Austin Villa 2011: Sharing is Caring: Better Awareness through Information Sharing

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

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

Technical Report UT-AI-TR-12-01

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. Austin Villa reached the quarterfinals at RoboCup 2011 in Istanbul, Turkey before falling to the eventual champions, B-Human. 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 [7].

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: Aldebaran Nao robots 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, Todd Hester, Piyush Khandelwal, and postdoc Michael Quinlan from UT Austin, and Professor Mohan Sridharan from TTU.

We started the codebase for our Nao team from scratch in December of 2007. Our previous work on Aibo teams [10, 11, 12] 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

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

July of 2008. Descriptions of the work for the 2008 tournament can be found in the corresponding technical report [3].

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 the 2009 team report [4].

For 2010, we continued making improvements, updating our architecture, adding a new kick engine, and changing our behaviors and strategies. These changes led us to victory in the 2010 US Open and 3rd place in RoboCup 2010 in Singapore. A full description of these changes can be found in last year's team report [1].

1.2 2011 Contributions

This report describes all facets of our Nao team codebase. For completeness, this report repeats relevant portions of the 2008-2010 team reports. The major change for this year was overhauling the localization and opponent tracking systems to better share information with teammates. The other main changes for 2011 included a new software architecture, kicking engine, and behavior simulation.

Section 2 describes our software architecture that allows for easy extendibility and debugability. Our vision system is described in section 3. Our new localization system (Section 4) is an unscented Kalman filter (UKF). Section 5 describes our new opponent tracking filter which also uses a UKF. Section 6 describes our motion modules used on the robot, including a new kick engine. Section 7 briefly describes the behaviors we developed and employed in 2011, as well as our new behavior simulation. Section 8 describes our soccer results at RoboCup.

1.3 2010-2011 Relevant Publications

Over the past year our team produced two publications involving RoboCup and/or the Nao platform [1, 6].

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:

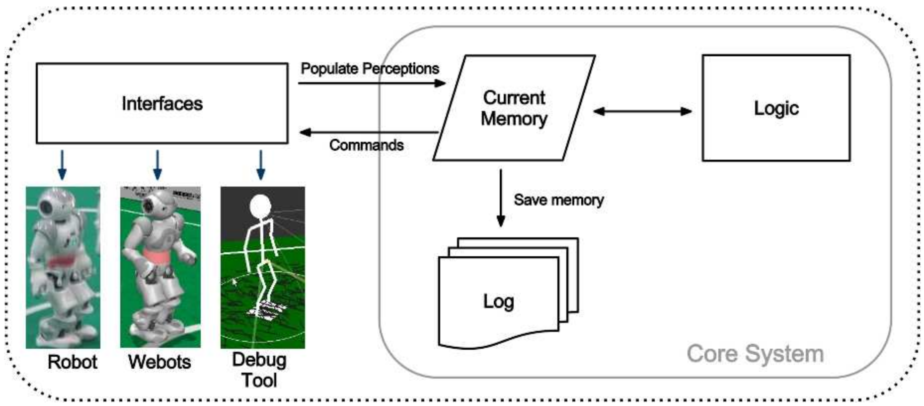


Figure 2: Overview of the Austin Villa software architecture.

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.

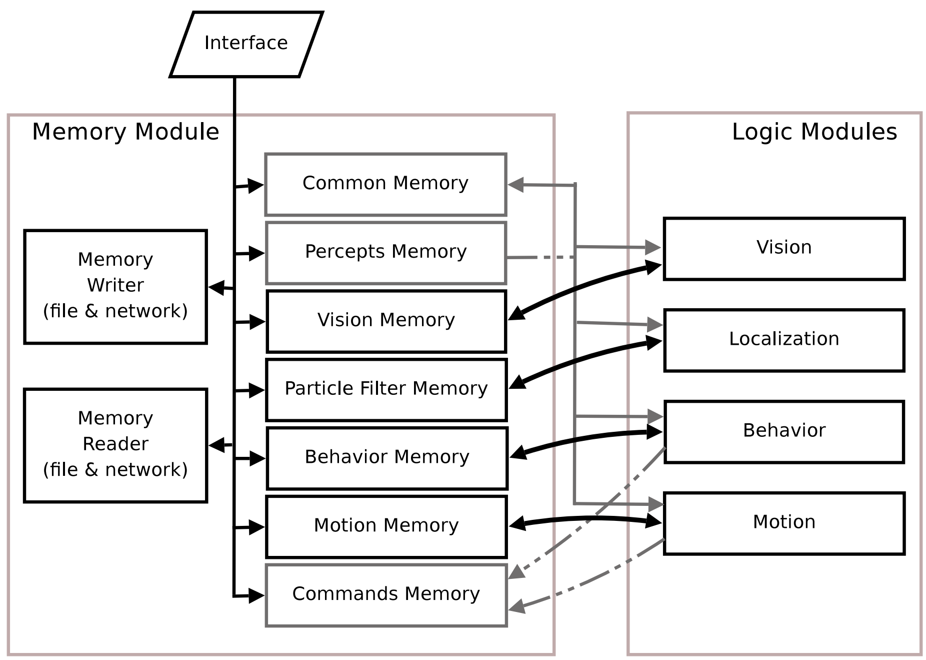


Figure 3: Design of the Memory module. The gray boxes indicated memory blocks that are accessed by multiple logic modules. Dotted lines show connections that are either read or write only (percepts are read only, commands are write only).

,

To increase the modularity of our logic code, we have split the logic into three processes:

1. NaoQi: this code runs at 100 hz, is called by naoqi (or a simulator), and copies the current sensor information to memory and writes out the current motion commands

2. Motion: this code runs at 100 hz and converts high level commands into direct commands for the motors

3. Vision: this code runs at 30 hz and processes the incoming imaging and selects the high level commands

Splitting execution into three processes allows us to restart a crashed process, though only restarting the vision process is currently supported. Using three processes required us to store our memory structure in shared memory. In addition, we use mutex locks to prevent processes from using incomplete information. However, the majority of the information is kept separate for each of the processes, so locking is only required when copying local information into these shared blocks.

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

(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.

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. 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 [9]. 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.

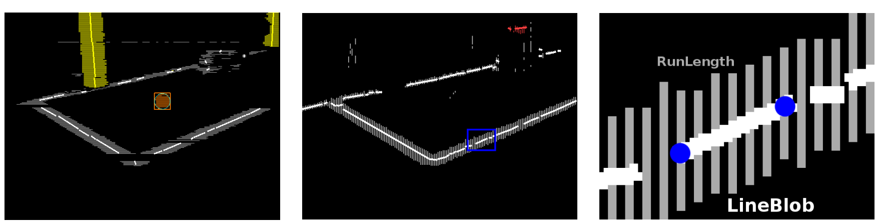
Method	Time (ms)
No optimizations	138.616
Color table reorganization	112.867
Color table reorganization & efficient retrieval	**96.023**

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.

3.1 Blob Formation



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

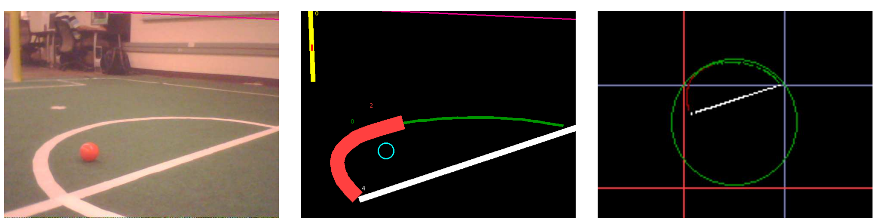
Figure 5: A representative image for the blob formation procedure. An enlarged view of the blue box in Fig. 5(b) is shown in Fig. 5(c) for clarity. In Fig. 5(c), the grey lines are the vertical run lengths and the white line is the corresponding horizontal blob formed.

Our 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



(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).

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

Algorithm 1 Line Segment Formation

```
1: Input: B ← Ordered set of Blobs
2: Output: S ← A set of candidate LineSegments
3:
4: for each Blob b_i in B do
5:    s = new LineSegment
6:    s.initialize(b_i) {Initialize A,B,C. Put blob in segment}
7:    bestSegment = recurse-check-segment (s, b_i, B)
8:    if isAboveThreshold(bestSegment) then
9:       for each Blob b in s do
10:         B.remove(b)
11:      end for
12:      S.add(s)
13:   end if
14: end for
15:
16: function recurse-check-segment (s, b_i, B)
17: bestSoFar = new LineSegment
18: for each Blob b_j in B s.t j > i do
19:    if b_j ≈ s.predictb_j.x_s then
20:       s.updateABC(b_j)
21:       s.add(b_j)
22:       current = recurse-check-segment (s, b_j, B)
23:       if is-better(current, bestSoFar) then
24:          bestSoFar = current
25:       end if
26:    end if
27: end for
28: return bestSoFar
29: end function
```

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

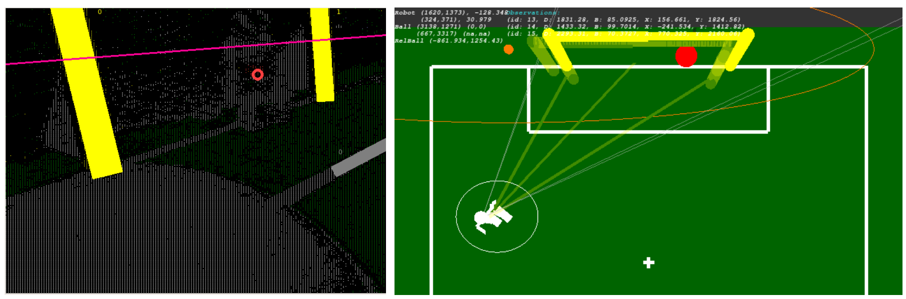
Ball detection runs the Union-Find algorithm [2] on orange blobs from the blob formation procedure to merge them into a set of candidate balls. Our ball detection method is fairly similar to our approaches in previous years [4, 1], and a number of heuristics are employed to calculate the probability 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 last 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 re-run 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 last year, and is used by our kick selection engine to avoid kicking 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 [4]. 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

This year, we changed from using Monte Carlo localization (MCL) to using a multiple model Unscented Kalman Filter (UKF) for localization. The multi-modal UKF is based on previous work by other SPL teams [8, 5]. Our main reason for using MCL in the past was that the particles could represent non-normal distributions over likely locations of the robot on the field. However, it was computationally inefficient, poor at integrating ball information into localization, and bad for sharing information between teammates. While a traditional Kalman filter would not be suited for RoboCup due to to the ambiguous observations and non-Gaussian belief distributions, the multi-modal unscented filter addresses these problems. In addition, we can better integrate the ball's information in with the robot's location by using a 7-state filter and easily share information between robots.

We used a 7 state filter to track both the location of the robot along with the location and velocity of the ball:

$$x(t) = \begin{bmatrix} x \\ y \\ \theta \\ ballx \\ bally \\ ballvelx \\ ballvely \end{bmatrix} \tag{4}$$

As with all Kalman filters, at each time step, a two-step update is performed:

1. *Time Update*: Given the belief of $x(t-1)$, $bel(x(t-1))$, and an odometry update for time t, we calculate $bel(x^-(t))$.

2. *Measurement Update*: Given $bel(x^-(t))$ and the sensor observations for time t, we calculate $bel(x(t))$.

In a standard Kalman filter, the belief over the 7-dimensional state space is represented by a 7-dimensional Gaussian. The multiple model version of the Kalman filter represents the belief by a weighted sum of N Gaussians. Each Gaussian represents a possible hypothesis of where the robot might be. Each of the N models has its own state estimate, x_i, covariance matrix P_i, and weight α_i.

The probability distribution for each model is given by:

$$bel_i(x) = \alpha_i \frac{1}{(2\pi)^{n/2}|P_i|^{1/2}} e^{(-\frac{1}{2}(x-\hat{x}_i)^T P_i^{-1}(x-\hat{x}_i))} \tag{5}$$

The overall probability distribution is simply the sum over all N of these distributions. New models are created when the agent makes ambiguous observations or actions with multiple possible outcomes (e.g. kicks) occur. Models are merged back together when they are deemed to be similar enough using the metrics from [8].

4.1 Unique Observations

When a unique observation is made, each of the N models in the Kalman filter is updated using the standard unscented technique. After all N models are updated, similar models are merged together, using the approach of [8]. Unlike a traditional Kalman filter, which requires a linear measurement update, the unscented Kalman filter deterministically samples points from the distribution and uses these points to update the mean and covariance of the Gaussian. Two sigma points are selected on each major axis of the Gaussian. For the 7-state filter, this results in 14 points, plus the mean of the Gaussian. These 15 sigma points are updated with the new measurement, similar to the way the particles in a particle filter would be updated. These updated points can then be used to compute the new mean and covariance of the Gaussian. In addition to enabling non-linear updates, this unscented filter also means that no Jacobians are needed for various updates, instead each point is updated similar to a particle in MCL. are sampled from the distribution and used to perform a measurement update on that model.

4.2 Ambiguous Observations

When an ambiguous object, such as a single goal post or a field line, is observed, the probability density function over the robot's possible location is no longer a normal distribution. Instead, there is some likelihood of the observed object being any of the possible real objects on the field, and thus, likelihood of the robot being in many different parts of the field. In this case, multiple models are created to represent this distribution.

When an ambiguous observation is made, for each of the existing N models, $M + 1$ copies of this model are made, where M is the number of possible objects that the ambiguous observation could be. One model is updated for each possible object that the observation could be, and the last model is updated for the case that a false observation was made (i.e. an object was detected that was not really there). Each of these observations are given an equally probable likelihood:

$$\alpha_j = \alpha_i \frac{1}{M+1} \tag{6}$$

Following the approach of [8], new models are thrown out immediately if their probability is deemed small enough to be an outlier.

Most of the observations that the robot makes on the SPL field are ambiguous observations. There are only four unique observations that can be made: seeing the entire yellow goal, the entire blue goal, the center circle, or the ball.

Seeing a single goal post, a line, or a line intersection is ambiguous. When seeing a goal post, there are $M = 2$ possible objects (left or right post), when seeing a line, there are $M = 11$ possible lines, and for line intersections, there are $M = 16$ possible objects. As these numbers show, when observing an ambiguous line or intersection, the number of models maintained can increase significantly. If the robot is well-localized, often only one of these possible objects is likely. Therefore, when the robot is confident in its location on the field, the most likely line or intersection is found and an update is performed as if this was a unique observation. When the standard deviation of the robot's location is high enough, multiple models are created for line and intersection observations as well. Goal posts are always treated as ambiguous observations. This approach allows the robot to recover quickly from kidnapping, but be able to maintain tight confidences on its location even when seeing ambiguous observations while localized.

4.3 Ball updates

A major reason for our switch to the multiple model UKF was to improve our ball tracking and information sharing between robots. By using a 7-state filter, the ball's location and velocity are tightly integrated with the robot's location. The robot shares information about the ball's location with its teammates by sharing its estimates of the ball's location and covariance matrix if it has seen the ball recently. The robot's teammates can use this information as a measurement update to improve their estimate of the ball's location. In addition to improving their estimate of the ball's location, this can help improve the robot's own localization. For example, it is often the case that the robot chasing the ball is staring down at it and rarely has other observations. However, if other robots are providing the robot with a confident estimate of where the ball is, then it can use this information along with its observations of the ball to deduce its own location on the field.

We also take advantage of the multi-modal aspect of the filter when kicking. Whenever the robot kicks the ball, two models are created for each of the N models. One model updates the ball with the velocity that should have been imparted by the kick and this model's probability, α, is updated by 0.75. The second model represents the possibility that the kick failed. In this model, the robot's uncertainty about the ball's location and velocity is increased, but it is given no velocity. This model is given a probability of 0.25 of the original model's probability.

5 Opponent Tracking

When an opponent robot is detected through the vision system, the robot needs to be able to track the opponent's location and share it with teammates. This information about the opponent's location is critical in making smart strategic decisions with the ball. Our opponent tracking system is based on the multi-modal UKF system used for localization and ball tracking. A four state filter is employed using the following state:

$$x(t) = \begin{bmatrix} oppx \\ oppy \\ oppvelx \\ oppvely \end{bmatrix} \tag{7}$$

Here, however, the multiple models are used in a different way. The filter maintains N filter models, where each model represents an opponent. Each opponent is represented by only one Gaussian model, rather than a sum of Gaussians.

When an opponent robot is detected, an update is attempted on each of the N models in the filter. This observation update incorporates both noise from the sensor observation and the uncertainties in the robot's own location from localization. If the model that best matched the observation is close enough to meet a set threshold, then that model is updated with the observation. If the best model's match does not meet the threshold, then a new model is created at the location of the observed opponent. Exactly one model is updated for each opponent observation. When multiple opponent observations are made within one time step, each one is used to update a different model. This ensures that multiple observations do not all update a single opponent model and that one observation is not used to update multiple opponent models.

Teammates share information about the opponents by sharing all of their opponent models that have been recently updated. They share both the state and covariance matrix of these models. When receiving this information, the robots take a similar approach as with the measurement updates. Each model from a teammate is compared to all of the robot's current models and the model with the best match is updated. If that match does not meet a given threshold, then a new model is created. By sharing opponent information in this way, team members can successfully avoid and kick around opponent robots without seeing them, requiring only a well-localized 'spotter' robot to tell them where the opponents are.

6 Motion

We used the provided Aldebaran walk engine for our walk, with tuned walk parameters and joint stiffnesses. In 2011 we re-developed our own kick engine, allowing us to use multiple kicks in different directions with varying power.

6.1 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 parameterized 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 5° to the outside of the kicking leg to 30° to the inside, and distances ranging from 0.5 meters to 4 meters.

6.1.1 Basic Control

The robot's kick engine used a 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. Likewise, we found that breaking the pre-kick alignment into two parts also improved stability. 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. We also use splines to compute the path for the foot to follow as it moves in the Kick state.

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 1	Begin to lift the swing leg and position it for the kick
Align 2	Finish lifting and positioning the swing leg for the kick
Kick	Move the swing leg forward 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

6.1.2 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 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, kicking distance and kicking angle; only the Align 2 and Kick states must be altered based on these factors. The Align 2 state must move the foot to the back point of the desired line segment, whereas the Kick state must move the foot to the forward point of the line segment. In both the Align 2 state and the Kick state, the actual distance behind/ahead of the ball and to the side of the ball are determined as functions of the desired kick angle, desired kick distance, and actual ball position with respect to the robot. However, we do bound both the side and back/forward distances of the foot in both the Align 2 state and the Kick state to avoid singular values in the inverse kinematics and to prevent clipping of the stance foot. An example of foot placement for a leftward kick is shown in Figure 8.

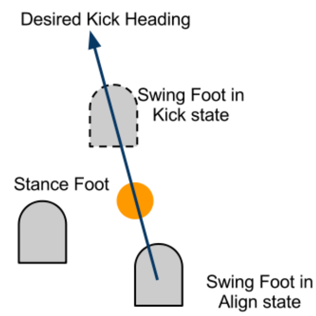


Figure 8: Foot placement during the Align 2 and Kick states.

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.

6.1.3 Distance Control

Kicking distance was controlled by changing the interpolation time of the Kick state, i.e. the time taken to swing the leg forward during the kick. We determined the relationship between this time and the distance empirically, collecting distances at seven interpolation times and then averaging over five kicks at each time. Then, we fit a linear model to the logarithm of the distance, arriving at the model:

$$t = -0.13342 \log(d) + 1.27666$$

We discovered through experimentation that using times of less than 0.17 seconds had no effect on the kick due to the maximum speed of the joint. We also found that using times longer than 0.4 seconds did not reliably result in the robot moving the ball at least 0.5 meters. Hence, all of our kick times were between 0.17 and 0.4 seconds. It is important to note that the function relating distances and interpolation times is dependent on the field surface, so it was necessary to re-calibrate it for the RoboCup competition.

7 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 subtask 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 9 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 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`.

7.1 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,

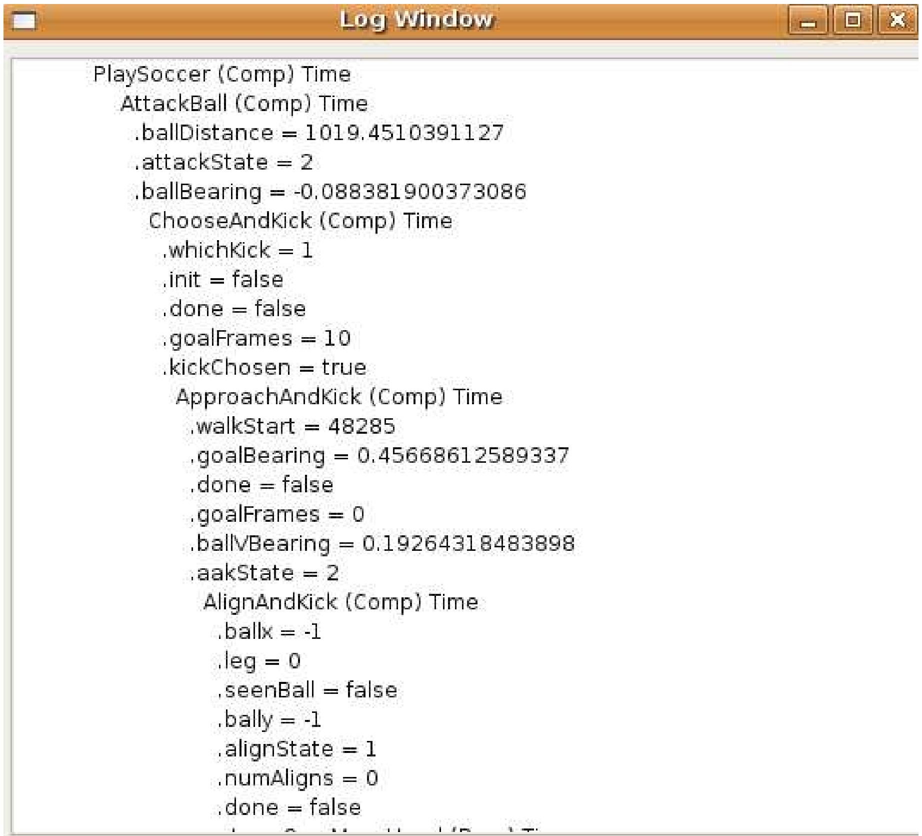


Figure 9: Example Behavior Log, showing the trace of task calls and their state variables.

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 6.1), 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 parameterized 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.

7.1.1 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 10. 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 10. 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.

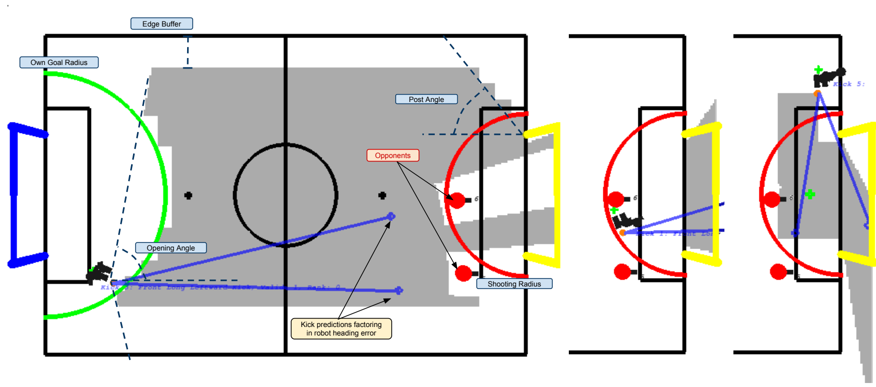
Parameter (unit)	**Description**
Edge Buffer (mm)	Creates a non-valid region of this width on each sideline.
Post Angle (deg)	Shapes the valid region such that it narrows towards the target goal with the given angle.
Opening Angle (deg)	Defines the valid region as angled outwards from the robot, i.e. 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

7.1.2 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. All possible kicks are checked with the opponent filter. All kicks that would go through or stop near an opponent in the filter are invalidated.

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



(a) Kick Region Parameters

(b) Shooting Radius

(c) Corner Behavior

Figure 10: (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.

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.

7.1.3 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 7.1.3 shows 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.

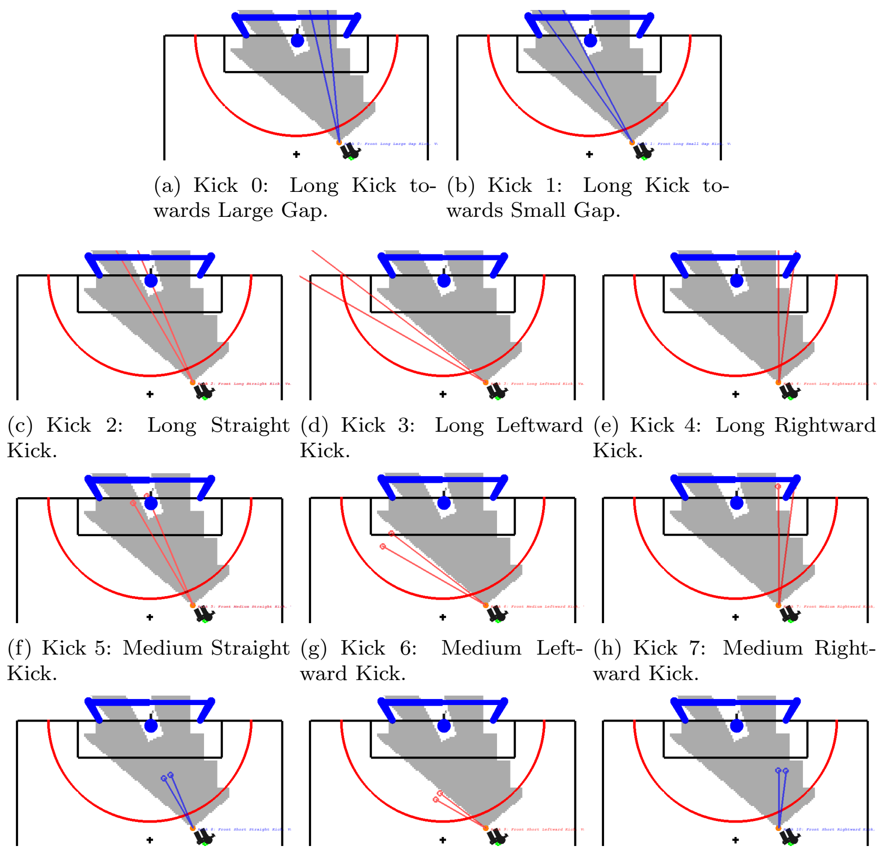
Algorithm 2 Kick Strategy

$b \leftarrow ball$
$r \leftarrow robot$
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}

Kick	Distance (m)	Heading (deg)
Large Gap Kick	3.5	Towards Large Goal Gap
Small Gap Kick	3.5	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

Figure 11 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 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.



(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 11: 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.

7.2 Behavior Simulation

Since we did not have enough robots to run full-team scrimmages, we created a behavior simulation to test our strategies and behaviors. This simulation ran the team's behavior code, providing it with the true positions of the robots and the ball by filling in the positions in memory that would usually be filled in by localization. In addition, the robots shared team information like they would in a game, with outgoing packets from one robot being directly copied into the memory for incoming packets for the other teammates. After the behavior code was run, the robot was moved based on the walk commands that were output, and the ball was imparted a velocity for any kick commands that were given. Noise was added onto the kicks and walks to simulate some of the uncertainty in the real game. In addition, the simulator tracked player pushing and illegal defender penalties, and tracked ball movements, out of bounds calls, and goals, to completely simulate a game.

The simulation could be run in a step-by-step mode where all the information being used by the robot to make decisions was visualized, such as the information it received from its teammates, where it thought all the robots and the ball were, what parts of the field it deemed to be valid kick regions, and the current roles of all members of the team. Stepping through games in this manner allowed us to debug and correct issues with role switching, approaching the ball, and kick decisions.

We could also run the simulation in a 'head-less' mode with no visualizations, running through many simulated games very quickly. This allowed us to try different strategy parameters on each team and see if they created significant differences between the two teams. Finally, we could also alter one team to have stronger kicks or faster walks and see how this altered the best parameters to be used when playing with or against such a team.

8 Competition

In 2011, Austin Villa competed in the 15th International RoboCup Competition in Istanbul, Turkey. Our results are described below. 27 teams entered the competition. Games were played with four robots on each 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 or four 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 teams 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 5. In the first round robin, Austin Villa was in a group with Nao Team Humboldt and the Dutch Nao Team. Austin Villa defeated Humboldt 6-0 and the Dutch team 5-0 to win the group. In the second round robin, Austin Villa was placed with the Nao Devils, Northern Bites, and TJArk. The team defeated the Northern Bites 5-0 before losing to the Nao Devils 2-4 and then defeating TJArk 3-1, to place second in the group behind the Nao Devils.

Placing second in the group meant that the Austin Villa faced the two-time champion B-Human in the quarterfinals. Austin Villa held B-Human to a 5-0 victory (the smallest goal differential of any of B-Human's victories), and hence was knocked out in the quarterfinal round.

Round	Opponent	Score
Round Robin 1	Nao Team Humboldt	6-0
Round Robin 1	Dutch Nao Team	5-0
Round Robin 2	Northern Bites	5-0
Round Robin 2	Nao Devils	2-4
Round Robin 2	TJArk	3-1
Quarterfinal	B-Human	0-5

Table 5: RoboCup 2011 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 its localization and opponent tracking to enable better sharing of information between teammates. In addition, the team employed a new kick engine and utilized a new behavior simulation to improve our decision making strategies.

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. We plan to continue our progress on this code base with a faster walk for RoboCup 2012.

References

[1] S. Barrett, K. Genter, M. Hausknecht, T. Hester, P. Khandelwal, J. Lee, M. Quinlan, A. Tian, P. Stone, and M. Sridharan. Austin Villa 2010 standard platform team report. Technical Report UT-AI-TR-11-01, The University of Texas at Austin, Department of Computer Sciences, AI Laboratory, January 2011.

[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] 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.

[4] 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.

[5] G. Jochmann, S. Kerner, S. Tasse, and O. Ur bann. Efficient multi-hypotheses unscented kalman filtering for robust localization. In T. Röfer, N. M. Mayer, J. Savage, and U. Saranli, editors, *RoboCup 2011: Robot Soccer World Cup XV*, Lecture Notes in Computer Science, page to appear. Springer Berlin / Heidelberg, 2012.

[6] P. Khandelwal and P. Stone. A low cost ground truth detection system using the kinect. In T. Roefer, N. M. Mayer, J. Savage, and U. Saranli, editors, *RoboCup-2011: Robot Soccer World Cup XV*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2012. To appear.

[7] 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.

[8] M. Quinlan and R. Middleton. Multiple model kalman filters: A localization technique for robocup soccer. In *2009 RoboCup Symposium*, 2009.

[9] 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.

[10] 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.

[11] 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.

[12] 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.