 m	0.015 m
Zmp Offset Y	0.02 m	0.018 m	0.018 m
LHipRoll	1 deg	2.5 deg	4.5 deg
RHipRoll	-1 deg	-25. deg	-4.5 deg
Hip Height	0.2 m	0.2 m	0.2 m
Torso Orient	9.0 deg	8.0 deg	6.0 deg
Samples Per Step	26	24	26

Table 1: Parameters for the Aldebaran walk engine based on walk direction.

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 used the provided Aldebaran walk engine for our walk, with carefully tuned walk parameters and joint stiffnesses. We also developed our own kick engine, allowing us to use multiple kicks in different directions. We have also developed our own kinematics models for use in the kick engine.

5.1 Walking

We use the walk engine provided by Aldebaran in the ALMotion module. This walk engine is a ZMP based walk, which then controls the robots center of mass to follow the pattern generated by the ZMP walk engine. There are 10 parameters for the walk engine, plus the number of samples per step for the walk. In addition to these parameters, we also adjusted the stiffnesses of the joints of the robot to make it more compliant in its walking. We used different parameters for the robot when walking straight, turning, or walking sideways. The parameters we used for walking are shown in Table 1. The parameters were tuned through trial and error, with an emphasis on stability and speed.

5.2 Kicking

For this year's RoboCup, we developed a general kick engine. Each kick is defined by a number of states, and each state is defined by seven parameters: the distance from the ankle to the hip in the x, y, and z coordinates for each leg, and the angle for the HipYawPitch joint. There is also a flag defining for each state which one is the *kick* state. Our engine uses inverse kinematics (Section 5.3.2) to create target joint angles for the given ankle-hip distances for each state. The robot is then told to move its legs to these positions through an interpolate command lasting 0.5 seconds. After the completion of each state, the robot continues to the next one. For the kick phase, the command is instead given with the interpolate with speed command, with a speed varying depending on the desired strength of the kick.

We used this general framework to develop multiple kicks, including a forward kick, a backward kick, an inside-out side kick, an outside-in side kick, and a 45 degree kick using the inside of the foot. The forward kick is a typical one, and incorporates the following 8 states: (0) stand, (1) shift weight to stance leg, (2) lift kicking leg and swing back, (3) align leg to ball, (4) swing leg through ball, (5) bring leg back, (6) bring leg down, (7) shift weight back.

When we first approach a ball, the robot chooses a kick from its engine based on the desired heading of the ball. It then chooses the kick that will provide the closest heading to the desired one. For example, if the robot desires a ball heading of 80 degrees, it will choose to do a side kick, which kicks the ball at approximate 90 degrees. Next the robot does some adjustment steps if the ball is not in a plausible location for that kick. The robot then decides which leg to use based on the ball location. Finally, it executes the kick engine, moving through the state machine described above.

5.3 Kinematics

5.3.1 Forward 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-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`.

5.3.2 Inverse Kinematics

For 2009 we implemented a Jacobian approach for solving the inverse kinematics problem. Primarily this was developed to allow the proper use of the 'groin' joint while kicking, in particular the 45 degree kick employed at RoboCup relied heavily on this joint.

The developed technique was an iterative least squares method, and terminated when the end effector position was inside a minimum acceptable error or we had performed too many iterations. In practice we could reach an acceptable error of 0.5 mm in only a few iterations. While this was sufficient for 2009 as we were only using it for kicking, a faster method maybe required in 2010 for computing joint angles at 50 Hz. It appears that a closed form solution can be implemented and should be more efficient [1].

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.

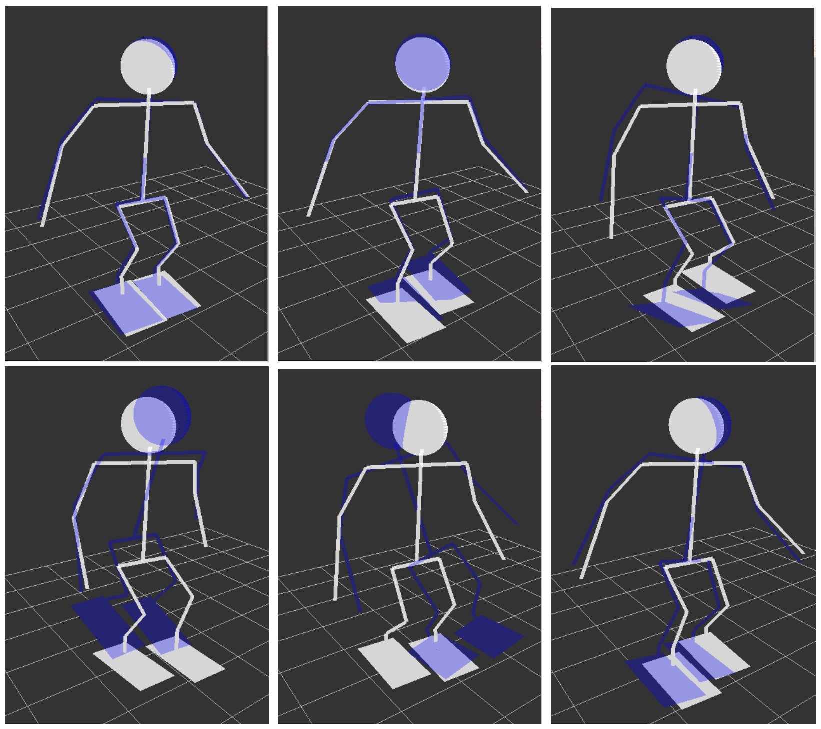


Figure 10: Estimations of the robots pose while walking forwards. The blue robot is the unfiltered pose while the white robot is the filtered pose.

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 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 emphasis in our team behaviors was to get to the ball quickly and be

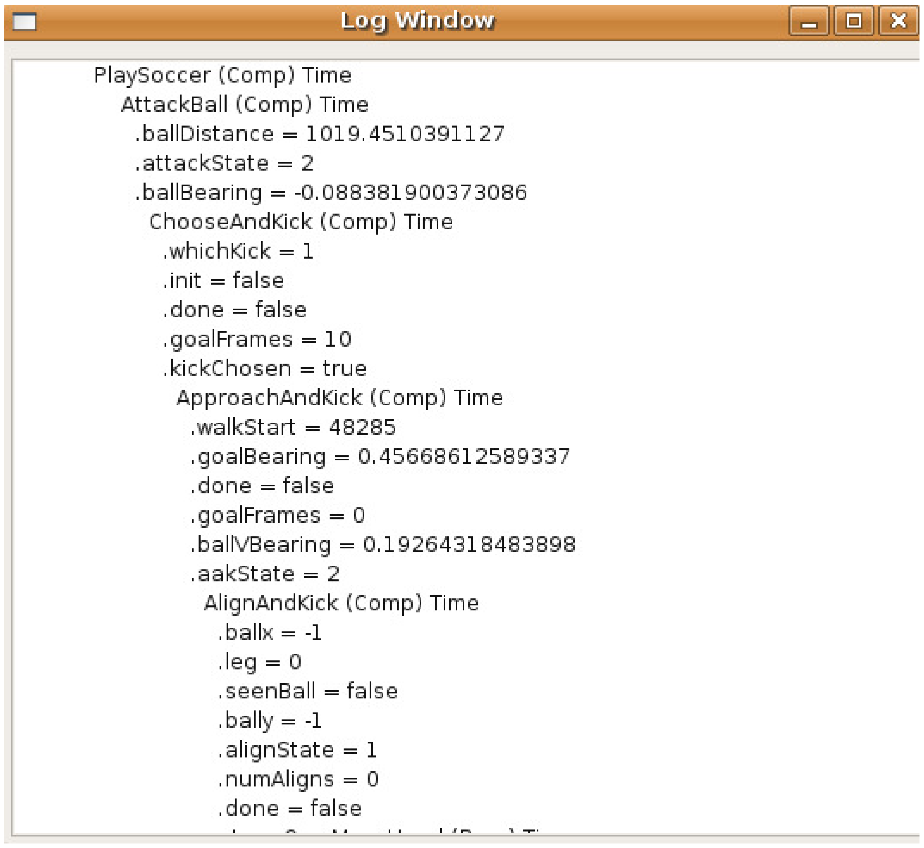

```
Log Window
        PlaySoccer (Comp) Time
          AttackBall (Comp) Time
           .ballDistance = 1019.4510391127
           .attackState = 2
           .ballBearing = -0.088381900373086
            ChooseAndKick (Comp) Time
             .whichKick = 1
             .init = false
             .done = false
             .goalFrames = 10
             .kickChosen = true
              ApproachAndKick (Comp) Time
               .walkStart = 48285
               .goalBearing = 0.45668612589337
               .done = false
               .goalFrames = 0
               .ballVBearing = 0.19264318483898
               .aakState = 2
                AlignAndKick (Comp) Time
                 .ballx = -1
                 .leg = 0
                 .seenBall = false
                 .bally = -1
                 .alignState = 1
                 .numAligns = 0
                 .done = false
```

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

sure to do the right thing once we got to it. First, the robot would search for the ball, scanning its head, then spinning in a circle, and finally walking to center of the field if it could not find the ball. Once the ball was found, the robot would walk to the ball as quickly as possible. Once the robot got to the ball, it would look up and scan the field to localize itself. Based on the bearing of the opponent's goal, the robot would decide which kick to do (forward kick, side kick, etc). Then the robot would check the ball's position and make small steps to adjust to it if required. Finally it would kick the ball and then continue chasing it. This behavior can be seen in our team highlight videos at:
`http://www.cs.utexas.edu/ AustinVilla/?p=competitions/RoboCup09`
`http://www.cs.utexas.edu/ AustinVilla/?p=competitions/US09`

7 Competition

In 2009, TT-UT Austin Villa competed in two tournaments, the US Open and RoboCup 2009 in Graz, Austria. Our results in each are described below.

Round	Opponent	Score
Round Robin	UPennalizers	3-0
Round Robin	CMWright Eagle	3-0
Round Robin	Northern Bites	2-0
Semifinal	CMWright Eagle	2-0
Championship	UPennalizers	1-1 (3-2 penalty kicks)

Table 2: US Open 2009 Results

7.1 US Open

The 2009 Standard Platform League US Open was held at Bowdoin College in Brunswick, Maine from May 1-3, 2009. There were four participants: TT-UT Austin Villa, Northern Bites, CMWright Eagle, and the UPennalizers. The tournament started with a round robin between all the teams on Saturday to determine seeding, followed by a four team elimination tournament on Sunday. In the round robin, TT-UT Austin Villa beat each of the other three teams, scoring 8 goals while allowing none. TT-UT Austin Villa faced CMWright Eagle in the semi-finals and won 2-0. In the final against the UPennalizers, the game ended with a 1-1 tie in regulation. TT-UT Austin Villa won the game 3-2 in penalty kicks to win the US Open championship. Scores from all of TT-UT Austin Villa's US Open games are shown in Table 2.

7.2 RoboCup 2009

The 13th International Robot Soccer Competition (RoboCup) was held in July 2009 in Graz, Austria[4]. 24 teams entered the competition. Games were played with three robots on a team. The tournament consisted of two round robin rounds, followed by an elimination tournament with the top 8 teams. The first round consisted of a round robin with eight groups of three teams each, with the top two teams from each group advancing. In the second round, there were four groups of four teams each, with the top two from each group advancing. From the quarterfinals on, the winner of each game advanced to the next round.

All of Austin Villa's scores are shown in Table 3. In the first round robin, TT-UT Austin Villa was in a group with SPQR and NTU Robot Pal. Austin Villa beat SPQR 2-0 and had a 0-0 tie with NTU Robot Pal. In the second round robin, Austin Villa was placed with Team Chaos, B-Human, and Burst. Austin Villa beat both Team Chaos and Burst 2-0, but lost to B-Human 9-0.

Austin Villa faced the Austrian Kangaroos in the quarterfinals and came away with a hard fought 2-0 victory. In the semi-finals, Austin Villa faced the eventual winners B-Human again, this time holding them to 7-0 (the closest game they had had at the time). There was a very fast turn around between the semi final and the third place game. With some remaining hardware issues from the semi final, Austin Villa lost the third place game to the Dortmund Nao Devils 4-1. Austin Villa finished 4th overall in the tournament.

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

Round	Opponent	Score
Round Robin 1	SPQR	2-0
Round Robin 1	NTU Robot PAL	0-0
Round Robin 2	Team Chaos	2-0
Round Robin 2	B-Human	0-9
Round Robin 2	Burst	2-0
Quarterfinal	Austrian Kangaroos	2-0
Semifinal	B-Human	0-7
Third Place Game	Nao Devils	1-4

Table 3: RoboCup 2009 Results

8 Conclusion

This report described the technical work done by the TT-UT Austin Villa team for its entry in the Standard Platform League. Our team developed an architecture that 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. Our work on this code paid off with a US Open Championship and a fourth place finish at RoboCup 2009. We plan to continue to progress on our codebase and continue to compete in RoboCup in 2010.

References

[1] T. Hermans, J. Strom, G. Slavov, J. Morrison, A. Lawrence, E. Krob, and E. Chown. Northern Bites 2009 Team Report. Technical report, Department of Compute Science, Bowdoin College, December 2009.

[2] T. Hester, M. Quinlan, and P. Stone. UT Austin Villa 2008: Standing on Two Legs. Technical Report UT-AI-TR-08-8, The University of Texas at Austin, Department of Computer Sciences, AI Laboratory, November 2008.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.