Austin Villa 2010
Standard Platform Team Report

Samuel Barrett, Katie Genter, Matthew Hausknecht,
Todd Hester, Piyush Khandelwal, Juhyun Lee,
Michael Quinlan, Aibo Tian and Peter Stone
Department of Computer Science
The University of Texas at Austin
{sbarrett,katie,mhauskn,todd,piyushk,
mquinlan,impjdi,pstone,atian}@cs.utexas.edu

Mohan Sridharan
Department of Computer Science
Texas Tech University
mohan.sridharan@ttu.edu

Technical Report UT-AI-TR-11-01

TT–UT Austin Villa
The University of Texas at Austin

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[1]. Austin Villa won the 2009 US Open and placed fourth in the 2009 RoboCup competition in Graz, Austria. In 2010 Austin Villa successfully defended our 1st place at the 2010 US Open and improved to finish 3rd at RoboCup 2010 in Singapore. This report describes the algorithms used in these tournaments, including the architecture, vision, motion, localization, and behaviors.

[1]For brevity we will often refer to our team simply as Austin Villa

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 [11].

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)[2]. 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[3], shown in Figure 1.

Figure 1: An Aldebaran Nao robot competing at RoboCup 2010.

1.1 Austin Villa Nao History

Our team is Austin Villa[4], 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, Graduate students Samuel Barrett, Katie Genter, Matthew Hausknecht, Todd Hester, Piyush Khandelwal, Juhyun Lee, Aibo Tian and postdoc Michael Quinlan from UT Austin, and Professor Mohan Sridharan from TTU. About half our members are new to RoboCup competitions.

We started the codebase for our Nao team from scratch in December of 2007. Our previous work on Aibo teams [18, 19, 20] 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

[2]http://www.tzi.de/spl/

[3]http://www.aldebaran.com/

[4]http://www.cs.utexas.edu/~AustinVilla

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 [4].

We continued our work after RoboCup 2008, making progress towards the US Open and RoboCup in 2009. We finished 1st and 4th respectively in these two competitions while making significant progress in our soccer playing abilities. The enhancements performed during 2009 can be found in last year's team report [6].

1.2 2010 Contributions

This report describes all facets of our Nao team codebase. For completeness, this report repeats relevant portions of the 2008 and 2009 team reports. The main changes for 2010 included a complete re-write of vision, appropriate enhancements to localization, a new kicking engine, and a behavior overhaul.

Section 2 describes our software architecture that allows for easy extendability and debugability. Our new vision system is described in section 3. Our localization (Section 4) is similar to that of previous years except we now have additional information from vision. Section 5 describes our motion modules used on the robot, including a new kick engine. Section 6 briefly describes the behaviors we developed and employed in 2010. Section 7 presents our challenge entries while section 8 describes our soccer results at the US Open and RoboCup.

1.3 2009-2010 Relevant Publications

Over the past year our team has produced a variety of publications involving RoboCup and/or the Nao platform [21, 14, 5, 3, 1, 10].

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.

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.*

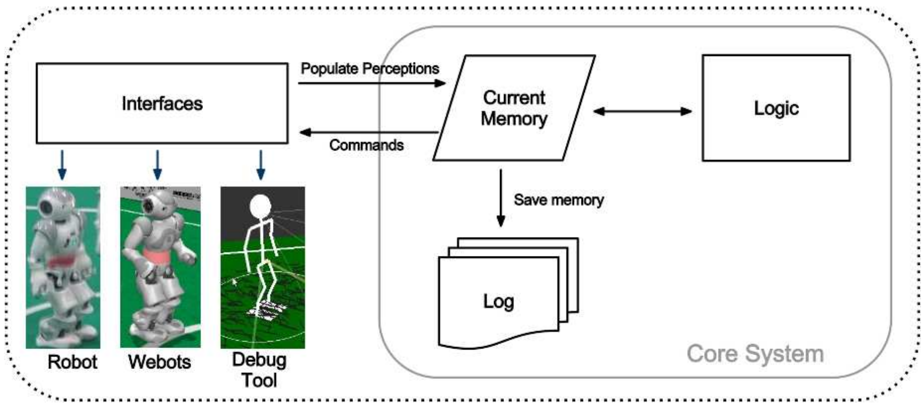


Figure 2: Overview of the Austin Villa software architecture.

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.

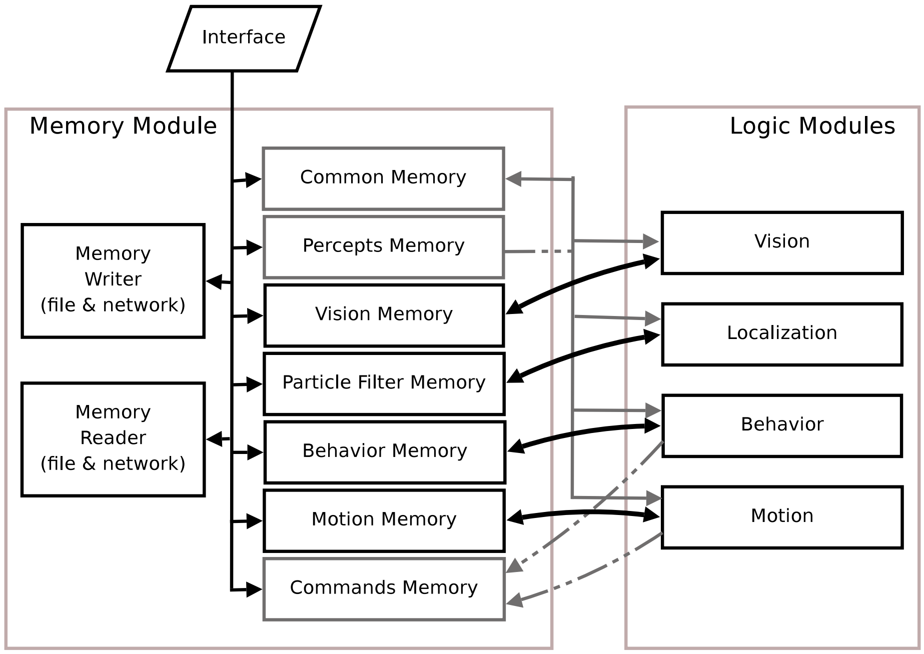


Figure 3: Design of the Memory module. The gray boxes indicated memory blocks that are accessed by multiple logic modules. Dotted lines show connections that are either read or write only (percepts are read only, commands are write only).

,

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

The raw images provided by the Nao are in the YUV422 format and have a 640 by 480 size, giving the images a size of 600 KB. A huge amount of the processing time in vision is consumed by accessing this image, which amounts to 8.8 MBps at 15 fps. This is a significant difference from previous years where a 16 times

(a) Raw Image (b) Segmented Image (c) Pixel Sub-sampling

Figure 4: The segmentation procedure is shown here. The raw image in Fig. 4(a) is converted to the segmented image using the color table. If we segment all the pixels, we get the image in Fig. 4(b). In Fig. 4(c), we show an enlarged portion of the blue box in Fig. 4(b), which shows the true segmented image with pixel sub-sampling.

smaller image (160 by 120) was used. Since image accesses are the bottleneck of the image processing system, we can increase frame rate by optimizing the memory accesses of the vision system. We present two independent techniques that should be applicable to any color based segmentation system.

The first technique happens at the lowest level of the hardware: cache, memory, and memory controller. The main idea is to reduce the number of memory accesses with type casting and to maximize the cache hit rate. When accessing two neighboring pixels, we burden the memory controller only once by retrieving a 32-bit value instead of querying four 8-bit pixel channel values independently. This retrieves information about two adjacent pixels together, containing the Y channel for both the pixels, and the shared U and V channels. Thus, the number of memory read requests can be reduced by a factor of 4. Once the pixels' YUV values are read, the YUV tuple values are looked up in the color table, which contains a one-to-one mapping between YUV tuple values and segmented colors [17]. The second technique reorganizes the color table to maximize the cache hit rate when querying a sequence of YUV tuples. Since two adjacent pixels share U and V channels, the order of the color channels in the color table is maintained as VUY instead of YUV. This reduces cache misses while segmenting a pair of adjacent pixels, due to their proximity in the color table. Results from memory optimization are presented in Table 1.

Method	**Time (ms)**
No optimizations	138.616
Color table reorganization	112.867
Color table reorganization & efficient retrieval	**96.023**

Table 1: The time taken to access and segment one image frame.

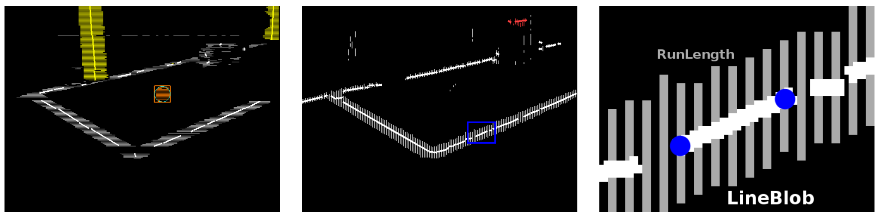
Our vision system divides the object detection task into 3 stages, which are listed below.

1. **Blob Formation -** The segmented image is scanned using horizontal and vertical scan lines, and blobs are formed for further processing.

2. **Object Detection -** The blobs are merged into different objects. In this paper, we shall primarily limit our discussion to line and curve detection.

3. **Transformation -** We use the information given by the pose of the robot to generate ground plane transformations of the line segments detected.

The new contributions in this years vision processing pipeline are a new method for rapid line and curve detection during the object formation phase, and using the transformations to better differentiate between lines and curves.



(a) *HorizontalRunLengths* & *VerticalBlobs* (b) *VerticalRunLengths* & *HorizontalBlobs* (c) Enlarged view

Figure 5: A representative image for the blob formation procedure. An enlarged view of the blue box in Fig. 5(b) is shown in Fig. 5(c) for clarity. In Fig. 5(c), the grey lines are the vertical run lengths and the white line is the corresponding horizontal blob formed.

3.1 Blob Formation

The blob formation procedure is similar to previous approaches such as [2] and is outlined in Fig. 5. The segmented image is scanned using vertical and horizontal scan lines, effectively sub-sampling the image (Fig. 4(c)). A vertical scan line is placed every 4 columns and a horizontal scan line every other row, and only these sub-sampled pixels are segmented (Fig. 4). Our system also retrieves and segments additional pixels in an ad hoc manner when we are unable to detect the ball. From these scan lines, we form *RunLengths* of same colors. *RunLengths* are a set of contiguous pixels along a scan line having the same segmented color value. Fig. 5(a) shows the *HorizontalRunLengths* generated using horizontal scan lines, and Fig. 5(b) shows the *VerticalRunLengths* from the vertical scan lines. Fig. 5(c) gives an enlarged view into a portion of Fig. 5(b), which shows the individual *VerticalRunLengths* in grey.

In the next stage we form blobs from these run lengths. Instead of using the Union-Find procedure as in [2], we simply merge adjacent run-lengths of the same color on the basis of overlap and width similarity. The main reason behind using this simpler approach is that it is more geared towards finding lines, and not a similar blob of color. Note that *VerticalRunLengths* combine to form a *HorizontalBlob*, as shown in Fig 5(c). Similarly *HorizontalRunLengths* are combined to form *VerticalBlobs* (Fig. 5(a)).

For each blob formed, we calculate some information required for further processing. For a given horizontal blob, we have a start point and an end point, as shown by the blue circles in Fig. 5(c). This gives us start coordinates (x_s, y_s) and end coordinates (x_e, y_e). We also calculate $\dot{y}_s$ and $\dot{y}_e$ as the start and finish slope respectively, by calculating slope across a few run lengths. Next we calculate $\ddot{y}$, which is the rate of change of slope across the blob. Some averaging is performed to account for noise. A similar set of values are also calculated for each vertical blob.

3.2 Object Detection

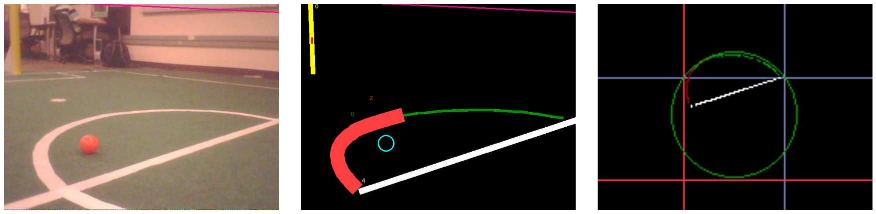
Algorithm 1 Line Segment Formation

```
Input: B ← Ordered set of Blobs
Output: S ← A set of candidate LineSegments

for each Blob b_i in B do
  s = new LineSegment
  s.initialize(b_i) {Initialize A,B,C. Put blob in segment}
  bestSegment = recurse-check-segment (s, b_i, B)
  if isAboveThreshold(bestSegment) then
    for each Blob b in s do
      B.remove(b)
    end for
    S.add(s)
  end if
end for

function recurse-check-segment (s, b_i, B)
bestSoFar = new LineSegment
for each Blob b_j in B s.t j > i do
  if b_j ≈ s.predictb_j.x_s then
    s.updateABC(b_j)
    s.add(b_j)
    current = recurse-check-segment (s, b_j, B)
    if is-better(current, bestSoFar) then
      bestSoFar = current
    end if
  end if
end for
return bestSoFar
end function
```

3.2.1 Line Segment Detection

In this section, we will primarily talk about our approach to line segment detection, as this forms the basis for our field line and goal detection algorithms. Vertical blue and yellow line segments serve as candidates for goal posts. Horizontal and vertical white line segments serve as candidates for the lines and center circles on the field.

We explain our methodology for segment formation in terms of horizontal blobs, and an equivalent discussion should automatically apply for the vertical blobs. To construct candidate segments, we make the simplifying assumption that each line segment, whether it be a straight line or an ellipse (i.e. a projection



(a) Raw Image (b) Detected Segments (c) Transformed Image

Figure 6: For the raw image in Fig. 6(a), the detected objects are shown in Fig. 6(b). The circle is detected in 2 separate segments. The transformation to the ground plane is shown in 6(c).

of the center circle - Fig. 6(a)), can be represented by the following parabola for short segments:

$$y = ax^2 + bx + c \tag{1}$$

The advantage of this assumption is that it gives simple expressions for the derivatives of this function. This allows us to reconstruct the curve given a point on the curve (x_1, y_1), and the slope and the rate of change of slope at that point $(\dot{y}_1, \ddot{y}_1)$:

$$\dot{y} = 2ax + b, \quad \ddot{y} = 2a, \quad \text{which gives} \tag{2}$$

$$a = \frac{\ddot{y}_1}{2}, \quad b = \dot{y}_1 - 2ax_1, \quad c = y_1 - ax_1^2 - bx_1 \tag{3}$$

The procedure for merging the blobs together is presented in Algorithm 1. To merge these blobs together, we traverse over the set of sorted blobs using a stack. We initialize a segment (the stack) with a single blob b, and calculate the parameters a, b and c for that segment, using the values $(x_e, y_e, \dot{y}_e, \ddot{y})$ calculated for the blob (Alg. 1 Line 6).

Once initialized, we start the recursion process (Alg. 1 Line 7). Using the current a, b and c values from the segment, we can predict the curve for any given x (using Eqns. 1 and 2). We then check if any blobs in the global list match our prediction. Once a match is found, we add that blob to the stack, recalculate the a, b and c values (using Eqn. 3), and recursively repeat the procedure.

We backtrack along the stack if we are unable to find a suitable blob to add, or when all possible options have been explored. While backtracking, we keep track of the best candidate found in front of every node, in a form of dynamic programming. Once the entire tree has been explored, the best candidate at the root node is the most suitable line segment. If it satisfies the threshold for being a line segment, we add it to the list of candidate line segments and remove all the blobs in this segment from the overall blob list. The final object detection is shown in Fig. 6(b).

3.2.2 Ball Detection

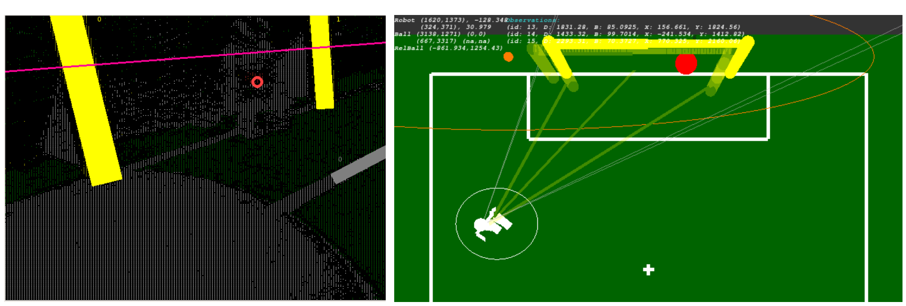
Ball detection runs the Union-Find algorithm [2] on orange blobs from the blob formation procedure to merge them into a set of candidate balls. Our ball detection method is fairly similar to last year's approach [6], and a number of heuristics are employed to calculate the probability of each candidate being the actual ball. Candidates are initially refined by a measure which calculates their similarity to a circle. The ball distance is then estimated using the width of the blob, and is used to project the ball in 3d space using the transformations described in Section 3.3. The height of the candidate then serves as the second main measure of refinement, as the true ball should be fairly close to the ground. There are a number of other less significant measures used to refine the hypotheses. If there are candidates still present after the refinements, the most probable candidate is detected as the ball.

A major change this year was incorporating the higher resolution image to see the ball at further distances. This was necessary due to a reduction in the ball size. We initially attempt to detect the ball from the orange blobs formed according to our pixel sub-sampling strategy. If no candidate is selected as the ball, we can optionally retrieve further detail from the image. This is done by retrieving all pixels adjacent to previously detected orange pixels. We then rerun the blob formation procedure, to form blobs with a finer grain of accuracy. This additional detail produces better probability than before while running the same heuristics for refinement. Due to a high computational cost, this scan is only run when required, and only if the initial scan fails.

Based on our testing the low resolution scan is able to detect the ball at a maximum distance of about 2.5 meters, and the high resolution scan can detect the ball at a maximum distance of about 4.0 meters. These values were observed with the robot standing still and searching for the ball. With movement, there may be excessive noise in the image, and it may take multiple scans to detect the ball.

3.2.3 Robot Detection

Robot detection was written from scratch this year, and was used by our kick selection engine to avoid hitting the ball into other robots. Robot detection takes all the pink and blue horizontal blobs detected from the blob formation procedure. These blobs may be generated from the team identification markers from the robots, and are taken as possible candidates for the robots. We then look for white pixels above and below each blob, to ascertain the presence of the white torso and legs of the robot. Additionally, all blobs that are close together are merged into a single blob. All refined candidates are treated as detected robots (Figure 7).



(a) Vision

(b) Localization

Figure 7: A red robot being observed in vision and its tracked position in localization.

One of the main difficulties with robot detection is that the the team identification markers are difficult to segment, as they quite often lie in the shadow of the robot's torso. Additionally they may also be obscured by the hands of the robot. Because of this reason, we found estimation of the detected robot's position based on the intrinsic properties of the color band fairly challenging. We found a solution to this problem by using the green pixels below the robot in the segmented image. If the robot was detected correctly, then these green pixels should represent the ground just below the robot. Using the transformations from Section 3.3, we can find the position of this region of the ground, and consequently that of the robot.

3.3 Transformation

For the purpose of localization, it is important to be able to distinguish between lines and curves. Otherwise, a robot observing the center circle could mistakenly believe that it is observing the center line or vice versa. Distinguishing between lines and curves can be difficult in the vision frame because the projections of lines and circles on the camera image often look fairly similar. The inevitable noise and incorrectly formed line segments exacerbate the problem.

For the purpose of distinguishing lines from curves, we use transformations. Based on the current pose of the robot, it is possible to construct a transformation matrix from a point in the vision frame onto the ground plane [6]. This transformation is possible because the height and orientation of the camera can be estimated relative to the ground using the robot's sensors. Since this matrix is calculated once every frame, we can efficiently transform from the vision frame to the ground plane.

After obtaining the transformation for each segment, we use a simple metric for first classifying whether a detected segment is a line or not. We calculate the angle between the first and second halves of the segment, and classify the segment as a curve if this angle is above some threshold. For all curves, we perform circle fitting by calculating the center and radius using 3 seed points. We then verify if the circle has roughly the same radius as the center-circle on the field.

This process is illustrated in Fig. 6. Given the raw image (Fig. 6(a)), we can apply the vision system to generate the final detected segments (Fig. 6(b) - note that two segments are detected from the circle). We transform these segments (Fig. 6(c)) and perform circle fitting on the curves. The green colored segment shows a curve that was fit to the circle shown in green. The red colored segment shows a curve which was not detected as a circle, due to an incorrect projection.

4 Localization

We used Monte Carlo localization (MCL) to localize the robot. Our approach is described in [15, 7]. 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:

$$r_d = d - \hat{d} \tag{4}$$

$$s_d = e^{-r_d^2/\sigma_d^2} \tag{5}$$

$$r_\theta = \theta - \hat{\theta} \tag{6}$$

$$s_\theta = e^{-r_\theta^2/\sigma_\theta^2} \tag{7}$$

$$p(O|h) = s_d \cdot s_\theta \tag{8}$$

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. The ambiguous landmarks include field lines, intersections, and individual goal posts, while both entire goals and the center circle are distinct landmarks.

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

$$n \times \frac{p_i}{\sum_{j=1}^{n} p_j} \tag{9}$$

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 [12].

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:

$$i = \text{MAX-DISTANCE} \cdot (1 - p) \cdot random(1) \tag{10}$$
$$j = \text{MAX-DISTANCE} \cdot (1 - p) \cdot random(1) \tag{11}$$
$$k = \text{MAX-ANGLE} \cdot (1 - p) \cdot random(1) \tag{12}$$

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. One addition to 2010 was a modification to the random walk. In previous years each particle was walked independently, as a result the mean position of the particles would shift from frame-to-frame even when no vision observations were made. This meant that the robot locations was unnecessarily 'jittery'. To overcome this we now randomly walk only 50% of the particles and apply the opposite walk to the remaining particles. This results in the overall mean position of the particles staying the same, while the std deviation increases (i.e. the robots location remains stable but its belief uncertainty increases).

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 [7], 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

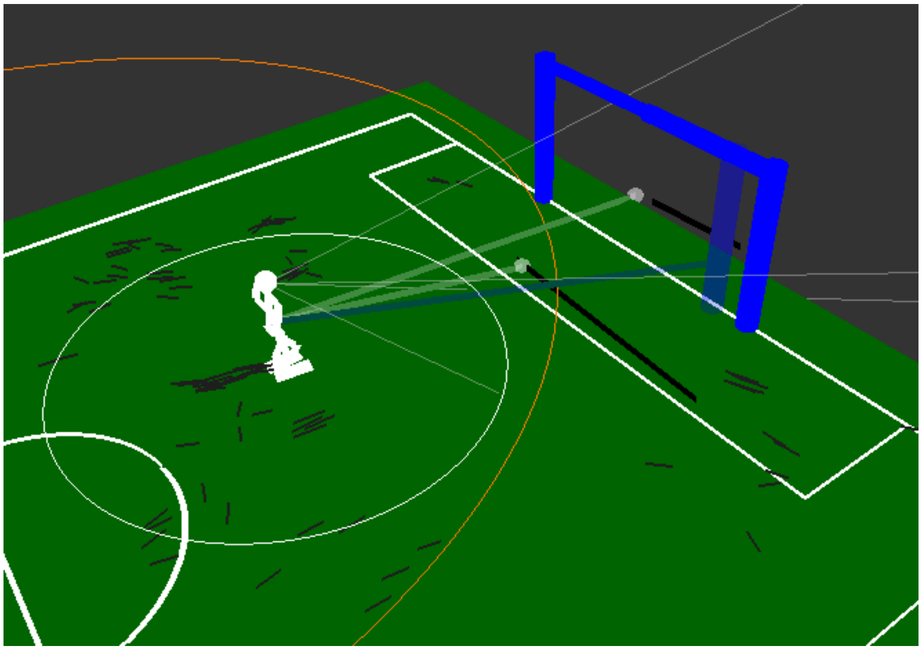


Figure 8: Example of Robot Pose Estimate and Particles. Here the robot has seen a blue goal post and two field lines. The particles are the small black lines that are scattered around the robot, with their length representing their probability. The robot's pose estimate is at the weighted mean of the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

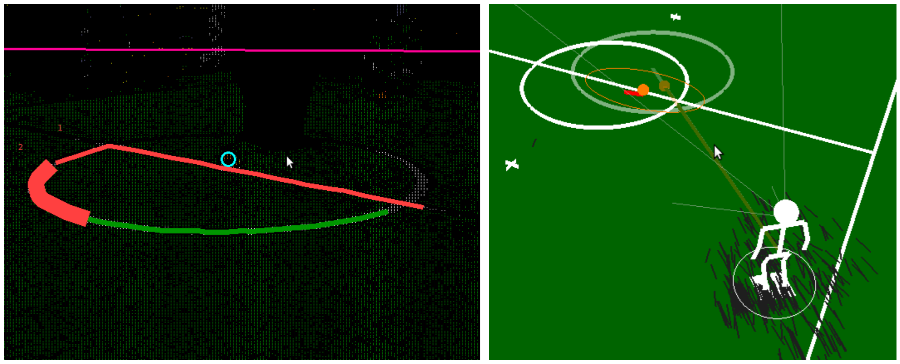
within the robot's field of view. It is important to note that the robot can also miss observations that are within its view for reasons such as image blurring or occlusions, and these situations need to be considered when updating a particle's probability based on negative information. Our work on negative information is based on work by Hoffmann et al [8, 9].

We update particles from line observations by finding the nearest point on the observed line and the expected line. We then do a normal observation update on the distance and bearing to these points.

In 2010, we added the capability of detecting the center circle to our vision system. We used the center circle as a distinct landmark similar to the goals. An example of using the circle to update the particle filter 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 [13].

When an opponent robot is detected through the vision system, the localization system needs to track it. It maintains estimates of the global locations of three opponent robots along with the last time they were seen. If the detected robot is within 300 cm of a previously detected robot, then that robot's global location is updated with the new one. Otherwise, the oldest opponent estimate



(a) Vision (b) Localization

Figure 9: The center circle being observed in vision, and the resulting localization update.

is replaced with the new observation. Essentially, upon seeing a robot, we assume that it remains in the same global location and use that estimate for the next 6 seconds for kick decisions.

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.

5 Motion

We used the provided Aldebaran walk engine for our walk, with tuned walk parameters and joint stiffnesses. In 2010 we re-developed our own kick engine, allowing us to use multiple kicks in different directions with varying power.

5.1 Kick Engine

Rather than having a large set of static kicks for each situation the robot might encounter, we instead created a small set of parametrized kicks. Our kick engine selects the appropriate parameters and then executes the sequence of actions for a given kick. Our kick engine is designed to provide a wide range of angles and distances, as well as handle variance in the ball's starting position. This reduces the need to align carefully to the ball, which allows the robot to move the ball faster with less chance of an opponent blocking the kick. The engine selects the parameters based on the position of the ball and the desired kick target as described in the following sections. Specifically, the engine can handle desired kick angles from $10°$ to the inside of the kicking leg out to $30°$ to the outside, and distances ranging from 0.5 meters to 3.5 meters.

5.1.1 Basic Control

The robot's kick engine used the state machine with the states described in Table 2. In our tests, separating moving the swing foot back and placing it down helped stability significantly. The Check Ball state stores the current position of the ball, calculated directly from the robot's camera image to eliminate the dependence of the kick on the filtering of the ball position. Furthermore, the head and body pose were fixed for the Check Ball state, so the robot did not need to filter out the noise from its own movement. Note that if the kick type does not specify a leg, the leg is chosen in the Check Ball state. For all of this work, we control the robot's joints using inverse kinematics.

State	**Description**
Check Ball	Assume a standard position and check the current position of the ball
Shift Weight	Shift the robot's weight to the stance foot
Align	Lift the swing leg and position it for the kick
Kick	Move the swing leg forwards to perform the kick
Move Foot Back	Move the swing foot back to be in line with the stance leg
Place Foot Down	Place the swing foot back on the ground
Shift Weight Back	Shift the robot's weight back to balanced on both feet

Table 2: States of the kicking engine's state machine

5.1.2 Variable Ball Position and Angle Control

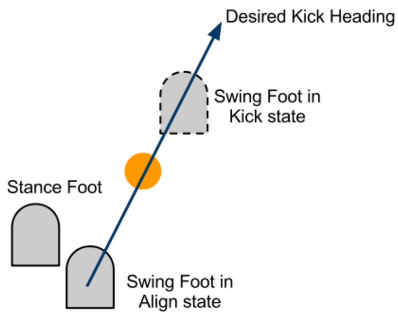
To handle variable ball positions and kicking at a range of angles, the robot must control the starting and ending positions of its foot for the kick. The goal is for the foot to move along a straight line through the ball at the desired kick angle, with the ball centered on the foot.

Most of the states defined above are not affected by the ball position and kicking angle; only the Align and Kick states must be altered based on these factors. The Align state must move the foot to the back point of the desired line segment, specifically 8 cm behind the center of the ball at the appropriate angle. The Kick state ends with the foot 4 cm in front of the original center of the ball. We use trigonometry to find these points from the angles, but bound both the side and forward distances of the foot to avoid singular values in the inverse kinematics and to prevent clipping the stance foot. Two examples of the foot placements for the kick are shown in Figure 10.

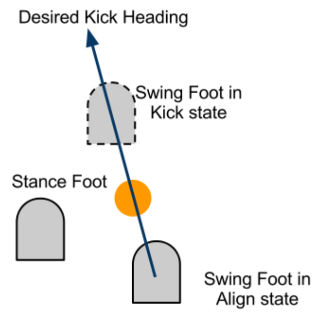
Note that in the Check Ball state, the robot saves the position of the ball in the image as a coordinate in pixel values from the camera image. For determining the positions of the foot, we directly use the ball's coordinates within the camera image. Unfortunately, not all of the robots are calibrated the same; as such, the same commanded angles of the head may result in different positions on different robots. Therefore, for each robot, we calculated the offset of the head and used this value to correct the estimate of the position of the ball.

5.1.3 Distance Control

Kicking distance was controlled by changing the interpolation time of the Kick state, i.e. the time taken to swing the leg forwards during the kick. We determined the relationship between this time and the distance empirically, collecting



(a) Kicking to the outside.



(b) Kicking to the inside.

Figure 10: Foot placements during the Align and Kick states.

distances for varying interpolation times and averaging over three kicks. Then, we fit a linear model to the logarithm of the distance, arriving at the model:

$$t = -0.215 \log(d) + 1.824$$

Also, we discovered that using times of less than 0.05 seconds had no effect on the kick due to the maximum speed of the joint and times longer than 0.5 seconds did not reliably result in the robot moving the ball at all. It is important to note that the function relating distances and interpolation times is dependent on the field surface, so it was necessary to recalibrate it for the RoboCup competition.

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 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

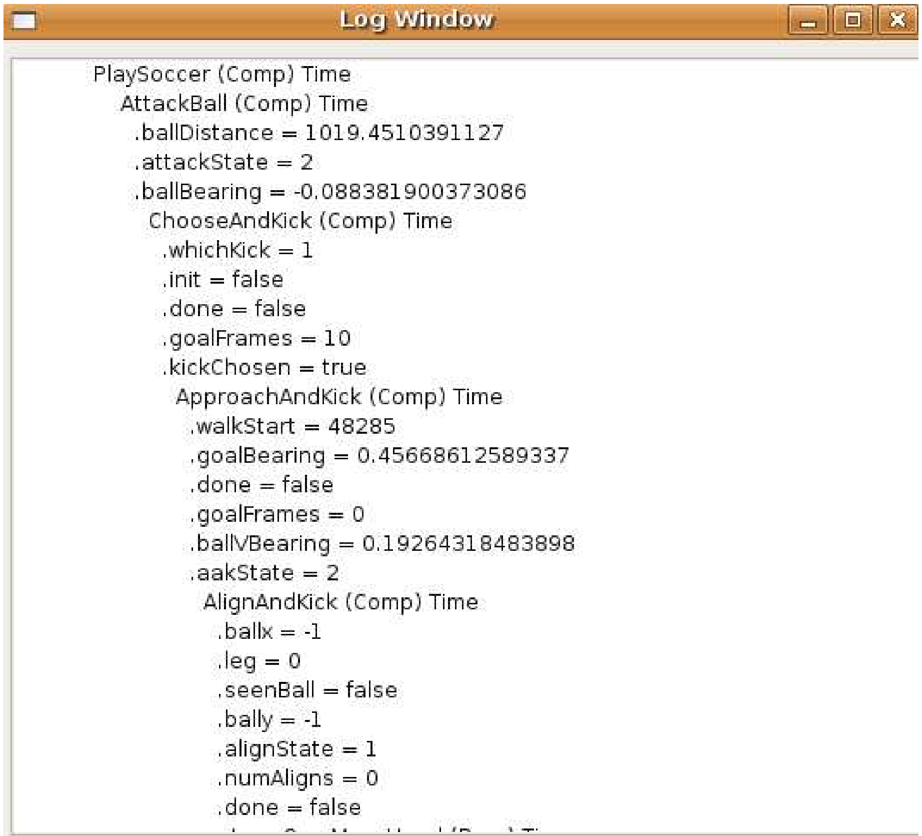


Figure 11: Example Behavior Log, showing the trace of task calls and their state variables.

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 sure to do the right thing once we got to it. Once the ball was found, the robot would walk to the ball as quickly as possible. While approaching the ball the robot would actively choose to scan for objects based on its current location and uncertainty. When a certain uncertainty threshold was reached the robot would slightly slow its walk and scan either left, right or both direction in search of goals or lines. If the robot sees enough information (i.e. both goal posts) it will exit the active head scan and return to focusing on the ball. Once the robot got sufficiently close to the ball it would either choose an appropriate kick or circle the ball, as further described in the remainder of this section.

Highlights of our behavior and kick strategy can be found at `http://www.youtube.com/watch?v=-9gEMOGW1Qg`.

6.1 Kick Selection Strategy

We created a kick decision engine that enables the robot to keep the ball away from the sidelines and to continue moving it towards the opponent's goal, all while avoiding opponents. To do this, we check if the result of the kick will place the ball in a dynamically calculated infundibuliform that forces the robot to funnel the ball towards the goal. We use this calculated region to select a preferred kick amongst all possible kicks the robot can make using its variable kick engine.

In addition to making sure all of our kicks move the ball consistently towards the opponent's goal, our second aim was to be *quick* at the ball. The final steps of approaching and lining up the ball for a kick can be quite slow, and opponents can get in the way of the kick if these phases take too long. For this reason, our robots take the first kick allowed by the strategy rather than spending more time to line up for a stronger, more accurate kick.

Having defined the method for implementing a quick kick with controllable kick and angle (Section 5.1), we are then presented with the challenge of selecting from among all the possible ways in which to kick the ball at any given time. In this section, we describe our approach to this problem. One of our main objectives was to minimize the time we spent at and around the ball, while still maximizing the outcome of each kick. To achieve this we construct a parametrized system that allows us to define a valid *kick region* that ensures that any kick chosen will result in the ball being moved closer to the opponent's goal and away from the sidelines. The robot has a set of possible kicks that can be made using the kick engine, and determines which ones are *valid* by determining if they will likely result in the ball being in the kick region. The potential kicks are ordered based on distance, and if there are multiple valid kicks, the kick that goes the farthest is selected. When no valid kicks exist, the robot continues to approach the ball, which consists of walking up to and circling it, until a viable kick is found. By following this approach the robot spends the minimum time approaching the ball, while still guaranteeing that the chosen kick will move the ball closer to the goal.

6.1.1 Defining the Kick Region

The kick region defines an area on the field which we deem as *valid* to kick into. This region ensures that the robot always kicks the ball towards the opponent's goal, but away from opponents and the sidelines. An example valid kick region for a particular ball position is represented by the shaded area in Figure 12. The principle is that any kick, factoring in the robot's possible orientation error, that is predicted to result in the ball staying inside the kick region is a *valid* kick to choose from. On the other hand, any kick that will not place the ball in this region is an *invalid* kick that should not be selected.

The kick region is described by 6 parameters, presented in Table 3 and visualized in Figure 12. The general concept is that the region allows most kick options (to keep the ball moving) but funnels the ball towards the goal, therefore keeping the ball out of the corners of the field. We can define the steepness of the funnel stem, the width of the funnel, and the opening angle of the funnel's mouth. The kick strategy can be changed at any stage during the match. For example, we can have different strategies depending on the game situation or even depending on the current field setup.

Parameter (unit)	**Description**
Edge Buffer (mm)	Creates a non-valid region of this width on each sideline.
Post Angle (deg)	Shapes the valid region such that it narrows towards the target goal with the given angle.
Opening Angle (deg)	Defines the valid region as angled outwards from the robot, i.e. always kick forwards at a minimum of the given angle.
Inside Post Buffer (mm)	Minimum distance inside each goal post that the robot should aim.
Shooting Radius (mm)	Defines a semi-circle in which the robot should strongly attempt to score, i.e the valid kick regions becomes the area inside the target goal.
Own Goal Radius (mm)	Defines a semi-circle around your own goal in which no valid kicks should exists, i.e. do not kick across the face of your own goal.

Table 3: Parameters that define the valid Kick Region

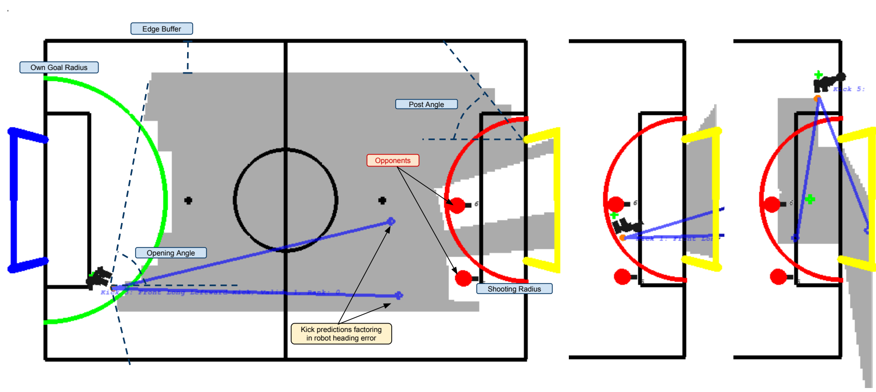
6.1.2 Avoiding Opponents

In addition to making sure that all kicks move the ball towards the opponent's goal, the kick region is also used to ensure that the ball is not kicked towards opponents. When an opponent robot is detected in the robot's camera image, its location and the time it was seen are saved to memory. The location of this opponent is remembered for 6 seconds, after which point we assume there is a high chance that the opponent has moved. All locations behind the opponent robot, and locations up to 20 cm in front of the opponent robot, are considered to be invalid. This prevents the robot from selecting a kick that would attempt to kick at or through an opponent.

Along with using the opponents to invalidate parts of the kick region, we also use them to calculate the best heading towards the goal. This heading gives us a possible direction to aim our kick to maximize the chance of scoring by shooting directly between the keeper and one of the goal posts. We calculate the bearing to each goal post and the bearing to any opponent between the two posts. We then calculate the size of the "gap" between the opponent and each post in degrees. The larger of these two is considered the "large gap" and the smaller is the "small gap." Along with the kicks at discretized distances and headings, two kick choices are added that kick at maximum distance towards the center of each of the two gaps. If no opponent is detected between the goal posts, the large gap kick is directed towards the center of the goal, and the small gap kick is directed to be just inside the near post.

6.1.3 Choosing a Kick

Algorithm 2 shows how the robot decides when and where to kick using the kick region. The robot has a set of possible kicks it can make, using a small set of the kick ranges and angles possible with our kick engine. Table 6.1.3 shows



(a) Kick Region Parameters

(b) Shooting Radius

(c) Corner Behavior

Figure 12: (a) The parameters that define the kick region and an example of avoiding opponents. The highlighted area indicates the valid kick region. (b) Example of the kick region when the ball is inside the shooting radius (the semi-circle). In this case, the robot will strongly prefer to score a goal; this is achieved by setting only the area inside the goal as valid kick region. (c) When the ball is in the corners, we modify the kick region to include an additional area in front of the goal.

the kick choices that are given to the robot. The first two choices use kick headings calculated to aim towards the gaps between the goal keeper and the goal posts. The remaining kicks use discretized headings of $-30°$, $0°$, and $30°$ with discretized distances of 0.85, 1.75, and 3.5 meters. The kicks are sorted based on distances and headings, such that longer kicks are examined first by the kick selection algorithm.

The robot goes through each of these kicks in order, and checks if they are valid by checking if the final ball location, including likely heading error, is in the valid region. For each kick, the robot adds both a positive and negative heading error to the kick based on the robot's localization uncertainty and the variance in the kick itself. These two locations indicate the maximum and minimum heading that we expect the ball might move. The robot calls *check_kick_region* for each of these locations, and it determines if this possible ball location is in the valid region. If both possible ball locations are valid, then the kick is a valid kick, otherwise, it is invalid.

The robot calls *approach_ball_arc_to_goal*, which walks towards and circles the ball to face the opponent's goal. It returns false after a timeout for circling the ball too long. Upon finding a valid kick, the algorithm stops circling the ball and executes that kick. If none of the kicks are valid, then the robot continues its approach until it has a valid kick or the approach times out. It is often the case that a short kick may be valid while a long kick would put the ball out of bounds, but had the robot circled the ball more it could have reached an angle such that a longer kick would have been better. This biases our robot to take faster, shorter kicks as opposed to longer ones that require more alignment.

Figure 13 shows all the possible kicks from one field location. The first two choices aim for the gap between the keeper and the goal posts. The robot is close

Algorithm 2 Kick Strategy

$b \leftarrow ball$
$r \leftarrow robot$
while approach_ball_arc_to_goal(A) **do**
  if $b_{dist} > A_{MAX\_DISTANCE\_FROM\_BALL}$ **then**
    continue
  end if
  $best\_kick \leftarrow -1$
  for $k \in K$ **do**
    $left \leftarrow$ check_kick_region(R, $k_{dist}, k_\theta, r_{\theta_\sigma}$)
    $right \leftarrow$ check_kick_region(R, $k_{dist}, k_\theta, -r_{\theta_\sigma}$)
    if $(left + right) = 2$ **then**
      $best\_kick \leftarrow k$
      break
    end if
  end for
  if $best\_kick > -1$ **then**
    execute K_{best_kick}
  end if
end while
execute K_{SHORT_KICK}

enough to the goal and has low enough localization uncertainty that it believes that both the left and right possibilities for these kicks will result in scoring. Therefore, it will determine that both of these kicks are valid and return Kick 0, towards the largest gap between the keeper and goal post. The validity of the other possible kicks is shown. Some of the kicks are invalid due to going out of bounds (Kicks 3 and 4), going towards the opponent (Kicks 2 and 5), or not meeting the opening angle from the robot (i.e. not going forward enough (Kicks 6 and 9)). In addition to the two scoring kicks, two of the shorter kicks (Kicks 8 and 10) are valid, but the longer scoring kicks would be preferred.

7 Challenges

In 2010 we finished in a excellent 2nd place in the challenges, narrowly missing 1st place by a single point (Table 5). We finished 1st in the passing challenge, 2nd in the dribbling challenge and 7th in open challenge.

7.1 Open Challenge

Seamlessly integrating both the Nao's cameras into a cohesive vision system remains a challenging problem. We take a step towards this goal by creating a system capable of auto-calibrating the hardware parameters for multiple cameras to achieve consistent color perceptions across all cameras.

The need for such an auto-calibration system arose when experiments indicated that the color perceptions of the Nao's top and bottom cameras were sufficiently different to preclude the Nao's color-based object identification with the same color table. The skewed color perceptions likely result from differences

Kick	**Distance (m)**	**Heading (deg)**
Large Gap Kick	3.5	Towards Large Goal Gap
Small Gap Kick	3	Towards Small Goal Gap
Long Straight Kick	3.5	0
Long Leftward Kick	3.5	30
Long Rightward Kick	3.5	-30
Medium Straight Kick	1.75	0
Medium Leftward Kick	1.75	30
Medium Rightward Kick	1.75	-30
Short Straight Kick	0.85	0
Short Leftward Kick	0.85	30
Short Rightward Kick	0.85	-30

Table 4: Possible Kicks

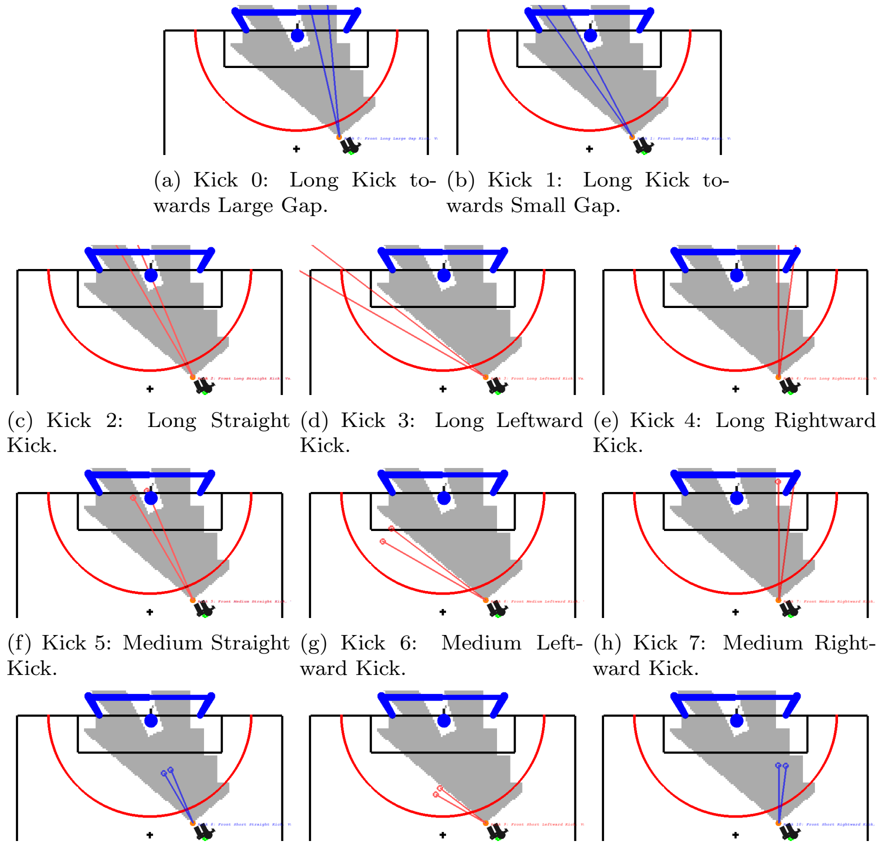
Team	Passing	Open	Dribbling	Total
rUNSWift	24	22	23	69
Austin Villa	25	19	24	68
CMurfs	22	13	21	56
B-Human	23	24	0	47

Table 5: Top 4 teams in the RoboCup 2010 Challenges. Maximum score in any challenge is 25 points.

in the amount and the manner in which light enters each camera as a result of the different camera enclosures and mounting angles used for the top and bottom camera. Additionally, small manufacturing differences even between top or bottom cameras of two different robots often result in different color perceptions. In the past this problem has been addressed by creating and tuning separate color tables for each camera, but already this takes considerable effort, even for experienced tuners, typically one hour per table. Adding the top camera of each robot to the list of cameras needing to be tuned would double the total number of color tables needing to created and tuned, increasing the amount of effort and time required to play on a new field.

Past work [16] has addressed a related problem of standardizing color perceptions by automatically creating color tables based on known objects in the robot's visual field. We take a different approach by automatically tuning the hardware parameters (brightness, contrast, hue, gain, etc.) of each camera until the perceived colors match color perceptions of a known good camera. After color perceptions are standardized, a single color table can theoretically be used for a full team of robots, saving the effort of manually creating one color table per camera or two per robot.

The process of standardizing color perceptions begins with a static image R being saved from a known good camera. R should be representative in the sense that it should contain non-negligible portions of each color defined in the color table. For the Naos this involved capturing an image containing orange balls, white lines, portions of the blue and yellow goal, and blue and pink team identifiers. Each color needs to be present in R because the color of other cameras will be adjusted to match this image and if only a few colors are present,



(a) Kick 0: Long Kick towards Large Gap.
(b) Kick 1: Long Kick towards Small Gap.
(c) Kick 2: Long Straight Kick.
(d) Kick 3: Long Leftward Kick.
(e) Kick 4: Long Rightward Kick.
(f) Kick 5: Medium Straight Kick.
(g) Kick 6: Medium Leftward Kick.
(h) Kick 7: Medium Rightward Kick.
(i) Kick 8: Short Straight Kick.
(j) Kick 9: Short Leftward Kick.
(k) Kick 10: Short Rightward Kick.

Figure 13: Example of kick choices from a location on the field. Kicks 0 and 1 are targeted at the gaps around the keeper. Kicks 2 and 5 are invalid because they would go straight at the keeper. Kicks 3, 4, 6, 7, and 9 go outside the valid region. Kicks 8 and 10 are valid, but have a lower ranking than valid kicks 0 and 1. Kicks are sorted by length, and the first valid kick is selected. So in this case, Kick 0 would be selected and the remaining kicks would never be evaluated.

these color will likely be well matched at the expense of the others.

After acquiring the static image R, another camera from the same or a different robot is selected and positioned such that the objects in its image I overlap those in R. We have found that using a graphical overlay of I and R serves as an effective guide to a human attempting to correctly position the new camera. The fact that the Nao's top and bottom cameras are mounted at different angles inside of the robot's head complicated the process of trying to align the top camera to exactly overlap a representative image taken from the bottom camera as the physical limits of the Nao's neck joints prevent the head from obtaining certain angles which would be necessary to compensate for the angular mount differences between the cameras. However, combining body and neck tilt allowed a close approximation of the original camera position. Figure 14 shows the graphical overlay of images taken from a Nao's top and bottom camera. This overlay is impossible to perfectly align for this reason.

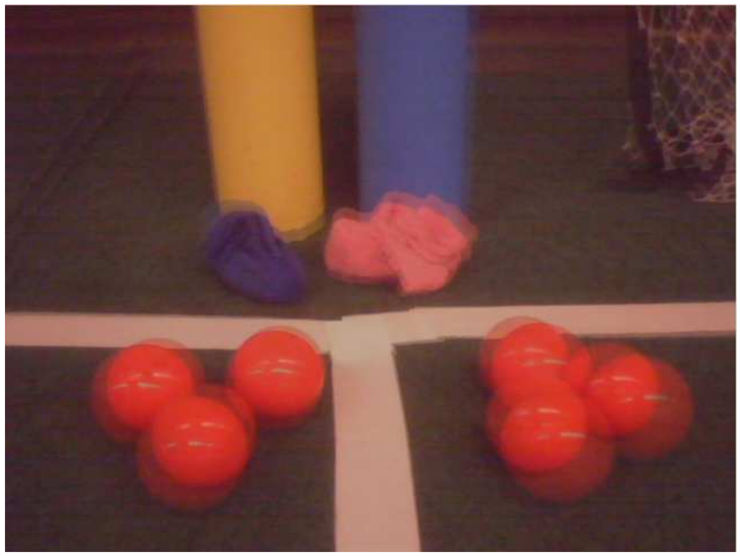
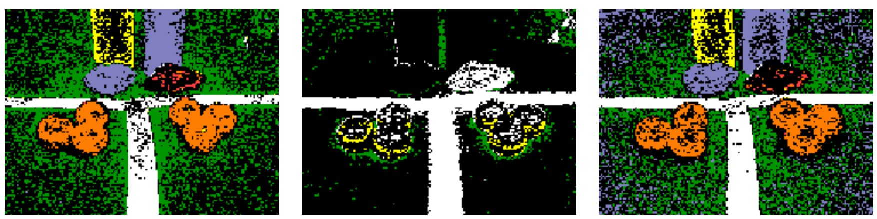


Figure 14: Approximate dual camera overlay



(a) Representative Image (b) Before Calibration (c) After Calibration

Figure 15: Segmented images before and after camera calibration.

Having aligned the current camera's image I with the representative image R, we proceed to automatically tune the gain, exposure, blue chroma, red chroma, brightness contrast, saturation, and hue of the current camera until we maximize the color match between I and R. Tuning is performed by a Hill Climbing search through the space of possible camera settings. At each iteration of hill climbing, a new set of parameters N is created by perturbing the current parameters C in a random direction. Next, some number of images (we used 3) are captured using the new settings. Finally, if the average color match of these images captured using N exceeds the average color match of the old parameters C, then N becomes the new baseline and another round of hill climbing begins.

The color match between each image I and R is computed by segmenting both I and R with some color table (Fig. 15(b) and Fig. 15(a) respectively), typically the one in use when R was saved. This allows us to filter out the irrelevant colors in both images and compute the image match over the relevant colors defined in the color table. To compute the image match, we examine each of the different colors in the representative image. For each pixel labeled with that color in R, we check if the corresponding pixel in I is also labeled with the same color. The average pixel match for that color is computed as the number of pixels in which R and I match divided by the total number of R's pixel of that color. Taking the average of these pixel matches provides the overall color match. Algorithm 3 shows the detailed computation. Computing the color match in this manner encourages the highest overall match between all colors present on the field. If certain colors were deemed more important than others, a weighted average could be used instead.

Algorithm 3 Color Match Computation

```
R ← Representative Image
I ← Current captured image
pixelCnt ← # pixels of each segmented color
matchCnt ← # pixel matches for each segmented color
avgMatch ← average pixel match for each color
for each segmented pixel p in image do
   truePixelColor ← R[p]
   givenPixelColor ← I[p]
   pixelCnt[truePixelColor]++
   if truePixelColor == givenPixelColor then
      matchCnt[truePixelColor]++
   end if
end for
for each color present in image do
   avgMatch[color] ← matchCnt[color]/pixelCnt[color]
end for
overallMatch ← average(avgMatch)
return overallMatch
```

In practice, the hill climbing search was run until convergence (Fig. 15(c)). Approximately two frames were scored per second, resulting in one iteration (3 frames) per 1.5 seconds. Convergence was typically found within 100 iterations or 2.5 minutes of real time. However, the algorithm was often allowed to run for several hundred more iterations to ensure it had reached a local maximum. Moreover, the only human intervention required throughout the entire process is the initial camera alignment.

Auto-calibration was successfully applied to each of the four Naos on UT Austin Villa's 2010 RoboCup team in order to learn camera parameters for each robot's top camera. When top cameras were integrated into the existing code, auto-calibration saved the team from having to tune the hardware parameters on each of the four robots' top cameras, as well as having to create new color tables for these four cameras. In practice, although the color match was likely good enough to use without color table modification, small changes were made to the color tables of each robot to reach a very fine grained color space. Even so, these modifications required time on the order of 5 minutes per color table rather than the typical hour to create a new table.

Figure 16 graphs the overall color match as the top camera of a Nao is auto-calibrated to match the color perceptions of the Nao's bottom camera. Figure 15 shows the representative image R from the bottom camera, and sample images from the top camera before and after calibration.

Several limitations of auto-calibration not present in UT's robotics lab became apparent after using it on the practice fields of RoboCup 2010. For in-

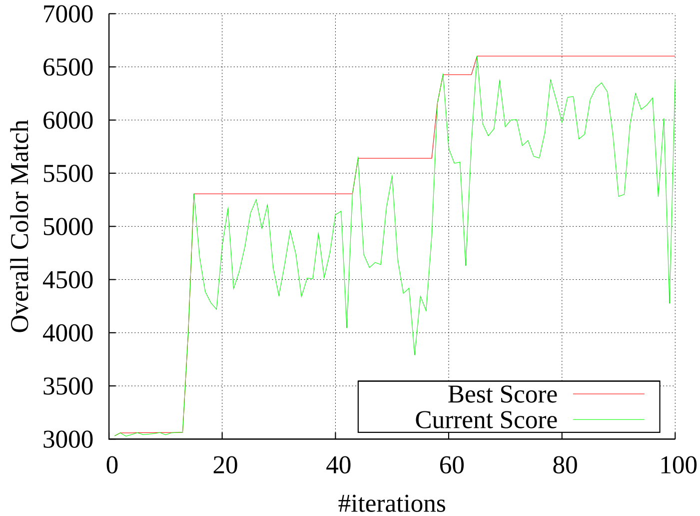


Figure 16: Sample calibration run converging to a local maximum.

stance, it was often challenging at RoboCup to create a representative image without having to re-arrange the goals on the practice field to fit into a single field of view, inconveniencing other people trying to simultaneously use the field. Additionally, auto-calibration works best with a static background image which was hard to maintain on a crowded practice field. However, temporary disturbances in background image pose little problem to the algorithm as they simply result in the current policy receiving low color match scores, which causes the hill climbing to stay at the same policy until the background disturbance is cleared.

We have presented an approach to the problem of standardizing color perceptions between different cameras. However, there is still much work to be done in coordinating the use of the top and bottom camera such as the choice of when to use each camera and how to identify the best times to switch between the two in order to maximize the amount of pertinent information acquired.

7.2 Results

The open challenge result is determined by each of the 24 teams voting for its favorite 10 competitors. We finished in 7th place including receiving some first places votes.

7.3 Passing Challenge

In the passing challenge, three successive passes must be made back and forth between two robots across the center of the field. The trial ends in failure if the ball or one of the robots leaves the field, if the ball stops inside the dead zone in the middle of the field or if one of the robots enters the dead zone. The trial ends in success once the receiving robot touches the ball after the third pass.

7.3.1 Using Variable Length Kicks

The passing challenge served as an excellent application of our team's variable length kicks described in Section 5.1. In the passing challenge, our approach was to have a player wait in the middle of each side of the field between the penalty cross and goal box. When the ball is on a player's side of the field, the player approaches the ball and aligns to kick straight to the penalty cross on the other side of the field. We calculate the desired kick distance as the distance from the ball to the penalty cross on the other side of the field, and give this distance and a $0°$ heading to the kick engine. It is important to note that unlike in games, we align exactly to our target and pass a $0°$ heading to the kick engine in this challenge. Although this takes more time, the added accuracy and consistency is worth the additional time in the challenges.

The use of variable length kicks allowed our team to kick the ball to approximately the same place on the field, regardless of where we kicked the ball from. This allowed us to recover robustly from a poor pass, as we could kick the ball softly if the previous pass barely crossed over the dead zone or with more power if the previous pass ended up near the goal box. Variable length kicks also allowed us to kick the ball any distance within our kicking range, as opposed to only being able to choose from a limited number of predefined kick distances and often having to select a shorter or longer kick than desired.

7.3.2 Results

Twenty teams attempted the passing challenge, but only three teams completed three passes in three minutes or less. We claimed first place with the fastest completion time of 1 minute and 26 seconds, beating the second place team by 17 seconds and the third place team by 62 seconds. A video of the event can be found at `http://www.youtube.com/watch?v=MnjH2Bhbcy4`.

7.4 Dribbling Challenge

In the dribbling challenge, three stationary opponents were placed on the field such that direct kicks to the goal would be unsuccessful. The dribbling robot starts near its goal, and must dribble the ball to the other side of the field and score in three minutes or less. However, the robot must be cautious because the challenge is restarted with the remaining time when the robot kicks the ball out of bounds or into a defender, or if it collides with a defender.

Although the dribbling challenge is designed to encourage development of skills that are useful in games (such as flexible ball manipulation, obstacle detection, and avoidance skills) our process of kick selection for the dribbling challenge is inherently different than the process used in games. For example, in games it is important to minimize the time we spend at the ball. However, in the dribbling challenge, much like in the passing challenge, it is better to take as much time as needed to ensure that our kicker is well aligned to the ball and the correct kick is chosen. As such, kick decision behavior for the dribbling challenge can afford to be slower, and needs to be more conservative in the kicks that it chooses. Due to these differing needs, we opted to base our kick decisions for the dribbling challenge on the location of defenders in an occupancy grid instead of using the kick decision strategy described in Section 6.1.

7.4.1 Using a Relative Occupancy Grid

We implemented a relative occupancy grid to help with kick selection in the dribbling challenge. Our occupancy grid is relative in the sense that it keeps track of where opponents are in relation to our dribbling robot.

7.4.2 Results

Fourteen teams attempted the dribbling challenge, but only five teams successfully completed the challenge in three minutes or less. Requiring one early restart after kicking the ball into a defender, we finished in second place. Our final time in the challenge was 2 minutes and 23 seconds, just 8 seconds slower then the winner.

8 Competition

In 2010, Austin Villa competed in two tournaments, the US Open and RoboCup 2010 in Graz, Austria. Our results in each are described below.

8.1 US Open

The 2010 Standard Platform League US Open was held at Bowdoin College in Brunswick, Maine from April 17-18, 2010. There were three participants: Austin Villa, Northern Bites, and the UPennalizers. The tournament started with a round robin between all the teams on Saturday to determine seeding, followed by a play-in game and championship match on Sunday. In the round robin, Austin Villa beat each of the other teams, scoring 6 goals while allowing none. Austin Villa faced the UPennalizers in the championship and won 3-1. Scores from all of Austin Villa's US Open games are shown in Table 6.

Round	Opponent	Score
Round Robin	UPennalizers	2-0
Round Robin	Northern Bites	4-0
Championship	Northern Bites	3-1

Table 6: US Open 2010 Results

8.2 RoboCup 2010

The 14th International Robot Soccer Competition (RoboCup) was held in June 2010 in Singapore. 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 7. In the first round robin, Austin Villa was in a group with rUNSWift and BURST. Austin Villa tied rUNSWift 3-3 and then beat BURST 8-0 to win the group by goal differential over rUNSWift. In the second round robin, Austin Villa was placed with CMUrfs, Les 3 Mousquetaires, and CHITA Hominids. Austin Villa beat both CMurfs and CHITA Hominids 7-0, and Les 3 Mousquetaires 5-0.

Austin Villa faced Nao-Team HTWK in the quarterfinals and came away with a hard fought 3-1 victory. In the semi-finals, Austin Villa faced the eventual winners B-Human again, this time holding them to 8-0. There was a very fast turn around between the semi final and the third place game. In the third place game, Austin Villa defeated CMUrfs 5-1 to take third place overall.

Round	Opponent	Score
Round Robin 1	rUNSWift	3-3
Round Robin 1	BURST	8-0
Round Robin 2	CMurfs	7-0
Round Robin 2	Les 3 Mousquetaires	5-0
Round Robin 2	CHITA Hominids	7-0
Quarterfinal	Nao-Team HTWK	3-1
Semifinal	B-Human	0-8
Third Place Game	CMurfs	5-1

Table 7: RoboCup 2010 Results

9 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 team improved the speed and effectiveness of the vision module, while working on the full size image. The vision module detects the center circle and opposing robots and is capable of detecting the ball at a greater distance. We developed a parametrized kick that gave us more control of the ball. Using this kick and the Aldebaran's omni-directional walk, we were able to work on a higher level soccer strategy. The team implemented a novel kicking strategy that ensured quick ball movement and progress towards the goal, and this strategy proved to be effective in competition.

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 third place finish at RoboCup 2010. We plan to continue to progress on our codebase and continue to compete in RoboCup in 2011.

References

[1] S. Barrett, M. E. Taylor, and P. Stone. Transfer learning for reinforcement learning on a physical robot. In *Ninth International Conference on*

Autonomous Agents and Multiagent Systems - Adaptive Learning Agents Workshop (AAMAS - ALA), May 2010.

[2] J. Bruce, T. Balch, and M. Veloso. Fast and cheap color image segmentation for interactive robots. In *Proceedings of IROS-2000*. Citeseer, 2000.

[3] M. Hausknecht and P. Stone. Learning powerful kicks on the aibo ers-7: The quest for a striker. In *Proceedings of the RoboCup International Symposium 2010*. Springer Verlag, 2010.

[4] 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.

[5] T. Hester, M. Quinlan, and P. Stone. Generalized model learning for reinforcement learning on a humanoid robot. In *IEEE International Conference on Robotics and Automation (ICRA)*, May 2010.

[6] T. Hester, M. Quinlan, P. Stone, and M. Sridharan. TT-UT Austin Villa 2009: Naos across Texas. Technical Report UT-AI-TR-09-08, The University of Texas at Austin, Department of Computer Science, AI Laboratory, December 2009.

[7] 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.

[8] 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.

[9] 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.

[10] P. Khandelwal, M. Hausknecht, J. Lee, A. Tian, and P. Stone. Vision calibration and processing on a humanoid soccer robot. In *The Fifth Workshop on Humanoid Soccer Robots*, December 2010.

[11] 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.

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

[13] 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.

[14] A. Setapen, M. Quinlan, and P. Stone. Marionet: Motion acquisition for robots through iterative online evaluative training. In *Ninth International Conference on Autonomous Agents and Multiagent Systems - Agents Learning Interactively from Human Teachers Workshop (AAMAS - ALIHT)*, May 2010.

[15] 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.

[16] M. Sridharan and P. Stone. Autonomous color learning on a mobile robot. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, July 2005.

[17] M. Sridharan and P. Stone. Real-time vision on a mobile robot platform. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, August 2005.

[18] 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.

[19] 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.

[20] 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.

[21] P. Stone, M. Quinlan, and T. Hester. Can robots play soccer? In T. Richards, editor, *Soccer and Philosophy: Beautiful Thoughts on the Beautiful Game*, volume 51 of *Popular Culture and Philosophy*, pages 75–88. Open Court Publishing Company, 2010.