The UT Austin Villa 2004 RoboCup Four-Legged Team: Coming of Age

Peter Stone, Kurt Dresner, Peggy Fidelman,
Nicholas K. Jong, Nate Kohl, Gregory Kuhlmann,
Mohan Sridharan, Daniel Stronger

Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, Texas 78712-1188
{pstone,kdresner,peggy,nkj,nate,
kuhlmann,smohan,stronger}@cs.utexas.edu
http://www.cs.utexas.edu/~AustinVilla

Technical Report UT-AI-TR-04-313

October 27, 2004

Abstract

The UT Austin Villa Four-Legged Team for RoboCup 2004 was a second-time entry in the ongoing series of RoboCup legged league competitions. The team development began in mid-January of 2003 without any prior familiarity with the Aibos. After entering a fairly non-competitive team in RoboCup 2003, the team made several important advances. By the July 2004 competition place in Lisbon, Portugal, it was one of the top few teams. In this report, we describe both our development process and the technical details of its end result. In conjunction with our previous technical report [12] this paper provides full documentation of the algorithms behind our approach with the goal of making them fully replicable.

Contents

1 Introduction **4**

2 General Architecture **4**
2.1 WorldState 5

3 Vision **7**
3.1 Camera Settings 7
3.2 Color Segmentation 8
3.3 High-level Vision 10
3.3.1 Threshold and Predicate changes 10
3.4 Line (and line intersection) detection 11
3.5 Position and Bearing of Objects 13
3.5.1 Landmark Distance calculations 13

4 Movement **13**
4.1 Movement Module 14
4.1.1 Walking 14
4.1.2 Head Scans 14
4.1.3 Collision Detection 15
4.2 Movement Interface 15
4.2.1 Mid-level movement tasks 15
4.2.2 Communication with the movement module 16
4.2.3 Odometry 16
4.3 High-Level Control 16

5 Learning Movement Tasks **16**
5.1 Forward Gait 16
5.2 Ball Acquisition 17

6 Kicking **17**

7 Localization **18**
7.1 Distance-based Likelihood Updates 18
7.2 Incorporating Field Lines 18
7.3 Better Motion Model 19
7.4 New Error Model 19
7.4.1 Landmark Histories 19
7.4.2 Two-landmark Reseeding 20
7.4.3 Three-landmark Reseeding 20
7.4.4 Pose Estimate 20
7.5 Active Localization 20
7.6 Simulator 21
7.6.1 Basic Architecture 21
7.6.2 Server Messages 21
7.6.3 Sensor Model 21
7.6.4 Motion Model 22
7.6.5 Graphical Interface 22

8 Communication **23**
8.1 Knowing Which Robots Are Communicating 23
8.2 Determining When A Teammate Is “Dead” 23
8.3 Practical Results 23

9 Behaviors **24**
9.1 Task Hierarchy . . . 24
9.1.1 Primitive Tasks . . . 24
9.1.2 Composite Tasks . . . 24
9.1.3 Example . . . 25
9.2 Goal Scoring . . . 25
9.3 Goalie . . . 28
9.3.1 Finding the Ball . . . 28
9.3.2 Dynamic Positioning . . . 28
9.3.3 Velocity Blocking . . . 29
9.4 Defensive Play . . . 30
9.4.1 Defensive Post . . . 30
9.4.2 Defender cannot go to the ball . . . 31
9.4.3 Defender can go to the ball . . . 31
9.4.4 Avoiding the penalty zone . . . 31
9.5 Active Localization . . . 33

10 Coordination **33**
10.1 Roles . . . 33
10.1.1 Attacker Behavior . . . 33
10.1.2 Supporter Behavior . . . 33
10.1.3 Defender Behavior . . . 33
10.1.4 Dynamic Role Assignment . . . 34

11 UT Assist **35**

12 The Competitions **35**
12.1 American Open . . . 35
12.2 RoboCup 2004 . . . 36

13 Conclusions and Future Work **37**

A Coordinate Transforms **39**
A.1 Camera Transform . . . 39
A.2 Body Transforms . . . 41
A.2.1 Body Tilt and Roll . . . 41
A.2.2 Transform for Back legs . . . 42

B Pixel Projection - Image plane to ground plane **43**

C Line and Line Intersection Thresholds **44**

D Simulator Message Grammar **45**
D.1 Client Action Messages . . . 45
D.2 Client Info Messages . . . 45
D.3 Simulated Sensation Messages . . . 46
D.4 Simulated Observation Messages . . . 46

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 [5].

RoboCup is organized into several leagues, including a computer simulation league and two leagues that use wheeled robots. This technical report concerns the development of a team for the Sony four-legged league[1] in which all competitors use identical Sony Aibo ERS-210A and/or ERS-7 robots and the Open-R software development kit.[2]

Since all teams use the same commercial robots, the four-legged league is essentially a software competition. In this report, we detail the development of our team, UT Austin Villa,[3] from the Department of Computer Sciences at the University of Texas at Austin.

For this report, we assume familiarity with the robots' specifications and the rules of the RoboCup games. For full details, see the legged league and Open-R sites footnoted above. Here we describe both our development process and the technical details of its end result, the UT Austin Villa team. In conjunction with our previous technical report [12] this paper provides full documentation of the algorithms behind our approach with the goal of making them fully replicable.

Our team development began in mid-January of 2003, without any prior familiarity with the Aibos. After entering a fairly non-competitive team in RoboCup 2003, the team made several important advances. By the July 2004 competition place in Lisbon, Portugal, it was one of the top few teams. In the process of making these advances, we also ported our code base to the new ERS-7 robots from the previous ERS-210A robots. In this document we fully describe the changes to our 2003 code base, which we previously documented in full detail [12]. As was the case in 2003, all of our code is our own, with no parts borrowed from other teams' implementations. We include all of our advances as of RoboCup 2004 in July.

The remainder of the report is organized as follows. In Section 2 we review our general architecture that combines sensing, decision-making, and acting. Then we separately treat each of the main low-level subtasks that go into creating a complete robot soccer team: vision in Section 3; movement in Sections 4 and 5; kicking in Section 6; localization in Section 7; and communication in Section 8. Section 9 describes both our new framework for coding behaviors and the soccer-playing behaviors we implemented, such as goal-scoring and goal-tending. Then in Section 10 we document our methods for coordinating the behaviors of the robots as a team. Section 11 introduces our debugging and development tool. Then in Section 12 we summarize our experiences at the American Open and RoboCup 2004 competitions, and Section 13 concludes.

2 General Architecture

The architecture of our soccer-playing agent did not change considerably from last year, but we will recap to provide context for the rest of the report. One significant change was the evolution of the GlobalMap component into the WorldState object, described in Section 2.1.

Three concurrent processes comprise our agent. One of them is the built-in OVirtualRobotComm object,[4] which provides an interface to the Aibo's sensors and effectors. It sends messages to our Brain object, which processes sensory data and messages from teammates. The Brain in turn sends movement commands to the MovementModule, which manages the sending of control signals to OVirtualRobotComm.

We encapsulated all of the code implementing low-level movement (Section 4.1) in the MovementModule object. This module receives Open-R messages dictating which movement to execute. Available leg movements include locomotion in a particular direction, speed, and turning rate; any one of a repertoire of kicks; and getting up from a fallen position. Additionally, the messages may contain independent directives for the head, mouth, and tail. The MovementModule translates these commands into sequences of set points, which it feeds as messages into the robot's OVirtualRobotComm object. Note that this code inhabits its

[1] `http://www.tzi.de/4legged/`
[2] `http://openr.aibo.com/`
[3] `http://www.cs.utexas.edu/~AustinVilla`
[4] For details on the Open-R interface, see the Open-R site footnoted above.

own Open-R object precisely so that it can supply a steady stream of commands to the robot asynchronously with respect to sensor processing and deliberation. For further details on the movement module, see [12].

The Brain object is responsible for the remainder of the agent's tasks: accepting messages containing camera images from OVirtualRobotComm, communicating over the wireless network, and deciding what movement command messages to send to the MovementModule object. It contains the remaining modules, including Vision, Fall Detection, Localization, Communication, and Behavior. These components thus exist as C++ objects within a single Open-R object. The Brain itself does not provide much organization for the modules that comprise it. In large part it serves as a container for the modules, which are free to call each other's methods.

From an implementation perspective, the Brain's primary job is to activate the appropriate modules at the appropriate times. Our agent's "main loop" activates whenever the Brain receives a new visual image from OVirtualRobotComm. Other types of incoming data, mostly from the wireless network, reside in buffers until the camera instigates the next Brain cycle. Each camera image triggers the following sequence of actions from the Brain:

Get Data: The Brain first obtains the current joint positions and other sensor readings from Open-R. It stores this data in a place where modules such as Fall Detection can read them. This means that we ignore the joint positions and sensor readings that OVirtualRobotComm generates between vision frames.

Process Data: Now the Brain invokes all those modules concerning interpreting sensor input: Vision, Localization, and Fall Detection. Note that for simplicity's sake even Communication data waits until this step, synchronized by inputs from the camera, before being processed. The end result of this step is an update to the WorldState object (see Section 2.1).

Act: After the Brain has taken care of sensing, it invokes the task hierarchy, described in Section 9.1, to determine the agent's behavior. This behavior depends only on the contents of the WorldState.

One notable change from 2003 was the transfer of the fall detection code from the Brain to the behavior module. Instead of noticing falls and acting on them in the Brain, a high-level behavior was introduced in the behavior module (see Section 9) that was responsible for noticing and recovering from falls.

2.1 WorldState

The general idea behind the WorldState object is to have a single location where we can store data describing the current state of the world. All of the objects that control the behavior of the Aibo look to the WorldState when they need information, and store any new information that they compute in the WorldState so that other objects can access it easily.

In 2004, our WorldState contained 178 different types of data, not including arrays of data (e.g. a history of accelerometer values over the last 13 cycles) and data structures that include many subfields (e.g. a joints and sensors data structure that contains all of the joint positions and sensor values of the Aibo). Some examples of this data include:

- Time information: the current time, the amount of time spent processing this Brain cycle, the amount of time spent processing the last Brain cycle
- The current score of the game
- Information about the state of each teammate: their positions, roles, ideas about the location of the ball
- Which teammates are still "alive" (i.e. not crashed)
- The position of the Aibo on the field
- The position of the ball and a history of where it has been

- What objects the Aibo currently sees
- Matrices that describe the current body position of the Aibo
- A history of which electrostatic sensors have been activated, and for how long
- All of the communication data, including messages from our teammates and from the game controller

Our WorldState object also provided 132 functions that access this data. Some of these functions return the data in its original form, but most of them process the data to provide a higher-level view of that data. For example, there are a number of ways we can access the position of the Aibo on the field:

- Position in absolute field coordinates
- Position in team-relative field coordinates
- Position relative to some object (e.g. own goal or opponent's goal)
- Fuzzy team-relative position (e.g. on right side of field, near own goal)
- Role sensitive position (e.g. is there an attacker in the opponent's half of the field?)
- Information about the location of the ball with respect to objects on the field (e.g. ourselves, teammates, our own goal)
- Raw angles and distances from various objects on the field to other objects on the field

One notable addition to the WorldState object for 2004 was a distributed representation of the ball. Instead of only having a single position estimate for the ball, this distributed representation used a variable number (in our case, 20) of particles, where each particle represented one possible location for the ball. A probability was associated with each particle that indicated how confident we were in that estimate of the balls position. While this approach was originally inspired by particle filtering (an algorithm used by our localization algorithm – see Section 7) it was implemented as a much simpler algorithm. In order to get the behavior that we wanted out of these "ball particles", we dramatically reduced the effect of the probability update step, instead choosing to update the particles almost entirely by reseeding. Particles were reseeded (i.e. one particle was added to the filter, replacing a low-probability particle) whenever the Aibo saw the ball for 2 consecutive frames or whenever one of its teammates was able to see the ball and communicate that information to the Aibo.

One way that we used the information from these ball particles was by averaging the particles in one angular dimension that centered around the Aibo, so that we could use behaviors that required that we pick a single angle for the ball (for example, in order to look at the ball we would want to know what angle to turn the head to). The method that we used to extract a single ball *position* from the ball particles was to cluster the particles according to their (x, y) coordinates on the field, in a manner similar to that of our localization algorithm (for more information about this clustering algorithm, see the localization section in our 2003 technical report [12]). Having a distributed representation of the ball allowed the Aibos to deal gracefully with conflicting reports of the ball's position caused by vision and localization errors.

Some of the information in the WorldState is shared between the Aibos, using the communication framework described in Section 8. One of the first steps in each Brain cycle involves the Aibo checking for information from other teammates and inserting that information into its own WorldState. At the end of each Brain cycle, each Aibo sends some of its own information to its teammates. The following information is transmitted between Aibos:

- Ping/pong messages - used to determine which Aibos are alive and connected to the network.
- Ball messages - the position of the ball, whether the Aibo currently sees that ball or not, and strategy information about whether the Aibo is currently allowed to approach the ball or not.
- Position messages - the position of the Aibo on the field as well as its current role.
- Command messages - information that instructs the Aibo to change its current role and tells it whether it can approach the ball or not (as elaborated in Section 10).

3 Vision

The ability of the robot to sense its environment is a prerequisite for any decision making on the Aibo. As such, we placed a strong emphasis on the vision component of our team. The vision module processes the images taken by the CMOS camera in the Aibo's nose. The module identifies colors in order to recognize objects, which we use to localize the robot and to plan its operation. By the end of the 2003 competition, we realized that behavior and strategic planning could improve only if the low-level modules were as robust as possible. Plus we faced the problem of porting our code from the old model (ERS-210A) to the new model (ERS-7). Although the ERS-7 image resolution was a bit higher, the quality of the captured images was significantly worse. The images were noisier and contained artifacts such as blue bands of color around the edges of the frame. In the following sections we report our changes to the vision module since the writing of our 2003 technical report [12]. Our visual processing uses the established procedure of color segmentation, followed by object recognition. We shall first elaborate on the changes in the low-level vision module (segmentation, region growing, blob formation), followed by the description of the changes in the high-level vision module (object recognition, calculation of distances and angles to the landmarks). First, we list the camera parameters on the new robot model (ERS-7).

3.1 Camera Settings

The SONY ERS-7 robot comes equipped with a CMOS color camera that operates at a frame rate of $30fps$. Some of its other preset features are:

- Horizontal viewing angle: 56.9°.
- Vertical viewing angle: 45.2°.
- Lens Aperture: 2.8.
- Focal length: 3.27mm.

We have partial control over three parameters, each of which has three options from which to choose:

- $WhiteBalance$: We are provided with settings corresponding to three different light temperatures.

  1. *Indoor-mode*: 2856K.
  2. *FL-mode*: 5000K.
  3. *Outdoor-mode*: 6500K.

  This setting is basically a color-correction system to accommodate varying lighting conditions. The idea is that the camera needs to identify the 'white point' (such that white objects appear white) so that the other colors are mapped properly. We found that this setting does help in increasing the separation between colors and hence improves object recognition. The optimum setting depends on the 'light temperature' registered on the field, which in turn depends on the type of light used, i.e, incandescent, fluorescent, etc. For example, in our lab setting, we noticed a better separation between orange and yellow with the *Indoor* setting than with the other settings. This helped us distinguish the orange ball from the other yellow objects on the field such as the goal and sections of the beacons.

- $ShutterSpeed$:

  1. Slow: $1/50sec$.
  2. Mid: $1/100sec$.
  3. Fast: $1/200sec$.

  This setting denotes for how much time the shutter allows light to enter the camera. The higher settings (larger denominators) are better when we want to *freeze* the action in an image. We noticed that both the 'Mid' and the 'Fast' settings did reasonably well, though the 'Fast' setting seemed the best, especially considering that we want to capture the motion of the ball. Here, the lower settings tended to result in blurred images. From a qualitative viewpoint, increasing the shutter speed on the camera makes the images *darker*.

- *Gain:*
  1. Low: $-6dB$.
  2. Mid: $0dB$.
  3. High: $6dB$.

  This parameter sets the camera gain. Again, from a qualitative viewpoint, the higher gain makes the image look *brighter*.

For most of our games we ended up using the combination of $WhiteBalance =$ *Indoor-mode*, $ShutterSpeed = Fast$ and $Gain = High$. Depending on the lighting at the competitions, we occasionally had to play with the parameters to get a better performance (for example, reduce the *Gain* if the field of view is too bright).

3.2 Color Segmentation

Low-level vision involves two main steps: creating a color map to segment the image, and forming run-regions and blobs from the segmented pixels (see [12] for more details). In our color segmentation algorithm from last year, the YCbCr was used exclusively throughout the process. We continue to use this color space as part of our current approach. However, based on experimental results published in the robot rescue domain [4][8], we decided to incorporate the LAB color space as well into color segmentation. This color space is spherically symmetrical and is related to the YCbCr space by a non-linear transformation. It has the advantage of being more robust to illumination changes than the other common color spaces (e.g. RGB, YCbCr, and HSV).

The input image from the robot's camera is obtained in the native YCbCr format. In the off-line training process, to generate the color map [12], each image obtained in the YCbCr format is converted to the corresponding LAB image. This conversion from YCbCr to LAB involves the conversion of the YCbCr values to the corresponding RGB (*rgb*) [5] values first, which are then converted to the LAB color space values. This two-step conversion can be represented by the following sets of equations:

- The conversion from YCbCr to *rgb* color space is a linear transformation [2]:

$$\begin{aligned} y = Y - 16, \qquad cb &= Cb - 128, \qquad cr = Cr - 128 \\ r &= 1.164384 * y + 1.596027 * cr \\ g &= 1.164384 * y - 0.391762 * cb - 0.812968 * cr \\ b &= 1.164384 * y + 2.017231 * cb \end{aligned} \tag{1}$$

- The conversion to LAB is then accomplished by transforming from the *rgb* color space by the following nonlinear transformation.

$$\begin{aligned} L_{int} &= \sqrt{(r * r + g * g + b * b)} \\ L &= \frac{L_{int}}{Range_L} \\ A &= \frac{\frac{180}{\pi} * \arccos(\frac{b}{L_{int}})}{Range_A} \\ B &= \frac{\frac{180}{\pi} * \arctan(\frac{g}{r})}{Range_B} \end{aligned} \tag{2}$$

[5] The *RGB* space is referred to as *rgb* in the equations and the figure only to avoid confusion with the *LAB* space defined later.

where $Range_L$, $Range_A$ and $Range_B$ are constants along the three dimensions. They define the bins over which the values along the three dimensions are discretized. For example, the angular component A can take values between 0^o and 90^o. So a value of $Range_A = 2$ implies that the corresponding values are discretized into 45 bins. This helps reduce the storage requirements.

Figure 1 shows a mapping between the *rgb* and LAB color spaces. For any point in the *rgb* color space,

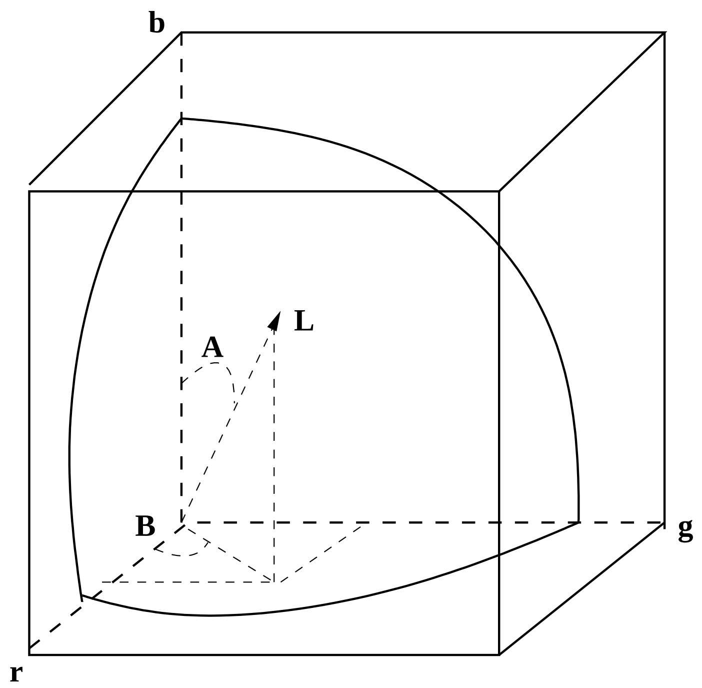

Figure 1: The transformation from the the rgb to the LAB color space.

the length of the vector provides the value of L. The angle made by this vector with the axis b is the angular component, A, while B is the angle between the projection of the vector on the $r - g$ plane and the axis r. As seen in the figure, we use only a section of the sphere for our calculations. For more details see [8].

The initial off-line color map generation is then summarized as follows:

1. Capture images from the robot's camera (YCbCr format).

2. Convert each image from the YCbCr to the LAB color space.

3. Hand label regions in the captured images into one of the 10 colors that the robot needs to be trained to recognize so as to function in its environment.

4. This hand labeling maps into the 3D color map (along the L, A, B dimensions).

5. The final color map is generated by performing a Nearest Neighbor (weighted average) analysis on the color map generated in the previous step. This enables us to limit the noise and edge based artifacts.

Further details on the color cube generation can be found in our 2003 technical report [12].

Under normal conditions, we would have loaded this color cube onto the robot's memory stick, which would then be used by the robot to segment all the input images. But this would involve the conversion of each pixel of the image into the LAB color space (from YCbCr) which is computationally expensive to perform on the robot considering the limited processing power available at our disposal. To avoid this, but still retain the positive benefits of the LAB color space, for each possible YCbCr color space cell (128 * 128 * 128 cells of the color cube), we predetermined the mapping to the corresponding LAB color space cell. Then by indexing into this cell of the already determined color cube (LAB space), the corresponding color value (0-9) is assigned to the corresponding cell of the YCbCr color cube. In this manner the color map is determined in the original YCbCr color space and this is loaded onto the robot's memory stick to be used for the color segmentation process. Based on informal experimentation, we found this color cube to perform better than the color cube in the original YCbCr color space. Based on informal comparisons with some other teams at the competitions, we also had lower occurrences of false ball sightings in beacons and red uniforms, both at the US Open and at RoboCup04.

In the next stage, run-region and blob generation, the contiguous constant-colored *blobs* in the image are identified as potential objects [12]. Here, we made several modifications both to speed up the processing and also to make it more robust to noise. For example, we stopped generating the run-regions and blobs corresponding to the "white" and "field-green" colors, because, in most images, these colors occupied a considerable amount of an image area. The white and field-green regions were used in the identification of the border and field lines that are useful inputs of the localization module, but we implemented an alternate method of detecting lines in the input image that is considerably faster (see Section 3.4). We also experimentally adjusted several thresholds governing the selection of the candidate blobs to be considered for object recognition. We had to do these anyway because we had to eventually port the code from the ERS-210A to the new ERS-7 model.

3.3 High-level Vision

Several changes were made in the high-level vision module, which involves the detection of objects from the colored blobs generated in the low-level vision module. Based on experimental observations, we modified several threshold values and added several predicates to improve the detection. We had to strike a balance once again between rejecting the artifacts (false positives) and identifying the objects better when they were only partially seen. Another important change was the detection of lines (field lines, field borders) and line intersections which were later used in localization. Further, we had developed a simulator for the localization module (see Section 7.6), which required the modification of the structure of the vision-localization interface such that the calculation of the distances and angles to landmarks and ball was performed in the high-level vision module. We also modified these calculations to take into account the camera and robot's body tilt, pan and roll parameters. The details of all these modifications are provided in the sections below.

3.3.1 Threshold and Predicate changes

Here we describe the changes made in the thresholds and also describe some of the predicates that were defined to make the object detection stage work better. First, let us look at some of the predicates in detail.

- *Tilt test:* For the objects that are expected to be in contact with the ground, namely the goals and the ball, we added predicates to ensure that candidate blobs that were too high (or too low) in the visual field were not misclassified as the objects of interest. Previously we calculated the compensation required for the camera tilt in the camera frame of reference, and then decided whether the candidate goal blob could actually be an object of concern. But this does not take into account the roll of the camera or the robot's body tilt and roll. Also, this year, we modified all landmark distance and angle calculations to provide measurements from the center of the robot's body (i.e., we chose this as our reference point). So, we incorporated the body and neck parameters in the calculation of the compensation (see Appendix A) and then determined if the candidate blob was too high (or low) to be an object of interest. We did this by calculating the net tilt and roll of the camera taking into account all the parameters that could affect it.

- *Ball-Floating-Above-Borders:* Last year, our code (at the competitions) did not include the routines needed to detect (and hence use) the borders and the field lines. But soon after the competitions, we tested and incorporated this in our code base (see Section 3.4). This also allowed us to define several other predicates, which proved highly useful in several of our modules (localization, behavior, strategy, etc.). Especially at the RoboCup2004 competition in Lisbon, the colors of the floor and walls around the playing field were close to that of the orange ball. This caused confusion even with the tilt test in place, considering the noise in the sensor readings and the fact that the robot had to be able to see the ball from a distance. In this case, we were able to very easily add another predicate that rejected any candidate ball blob if it happened to occur above a field border. This helped to dramatically reduce the occurrence of false positives.

Several other predicates that we use to reject false object detections exist separately for the beacons, ball, and goals. Their semantics are clear from their names. Some of them are listed below:

- *Beacon*
  - *Beacon-Too-Big.*
  - *Possible-Beacon.*
  - *Flying-Beacon.*
  - *Noisy-Beacon.*
  - *Low-Prob-Beacon.*
- *Ball*
  - *Ball-Aligned-In-Beacon*
  - *Ball-In-Around-Yellow-Goal*
  - *Ball-Floating-Above-Borders*
  - *Ball-Too-High*
- *Goal*
  - *Goal-In-Beacon*
  - *Goal-Above-Borders*
  - *Goal-Too-High*
  - *Goal-Too-Low*

3.4 Line (and line intersection) detection

As mentioned above, we do not form the blobs corresponding to the white and field green colors because it would be too computationally intensive. The main potential use of these colors is to identify the field edges. For localization, the distance to a particular line type provides a lot of information, especially when the robot is running after the ball and is unable to look at landmarks. Also useful are the relative positions of the line intersections (field-field, border-border and field-border), as the global positions (field frame of reference) of the line intersections of each type are known. Common edge detection algorithms such as Hough Transforms were also not used for the same reason. Some fast approximation algorithms do exist for edge detection but we do not use them. Instead, we use a simple scanning and line fitting procedure to identify the lines in the visual field. This procedure is a lot simpler and faster but provides just as accurate line detection. Below, we describe our approach to line and line intersection detection. This section is not a prerequisite to understanding the localization algorithm (Section 7), so the reader not interested in the details of line detection can skip to Section 3.5.

Edges in the robot's environment (the field) that are of interest are characterized by a clear green-white transition (field borders) or a green-white-green transition (field lines). The process of line detection starts by first finding the candidate line pixels in the image. To do so, we perform vertical scans in the image, along lines that are five pixels apart, looking for the green-white transitions (we do a bottom-top scan). Once such a pixel is found we determine if it can be a pixel on a border or field line by incorporating "sanity checks" and special-purpose predicates (described briefly in Appendix C). First a scan is performed along the line for a certain number of pixels below this pixel to ensure if sufficient amount of "field-green" exists below this pixel. The next step involves projecting this pixel on the ground, to determine the distance of the pixel from the robot (center of its body), along the ground. Pixels that do not result in "legal" projections (for example, extremely large or extremely small displacements relative to the robot due to noise near the image center) are rejected from further analysis. This projection is performed using the transformation matrices that incorporate the camera and robot body tilt, pan and roll (see Appendix A and Appendix B). This projection provides the (x, y) position of the pixel with respect to the robot. Then we perform some sanity checks to reject pixels that could have been formed as a result of image artifacts. We reject pixels that are beyond a certain distance from the robot and a certain pixel offset in the image plane [6].

Pixels that project into legal points and pass the sanity checks are then classified into four line categories: border line, field line, blue goal line and yellow goal line. The last two options correspond to the field lines directly below the goals that have a significantly large number of goal-colored pixels above the candidate pixels. We perform a scan (of a fixed number of pixels) above the candidate pixel and maintain a count of the different colors. It is relatively easier to identify the blue goal and yellow goal line pixels (if they exist). But to distinguish between the border and field line pixels, we needed to incorporate another predicate. Using the distance of the pixel (along the ground plane) and the known width of the border and field line in the global frame (currently 100mm and 25mm respectively), we determine the expected width of the two different kinds of lines in the image plane at the distance under consideration. Then, by comparing the expected and the observed widths, we distinguish between the border and the field line pixels. By this procedure, we classify the pixel as belonging to one of the four line (pixel) types. Each candidate edge pixel therefore stores the information regarding the image (and ground plane) position(s) and the line pixel type.

Once line pixels have been determined, we fit lines to the candidate edge pixels. This fitting is performed in the image plane since the original edge points were determined in this space. To reduce the effects of noise, we allow for only one edge pixel along each vertical scan line. This does have the disadvantage of not allowing us to determine multiple lines if they lie one below the other but the bottom-top scanning procedure does enable us to detect the line that is the closest to the robot. A similar problem occurs when the robot's camera has a significant roll; a solution to this problem requires a scan based on the net roll of the robot's camera but we did not have time to do this before the competitions. Lines are fit separately and incrementally to edge pixels of each type using the Linear Line Fit principle and mean square error minimization [1].

We do include noise filtering by incorporating thresholds. For example, an edge pixel is considered to be along a previously constructed line only if it is within a threshold distance from this line. Also, a candidate line is finally accepted for further processing only if it involves more than a threshold number of edge pixels. At the end of this process we have lines corresponding to the edge pixel types. Each such candidate line therefore stores the information regarding the line (a, b, c coefficients that define a line of type ay + bx + c = 0), the line type (field line, borders and goal edges) and also its starting and ending point in the image plane.

The next step is the determination of the line intersections. The intersections that are of importance with respect to the localization of the robot (especially the keeper) are the intersections between two field lines (FF), between two field borders (BB) and between a field line and a field border (FB). We determine the line intersections using the following procedure:

1. The various lines found in the current image (if any) are grouped in a list.
2. Each line in the list is checked for a possible intersection with every other line in the list. We extend the lines if required to determine if they intersect because sometimes only part of the line is identified though the entire line is actually visible.
3. Once a pair of lines is observed to be part of an intersection, details of the intersection are stored and the lines are removed from further analysis.
4. The intersection categories are assigned based on the categories of the lines that constitute the intersection.

The information on intersections is stored separately. Corresponding to each intersection, we store the position in the image plane and on the ground plane (relative to the robot) and also store the intersection *type*, i.e., FF, BB or FB. At this stage we do have some sanity checks for noise filtering.

- A line intersection should exist on the image plane or within a certain threshold buffer around it.
- An intersection is accepted only if the lines under consideration make an angle (absolute value of the acute angle between the lines) that is greater than an experimentally determined threshold.

[6]The distance threshold = $1800mm$ while the maximum offset value corresponds to the robot's horizontal field of view.

- The projection of the intersection onto the ground plane (relative position of the intersection point with respect to the robot frame of reference) should lie within a certain threshold distance (based on the lines involved in the intersection) of the robot.

Some more details on the thresholds are in Appendix C.

3.5 Position and Bearing of Objects

The last stage of high-level vision provides the input to the localization module (Section 7). The input is provided in the form of distances and angles to the fixed markers in the robot's environment (playing field). The localization module uses these inputs to determine the global location of the robot in the field. In this section we describe the process by which the distances and angles to the landmarks/markers are determined.

3.5.1 Landmark Distance calculations

As mentioned at the beginning of this section we perform the calculation of the distances and angles to all objects of interest in the visual frame (landmarks, lines, ball, opponent etc) in the high-level vision module and pass on these observations to the localization module to use as required. After the competitions last year several modifications were made:

1. Distance and angle calculations were made with respect to the center of the robot's body by incorporating the net tilt, pan and roll using all the camera and robot body parameters (accelerometers). Transformation matrices made this calculation easy and efficient to perform (see Appendix A). For more details on the actual estimation of distances and angles to objects based on the image input, see our technical report from last year [12].

2. We observed that the distances to landmarks were consistently underestimated. We fit functions to account and hence compensate for this bias. The functions were polynomials (we used cubics) whose coefficients were determined experimentally (see [1] for more details). For further details on this process and also on the use of landmark estimates in localization, see the corresponding section in the localization module; Section 7.1.

For the line intersections and a few sample line points (Section 3.4), the distance and angle calculations were already performed when the corresponding objects were generated. So the relative positions were used to just generate the distances and angles that formed the corresponding observations. Remember that these already include the transformations mentioned above.

For the landmarks, the relative distances and angles were determined in the camera frame of reference using the method described in the report from last year [12]. After that, the transformation matrices were used to determine the distance and angles in the frame of reference at the center of the robot's body. In addition, the correction functions were applied to the calculated distances. The final values, with a corresponding object specification (an ID which specifies the landmark under consideration), were stored as observations to be used in the localization module.

4 Movement

Enabling the Aibos to move precisely and quickly is just as important to the overall RoboCup task as is vision. In this section, we introduce our approach to Aibo movement, including walking and the interfaces from walking to the higher level control modules.

Control of the Aibo's movements occurs at three levels of abstraction.

1. The lowest level, the "movement module," resides in a separate Open-R object from the rest of our code (as described in the context of our general architecture in Section 2) and is responsible for sending the joint values to *OVirtualRobotComm*, the provided Open-R object that serves as an interface to the Aibo's motors.

2. One level above the movement module is the "movement interface," which handles the work of calculating many of the parameters particular to the current internal state and sensor values. It also manages the inter-object communication between the movement module and the rest of the code.
3. The highest level occurs at the level of the behaviors themselves, where the decisions to initiate or continue entire types of movement are made.

4.1 Movement Module

The movement module has changed very little since last year [12]. There have been a few innovations worth noting, however, including the introduction of an omni-directional walk, a new head scan which is controlled at this level rather than in the movement interface, and collision detection during some types of movement.

4.1.1 Walking

Our walking machinery has not changed much since last year [12]. The inverse kinematics and parameters used at the lowest level are the same. However, we have made changes to the way the robot chooses parameters for arbitrary combinations of attempted forwards, sideways, and turning velocities. For this purpose, we treat a set of walking parameters (corresponding to one continuous walking motion) as a 17-dimensional vector, and take weighted averages of these vectors to attain the walking motion used at any point. The 17 parameters define the timing and locus of the motion of the feet, and they are described in detail in [12].

One adjustment is made to the parameter space before the interpolation is performed. The parameters d_{fwd}, d_{side}, and d_{turn} determine the length and direction of the steps being taken by the four legs. The parameter $MovingCounter$ specifies the length of time taken in each walking cycle. Before interpolating, the parameters d_{fwd}, d_{side}, and d_{turn} are replaced with $d_{fwd}/MovingCounter$, $d_{side}/MovingCounter$, and $d_{turn}/MovingCounter$. Then, instead of interpolating between step sizes, and between step times, we interpolate to get directional velocities and step times, and then convert these back into step sizes. We believe this process yields somewhat more accurate velocities.

The omni-directional walk takes as input a desired forward velocity, yv, a desired sideways velocity, xv, and a desired turning velocity, $thetav$. The walking module uses these values to interpolate between pre-set walks. The pre-set walks are tuned by hand (or in one case via machine learning — see Section 5), and they are the walking parameters that cause the robot to move as fast as possible in the cardinal directions (forwards, backwards, left, right, clockwise, and counterclockwise), plus one set of walking parameters that steps in place. The speeds of all of the cardinal walks are measured manually, and their speeds are stored as constants called YV_{MAX} (forwards), YV_{MIN} (backwards), XV_{MAX} (rightwards), and so on.

To compute the walking parameters corresponding to yv, xv and $thetav$, we interpolate between the idle walk and the cardinal walks in the directions indicated by the signs of yv, xv, and $thetav$. For example, if yv and $thetav$ are positive, and xv is negative, we would use a combination of IdleWalk (which steps in place), ForwardsWalk (for positive yv), LeftWalk (for negative xv), and CounterWalk (for positive $thetav$). The vector used would be given by:

$$\begin{aligned}\text{IdleWalk} \quad &+ \quad \frac{yv}{YV_{MAX}}(\text{ForwardsWalk} - \text{IdleWalk}) \\ &+ \quad \frac{-xv}{XV_{MAX}}(\text{LeftWalk} - \text{IdleWalk}) \\ &+ \quad \frac{thetav}{THETAV_{MAX}}(\text{CounterWalk} - \text{IdleWalk})\end{aligned}$$

4.1.2 Head Scans

In 2003, all of the Aibo's head movements were controlled at the highest behavioral level. This level of abstraction placed rather low-level tasks, such as scanning the head from side to side, on the same level as high level tasks, like deciding where to move on the field. Our 2004 code moved simple head behaviors into the movement module at a lower level of abstraction. This change freed our high-level movement code from the responsibility of having to explicitly move the head through a series of positions to perform a simple action like a head scan. In addition, the faster frame rate of the movement module meant that the head movements could be requested with finer granularity, resulting in smoother actions. In order to implement this change, the movement module interface had to be expanded to include requests for different types of head scans. In particular, at some times we used a left-right scan of the head, and at others we used a circular scan.

4.1.3 Collision Detection

Collisions during the "chin pinch turn" (the motion which moves the ball by having the robot turn in place while pinching the ball under its lowered head — see Section 5.2) can now be detected by the movement module. When a collision is detected, this information is communicated to the world state via the same connection that updates the main Open-R module about the movement module's internal state. It can then be used to make behavior-level decisions and improve the accuracy of localization (by improving the accuracy of the odometry estimate). Collision detection is accomplished by a statistical method very similar to the one introduced by Quinlan et al.[9]. As the first step, the robot walks around the field, following the ball, and a human leads it around (by moving the ball) in such a way that the robot is never colliding.[7] Whenever the robot does a chin pinch turn, it records its leg joint angles at 8 specific points in each walk cycle. This data is then used to calculate a mean and standard deviation for each joint's position at each of the 8 times in the walk cycle. Later, when the robot is doing a chin pinch turn, the actual joint readings are compared against the statistics calculated earlier. If joints on at least 2 of the legs give readings which are more than 2.3 standard deviations away from the mean for that joint at that time in the walk cycle, then the system declares that a collision has been detected.[8]

4.2 Movement Interface

The movement interface is the portion of the movement code which resides in the same Open-R object as the behavior code and abstracts away most of the details of interaction with the movement module, including calculation of movement parameters and inter-object communication with the movement module. This level of the movement system has changed organization quite substantially since last year. Whereas it was previously contained in a single module separate from the high-level behaviors, it is now mostly handled by parts of the task hierarchy itself, and its various functional parts are located in different parts of the code.

4.2.1 Mid-level movement tasks

In the task hierarchy, tasks specifying behaviors at a sufficiently low level as to require sending commands to the movement module are, by definition, "primitive tasks" (see Section 9.1). Thus, all mid-level movement behaviors can be found in this form. In the general case, a primitive task accepts parameters from higher-level tasks.

Primitive tasks cover basic movements such as walking (parameterized by direction and velocity), turning left or right, moving the head to a position, scanning the head in the various ways defined in the movement module, opening or closing the robot's mouth, kicking (parameterized by type of kick), stopping the legs and/or head, and getting up from a fall. They also provide a few functions for more complex movements, such as:

- **Look At Ball:** This function moves the robot's head so as to keep the ball (or another object) in the center of the robot's visual field.

[7]At first, we tried to gather this data by having the robot simply turn in place with the chin pinch turn for several minutes. However, we found that the joint angles observed in this scenario were significantly different from those observed when the robot did a chin pinch turn for only a few seconds after walking around the field after the ball. Since this latter scenario is more like a game scenario and thus more closely corresponds to the circumstances under which we will want to be able to detect collisions, we decided to gather data this way instead.

[8]These parameters were chosen by informal experimentation, with the specific aim of reducing false positives as much as possible.

- **Walk Towards Angle:** This function computes a walk for the robot which will move the robot towards a specified angle relative to its current heading. It can be used to follow the ball or to walk to a specific position on the field.
- **Chin Pinch:** This function takes care of the process of turning the ball to face a certain angle, including stopping the turn at the right time. There are two variants of this function: one which is closed-loop, in the sense that it consults all localization data to determine when to stop, and another which is open-loop, in the sense that it relies only on odometry.

4.2.2 Communication with the movement module

Commands are sent to the movement module via a `struct` known as a `MoveParamData`. The task hierarchy maintains a single instance of this structure. In every Brain cycle, each executing primitive task will change some of the fields of this `MoveParamData` so that they reflect the movement module command(s) corresponding to the action this primitive task is currently carrying out. After all executing primitive tasks have had the chance to change the `MoveParamData`, it is sent to the movement module.

4.2.3 Odometry

As the Aibo walks, it keeps track of its forward, horizontal, and angular velocities. These values are used as inputs to our particle filtering algorithm (see Section 7) and it is important for them to be as accurate as possible. Odometry information for many of our walks is determined by direct, manual measurement. For the omni-directional walk, the velocity is assumed to be the velocity that was requested and is being attempted. During execution, the fields of the `MoveParamData` maintained by the task hierarchy are used to update the robot's world state as to the odometry of the currently-executing movement.

4.3 High-Level Control

The highest level of motion control occurs in what are known as the "composite tasks" (see Section 9.1.2) of the task hierarchy, along with control of the robot's general behavior. At this level, motion control consists of invoking primitive tasks appropriate to the robot's current highest-level state and objectives. For instance, if the robot currently holds the ball under its chin and wishes to shoot at the opponent's goal, the composite task which is currently executing will invoke the primitive task that executes the chin pinch turn, after setting its parameters for which direction to turn and what angle to stop at. We give a full treatment of our high-level control in the context of our robots' "behaviors" in Section 9.

5 Learning Movement Tasks

One major research focus in our lab is machine learning. Learning on physical robots is particularly challenging due to the limited training opportunities that result from robots moving slowly, batteries running out, etc. In this section, we briefly mention two of our learning-research results that have been usefully incorporated into our team development.

5.1 Forward Gait

Our specification of the Aibo's gait left us with many free parameters that required adjustment. While it was possible to hand-tune these parameters, we thought that machine learning could provide a more thorough and methodical approach. As shown in Figure 2, our training environment consisted of multiple Aibos walking back and forth between pairs of fixed landmarks. The Aibos evaluated a set of gait parameters that they received from a central computer by timing themselves as they walked. As the Aibos explored the gait policy space, they discovered increasingly fast gaits. This approach of automated gait-tuning circumvented the need for us to tune the gait parameters by hand, and proved to be very effective in generating fast gaits. Full details about this approach are given in [6].

Figure 2: The training environment for the gait learning experiments. Each Aibo times itself as it moves back and forth between a pair of landmarks (A and A', B and B', or C and C').

5.2 Ball Acquisition

To transition from approaching the ball to performing the chin pinch turn with it, the Aibo must *acquire* the ball so that it is securely beneath its head. This process of acquisition is very delicate and brittle, and previously has relied on repeated hand-tuning. For this reason, we chose to automate this tuning process as well, using some of the machine learning algorithms that worked well for tuning the forward gait.

The training environment for this task consists of a single Aibo on a field with a single ball. It repeatedly walks up to the ball, attempts to acquire it, and knocks it away again before starting the next trial. This was an effective way to generate a reliable acquisition behavior, and it was used with some success at competitions to accomplish the re-tuning necessitated by walk variation on the new field surfaces. Full details about our approach to learning ball acquisition can be found in [3].

6 Kicking

Kicking is one of the most fundamental soccer skills. The robot's kick is specified by a sequence of poses. While this pose-oriented method of executing kicks did not change from 2003, we did add several new kicks to the Aibos' repertoire.

To assist us in developing kicks, we created *Vogue*, a tool which allows us to create and modify kicks interactively. Vogue is implemented as a task in our task hierarchy (see Section 9.1). It uses a simple telnet interface, and the user can edit or create the poses that make up a kick by either entering joint angles or posing the robot manually. It uses the movement module (Section 4.1) to execute all created kicks. We used Vogue to create the following new kicks.

Fall kick. In 2003, we chiefly used a kick that focused on rearing up and falling heavily on the ball. Amid worries of damaged Aibos, we designed a new kick in a similar spirit that was slightly less violent. Instead of letting the Aibo's chest hit the ground, this modified kick used the front arms of the Aibo to absorb some of the shock of the fall. Another addition to this kick involved a pre-kick ball-grab in which the front arms center the ball, which greatly increased the accuracy of the kick.

Head kick. The head kick that we used in 2003 moved the ball either to the right or left of the Aibo, somewhat resembling a golfer's putting stroke. We redesigned this kick to get more angular speed by rocking the Aibo back on its hind legs with its head in the air before kicking. This added enthusiasm resulted in a kick that moved the ball to the left and right at 60 degrees, with a good deal more force than the previous version.

Lunge kicks. We also implemented a variety of lunge kicks, which involved the Aibo lunging forward with one or both arms. These kicks were designed to push the ball forward through any obstructions (like opposing players) and were chiefly used when in scrums. One benefit of these kicks was that they were not very sensitive to the initial position of the ball, and could therefore be used without first executing a controlling move, like a chin pinch.

7 Localization

At the high level, our approach to localization was very similar to last year's approach. We continue to use particle filtering (Monte Carlo Localization) as the basic localization algorithm. However, new challenges due to rule changes and our general desire to improve accuracy, motivated us to develop several enhancements to our approach. The text in this section relies heavily on an understanding of the basic particle filtering approach as described in our 2003 report [12].

7.1 Distance-based Likelihood Updates

Using the distances to landmarks effectively in localization requires that we first account for the non-linear bias in the estimates of landmark distances. The distance computation is performed initially as described in Section 3.5.1.

Using this analytic approach, we found that there was lag and noise in sensor measurements. Also, the farther the robot was from the landmark, the larger the effect of minor defects in visual recognition. As a result, the distances were consistently underestimated. The bias was not constant, and as the distances to the landmarks increased, the error increased to as much as 20%, rendering distance estimates actually harmful to localization.

To overcome this drawback, we introduced an intermediate training phase of *function approximation*. We collected data corresponding to the measured (by the robot) and *actual* (using a tape measure) distances to landmarks at different positions on the field. Using polynomial regression, we estimated the coefficients of a cubic function that when given a measured distance estimate, provided a corresponding *corrected* estimate. That is, given measured values X and actual values Y, we estimated the coefficients, a_i, of a polynomial of the form:[9]

$$y_i|_{y_i \in Y} = a_0 + a_1 x_i + a_2 x_i^2 + a_3 x_i^3|_{x_i \in X} \quad (3)$$

Then, during normal operation, this polynomial was used to compute the *corrected* distance to landmarks. Once this correction was applied, the distance estimates proved to be much more reliable, with a maximum error of 5%. At that error rate, they could be incorporated in the localization algorithm, both for the probability updates and for reseeding, which can then be done with observations corresponding to just two landmarks.

7.2 Incorporating Field Lines

This year, the legged league organizing committee made the localization task more difficult by removing the two center beacons from the sides of the field. The intention of this rule change was to encourage the use of field markings for localization, with the long term goal of removing beacons entirely.

Although our approach continued to work with the center beacons removed, we found that our accuracy was unsatisfactory. Also, considering that the robots focus on the ball for a large proportion of the time, they are able to see the field lines and borders much more frequently than they are able to see the other markers. So localizing based on the field borders and field lines is highly beneficial to the robots. But individual edge/line pixels can be an extremely noisy source of information. So, we opted to use the intersection of field lines as the input to localization. Due to the symmetry of the field, the intersections are not distinct, i.e., their identity at any point of time is not exactly known. But they can still be used to improve localization. Section 3.4 describes the process of identifying the lines and line intersections from the camera image. Once we have

[9] This operation can be performed easily using MATLAB [2].

the position of the line intersections relative to the robot, we treat them as landmark observations but with ambiguous identity. Then, during the probability update phase (see our previous technical report [12] for more details on our baseline implementation), each particle updates its probability based on the comparison between the estimated distance to the intersection and the distance to the closest such intersection, calculated based on the particle's current pose estimate. The addition of line intersections as inputs to the localization module helped a lot in making our localization more robust to the sudden changes that occur during a game.

7.3 Better Motion Model

A motion model designed with the assumption that the robot always moves at full speed prevented us from using the baseline implementation for navigating precisely to a specific location, a capability which is desirable in many practical problem domains. To overcome this drawback, we incorporated an effective modification in the behavior of the robot during localization. When navigating towards a point, the robot moved at full speed when it was more than a threshold distance ($300mm$ in our implementation) away from its target location. When it reached closer to the target position, it progressively slowed down to a velocity almost $\frac{1}{10}$ the normal velocity by shortening its strides. The change in stride-length is necessary due to the discrete nature of "steps" taken by legged robots. The threshold distance affects a trade-off between the accuracy achieved and the associated increase in time. We chose the threshold based on limited experimentation, and found that the robot's behavior is not very sensitive to its exact value. Though this is a minor enhancement, a properly calibrated motion model allowing for accurate slow movements led to a considerable decrease in oscillation, which significantly improves the localization accuracy. Also, as a result of the reduction in the oscillation about the target position when using the baseline approach, this enhancement did not cause positioning to take any longer than the baseline approach, but achieved greater accuracy and smoother motion.

7.4 New Error Model

The way that we chose to represent error and uncertainty in localization has changed quite significantly from last year. We decided to overhaul the error model to make the algorithm parameters easier to understand. To illustrate the complexity of the old approach, here is an example trace of error propagation: variances in beacon height for a given landmark were combined using Gaussian merging; the resulting variance was translated into probabilities of reseed values; the probabilities of all particles were then used to calculate a variance in the pose estimate. Switching back and forth between variances and probabilities requires a known probability distribution that must be found empirically. To eliminate this requirement, we moved to a model consisting exclusively of probabilities.

In our current localization implementation, each observation returned by the high-level vision module (Section 3.3) has a corresponding probability, which represents the confidence in the accuracy of the observation. Observations are only sent to localization if they have a probability of at least 0.5. These probabilities were calculated following a filtering procedure similar to that from the previous year (see the appendices in [12]), except that we also added another filter that caused the probabilities to drop off with distance (the drop off was approximated by a cubic polynomial). The final probability was the product of these two filter outputs.

Observation confidence is ignored during likelihood updates. In other words, all observations are treated equally in the particle probability update phase of Monte Carlo Localization. The probabilities are only propagated through landmark histories to the reseeding estimates. We describe how this is done in the following sections.

7.4.1 Landmark Histories

Previously, observations in the landmark histories were stored as normal distributions, with a mean and variance. Now that we moved to using probabilities to represent confidence, observations stored in the histories can no longer be merged using Gaussian merging. Instead, we compute the merged estimates using a weighted average. We calculate the merged observation $\langle d, \theta, p \rangle$, from each of the n observations, $\langle d_i, \theta_i, p_i \rangle$

in the history, as follows:

$$d = \frac{\sum_{i=1}^{n} p_i \cdot d_i}{\sum_{i=1}^{n} p_i} \tag{4}$$

$$\theta = atan2\left(\sum_{i=1}^{n} p_i \cdot \sin(\theta_i), \sum_{i=1}^{n} p_i \cdot \cos(\theta_i)\right) \tag{5}$$

$$p = \frac{1}{n}\sum_{i=1}^{n} p_i \tag{6}$$

Also, observations can no longer be thrown out of the history for having too high of a variance. Instead, they are now discarded if their probability is less than 0.55. This threshold was found with only minimal experimentation.

7.4.2 Two-landmark Reseeding

Given two landmark observations: two distances and two angles, we can use triangulation to compute two pose estimates. Each pose estimate requires both distances and one of the angles. The two distances are used to find the estimated location, and the angle is used to find the orientation. The two estimates are combined into one reseed value by using the average orientation of the two estimates.

Given perfect observations, the two orientation estimates will come out the same. With poor estimates, the estimates will be quite different. Therefore, the difference in orientation estimates informs us about the quality of the reseed estimate. This information is combined with the average observation confidence to compute the confidence of the reseed value:

$$p_{reseed} = \frac{p_1 + p_2}{2} \cdot sim(\theta_1, \theta_2), \tag{7}$$

where p_1 and p_2 are the confidence probabilities of angle observations θ_1 and θ_2, respectively and $sim()$ is the same angle similarity function used for particle probability updates.

7.4.3 Three-landmark Reseeding

We continued using only angle information for three landmark reseeding, but the error representation has changed. A single reseed value is computed from the three observations. The probability is simply the average of the observation probabilities:

$$p_{reseed} = \frac{p_1 + p_2 + p_3}{3} \tag{8}$$

7.4.4 Pose Estimate

Consistent with the rest of our localization implementation, we no longer represent error in the pose estimate using variance. The new confidence value returned by localization for its pose estimate is simply the average probability of all of the particles. Representing our confidence as a single number has allowed us to improve our ability to debug the robots. We now use the red LED on the Aibo's back to display localization confidence. The LED displays a color gradient between bright red for confident and bright white for completely lost. This has allowed us to easily distinguish between localization inaccuracy and behavioral errors during testing.

7.5 Active Localization

In addition to the algorithmic enhancements described above, changes were made to the robot's behavior with the intention of improving localization accuracy. These changes are described in detail in Section 9.5.

7.6 Simulator

Debugging code and fine-tuning parameters are often cumbersome to perform on a physical robot. Particle filtering implementations require many parameters to be tuned. In addition, the robustness of the algorithm often works to mask errors, making them difficult to track down. For these reasons, we constructed a simulator that allows the robot's high-level modules to interact with a simulated environment through abstracted low-level sensors and actuators. The following sections describe this simulator in detail.

7.6.1 Basic Architecture

The simulator runs as a server, awaiting connections from client processes. One thread processes incoming UDP messages from clients. The other thread runs the simulation and sends UDP messages back to clients. The simulations runs at 20 frames per second. The simulation begins after the first client connects.

All new robot clients start in the center of the field, facing the yellow goal. Each frame, the simulator uses the current action command for each client to update the robots' body positions and head angles according to a stochastic motion model. A stochastic sensor model is used to compute noisy observations for each of the robots. The observations are sent to the clients. The simulator displays the current state of the world and the internal state of the robots. The server then waits for update messages from the clients.

The client program is really just a wrapper around our world state code (Section 2.1) that handles all of the communication with the server. In addition, the client code specifies the agent's behavior using a simple mechanism, independent of the task hierarchy (Section 9.1). This mechanism resembles a simple production system, and is capable of representing behaviors like following a figure-8 path around the field while performing a horizontal head scan.

7.6.2 Server Messages

The client and server communicate using plain text message strings. The following message types are sent from the client to the server:

- **init** - Initialize contact with server
- **param_walk** - Specify change in body movement
- **move_head** - Specify change in head angle
- **info** - Give server internal state info to be displayed by GUI

The following message types are sent from the server to the client:

- **connect** - Acknowledge client initialization
- **see** - Send observations for this time step
- **sense** - Send joint feedback for this time step
- **error** - Respond to incorrectly formatted client message

The full grammar is specified in Appendix D.

7.6.3 Sensor Model

The sensor model returns the distance and angle to fixed landmarks and the ball that fall into the view cone of the robot. The simulated robot is given the same view angle as the real Aibo. Currently, head tilt is ignored and assumed to be horizontal to the ground.

Gaussian noise is added to the distances and angles of each observation to model vision errors. The mean (bias) and variance of this noise can be varied by modifying parameters in the simulator. All observations are returned with a fixed confidence (currently 90%).

7.6.4 Motion Model

The robot is moved by the motion model according to the last specified action command. The robot's body is moved according to the velocities specified in the param_walk command. The head pan is moved according to the move_head command. Unlike the param_walk command, move_head specifies an absolute position rather than a velocity. The simulator moves the head at a fixed speed to the specified pan angle then holds it there.

The translational and rotational displacements of the robot are calculated by multiplying the velocities by the duration of a simulator step. Gaussian noise is added to the displacements and angles. The mean (bias) and variance of this noise can be changed by modifying parameters in the simulator. Joint feedback for the head pan angle is returned noise-free.

7.6.5 Graphical Interface

As the simulator is running, the true position of each robot is displayed on a map of the field as a dark blue isosceles triangle with its apex pointing in the direction of its global orientation. A smaller triangle at the front of the robot, shows the robot's neck pan angle. In addition, the robot's pose estimate from localization is displayed as a light blue triangle, and its particles are shown as white dots.

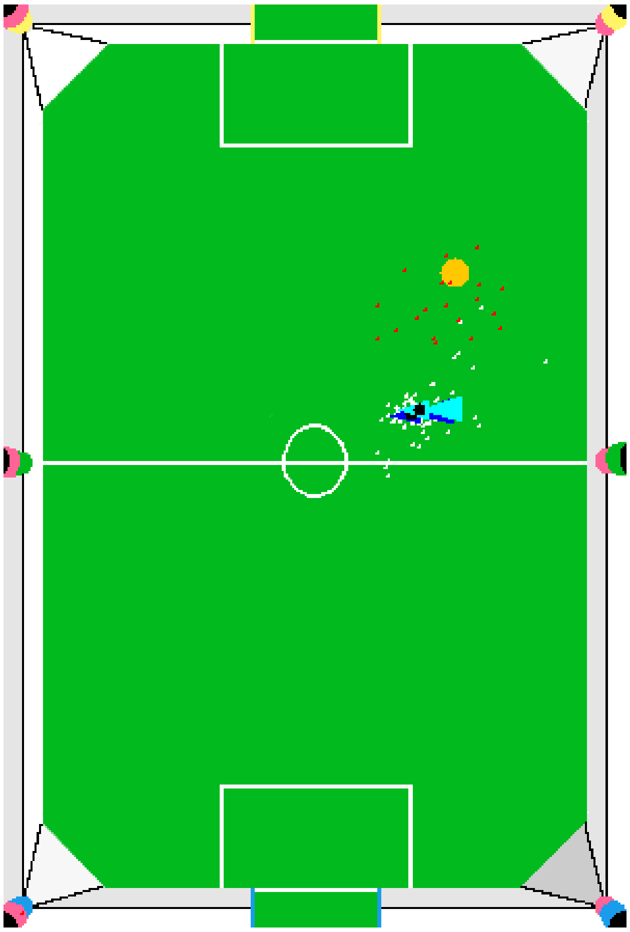

Figure 3: Screen shot of simulator. The robot's true position is in dark blue. Its estimated position is in light blue. The localization particles are shown in white. The ball estimate particles are shown in red.

The user can reposition the robot by clicking on the robot's body with the left mouse button, dragging, then releasing. The orange ball can be moved around in this way as well. The robot's orientation can be changed by clicking on the robot's body with the right mouse button, dragging up and down, then releasing. Additional controls allow the user to pause the simulator and temporarily "blind" the robots by turning off sensory messages.

8 Communication

Collective decision-making is an essential part of a multiagent domain such as robot soccer. Thus, the robots need to share information among themselves. In this section we discuss the methodologies we adopted to enable communication and the various stages of the resulting module's development. The communication module has not undergone any major changes since last year [12]. It has, however, undergone several minor changes as described in this section.

8.1 Knowing Which Robots Are Communicating

As we played more games with our robots, we noticed that sometimes there were communication errors because two robots for some reason or another were not connected to one another. Unfortunately, this kind of error would only be detected when we noticed the robots behaving strangely - something that would not show up until well into a game. Because of the large negative effect of communication failures in our game-play, we decided that we required a way to determine which robots were connected to each other, such that we would have a chance to correct it before starting the game. Additionally, if communication links were to fail during a game, we want our robots to notice this and be able to incorporate this knowledge into their behavior.

Previously, we had a system that allowed us to tell if a robot was "hearing" from any other robot - a blue LED on the top of the head would blink every time a message was received. This was not sufficient - a robot could be hearing from only one other robot, but missing out on the important information from the other two robots. In reality, the system is much more complex than this. With two connections between each robot (each direction is a separate connection), there are a total of 12 connections of which to keep track. Eventually, we decided to light up 4 LEDs on the face (the new ERS-7 robots made this possible), one for each robot. If the robot believes another robot to be "dead", that is, it hasn't heard from it in a while, then that LED is not illuminated. This provides us with a very quick and easy way to establish both whether communication is present (if the lights are illuminated) and what the quality of the communication is (if the lights are blinking in and out).

8.2 Determining When A Teammate Is "Dead"

In order to determine whether or not a robot has heard from another robot lately, each robot keeps track of how many brain cycles ago it last heard from each other robot. If this exceeds a certain quantity, which we called START_PINGING_THRESHOLD (50), the robot will then send the other robot a new message type, called a PING_MESSAGE. Whenever a robot receives a PING_MESSAGE, it is required to immediately respond with a PONG_MESSAGE. If the first robot receives the PONG_MESSAGE, it resets the counter for the last time it heard from the other robot, and things proceed normally. However, if it fails to hear back by a specified interval, CYCLES_BETWEEN_PINGS (25), another PING_MESSAGE will be sent. This will continue until a PONG_MESSAGE is received. If the count of cycles since it last heard from the other robot exceeds the threshold CONSIDER_DEAD_THRESHOLD (100), the other robot is considered "dead" by the first robot. This means that it will act as if the other robot does not exist, making decisions that reflect a 3-, 2-, or 1-robot strategy. If after this point, the first robot receives a PONG_MESSAGE (or any other message), it will restore the other robot's "alive" status. By varying these parameters, we can control how sensitive and responsive the robots are to communication failures.

8.3 Practical Results

This became very useful in the World Cup competition, where communication experienced very high latencies. It allowed our robots to cope reasonably well with the communication problems as well as adjust their strategies accordingly. The main problem we experienced was that one of the other leagues was transmitting on a frequency very close to ours. This caused our communications to be delayed by sometimes up to 10 or 20 seconds. With our original configuration, our robots would not have known what to do — they would have been stuck waiting for instructions from other robots. However, because they could tell that their teammates were *incommunicado*, they were able to act independently.

9 Behaviors

In this section we describe the robots' individual soccer-playing behaviors, broken into three basic tasks: goal-scoring, goal-tending and goal-defending. This year 2003, we were able to spend much more time focusing on this level of the robot code because many of the low-level subtasks were in good shape long before the competition. But first we describe our new framework for writing behavior code, which replaced our previous FSM-based design [12]. The need to implement increasingly sophisticated and high-level behaviors motivated this more scalable approach.

9.1 Task Hierarchy

Previously we specified our soccer-playing agent's behavior with a finite state machine (FSM). Each state corresponded to an activity, such as "Kicking" and "Walking to Seen Ball." These states determined both the low-level commands sent to the movement interface as well as the transition to the next state. This approach proved easy to implement but hard to maintain, refine, and expand. Since the current state unilaterally determined the successor state, developing behaviors as FSMs became as frustrating as writing a program using only goto statements to link together the various segments of code.

We designed the task hierarchy framework to elevate our programming of the agent's behavior to the level of using subroutines. Instead of writing every behavior or activity as an atomic state, we create tasks that may recursively call other tasks. Like a subroutine call, the invocation of a task may persist for some time (throughout multiple Brain cycles) and maintain local state information, to help it decide what subtasks to invoke. We deviate from a pure subroutine-calling paradigm in one substantial way. Classically, when one subroutine calls another, the first subroutine blocks until the second returns. In robotics, we must monitor plan execution so that we can react to external events. Hence, at each Brain cycle, the flow of control does not remain at the top of the stack of active behaviors. Instead, each behavior from the root of the hierarchy to the current leaf will have the opportunity to examine the world state and to terminate its currently active subtask, if necessary. In fact, currently a task cannot terminate itself; the parent task always makes the final decision of when to terminate.

We define two classes of tasks, described below: primitive tasks and composite tasks. For concreteness, we then illustrate a portion of our task hierarchy that includes each kind of task.

9.1.1 Primitive Tasks

Primitive tasks are the leaves of the task hierarchy. When executed, they simply emit commands to send to the Movement Module (see Section 4), without recursively calling any other tasks. Walking in a specific direction, performing a kick, and moving the head in a certain motion are all examples of primitive tasks. For more detail on primitive tasks, see Section 4.2.1.

9.1.2 Composite Tasks

Composite tasks never directly generate commands for the Movement Module. They instead recursively invoke subtasks, which may themselves be composite tasks or primitive tasks. We have two kinds of composite tasks: serial tasks and parallel tasks.

Serial Tasks Serial tasks, which comprise most of the internal nodes of the task hierarchy, perform a sequence of subtasks. When executed, they examine the current WorldState and decide whether to terminate the previously running subtask, if any. If the serial task terminates a subtask or has just been called, it immediately decides on a subtask to call and execute. One example might be a task that searches for the ball if its location is unknown, walks toward the ball if its location is known but distant, and kicks the ball if the ball is known to be right in front of the robot.

Parallel Tasks Parallel tasks allow the robot to invoke multiple tasks simultaneously. Instead of using WorldState information determine what one subtask to execute during a Brain cycle, they blindly execute every subtask during every Brain cycle that they are active. Typically the subtasks control disjoint sets of actuators. For example, one subtask may control the legs while another controls the head. If multiple subtasks do control the same actuator, the last one executed overrides the previous subtasks. This feature allows us to implement high priority behaviors that occasionally override default behaviors.

9.1.3 Example

Figure 4 displays a portion of the robots' task hierarchy for the full soccer task. Each node in this tree represents a task, and connections underneath a given task represent subtasks of that task.

For example, the **xActOnRoleTask** is a major splitting point in the task hierarchy: the duty of this task is to examine the current role of the Aibo and call one of 4 subtasks that will cause the Aibo to act like either an Attacker, a Defender, a Supporter, or a Keeper. The parent of the **xActOnRoleTask** is **xSoccerTask**, which performs administrative game duties like localizing and moving back to a kick-off position after goals. The parent of this task, **xFallSoccerTask**, deals chiefly with getting the Aibo back on its feet after it has fallen over.

The attacker (goal scoring), keeper (goal-tending) and defender (defensive play) tasks make up a large part of this hierarchy, and are described below. The supporter task is described in Section 10.

9.2 Goal Scoring

One of the most important skills for a soccer-playing robot is the ability to score on a goal, even with opponents that are likely to be in the way. In this section we describe the strategy that an individual robot uses when interacting with the ball (i.e., when it is in the "attacker" role). For an algorithmic summary of this strategy, see Figure 7.

Generally, the robot goes to the ball and attempts to grab it using the "chin pinch" maneuver (Section 5.2). However, there are two cases in which we have the robot kick the ball without first bothering to grab it. The first of these cases is when the robot is already aligned in such a way that one of the kicks in its repertoire can be expected to move the ball to the desired position on the field. For example, suppose the robot is on the offensive half of the field and the goal is at a heading of 60° relative to the robot. There is no need for the robot to walk up to the ball, then try to acquire it, then turn it 60° so that it can kick it straight forward with the Fall kick; the robot can simply do the Head kick to the left and achieve the same expected result more quickly and with fewer opportunities for something to go wrong in the process.

The second case is when the robot sees nearby opponents. In this case, it is more important that we move the ball in the right general direction *quickly* than that we align the ball perfectly before kicking it (since we are unable to locate opponent robots very precisely anyway). Thus the robot will choose to execute a version of either the Head kick or the Lunge kick based on its estimation of the relative positions of the opponent robot(s) and the opponent goal.

Except in these two cases, the Aibo always attempts to acquire the ball under its chin and turn it toward a certain angle (using the "chin pinch turn") before kicking. Figure 5 shows a summary of the robot's ball-manipulation behavior on the various parts of the field.

In the defensive half of the field, the robot attempts to kick the ball toward the same-side offensive corner of the field (for example, if the robot is on the left half of the defensive half, it kicks toward the left offensive corner). It always turns away from its own goal, and it chooses whichever kick allows it to turn the least. For instance, if the robot is 120° away from facing the corner it wishes to kick toward, it first turns 60° and then does a Head kick (rather than turning the entire 120° and then doing the Fall kick).

In the offensive half of the field, the robot attempts to kick the ball toward the offensive goal. In the part of the field which is in the offensive half but not in the quarter of the field closest to the offensive goal, the robot chooses whichever kick allows it to turn the least, just as described above for the defensive half of

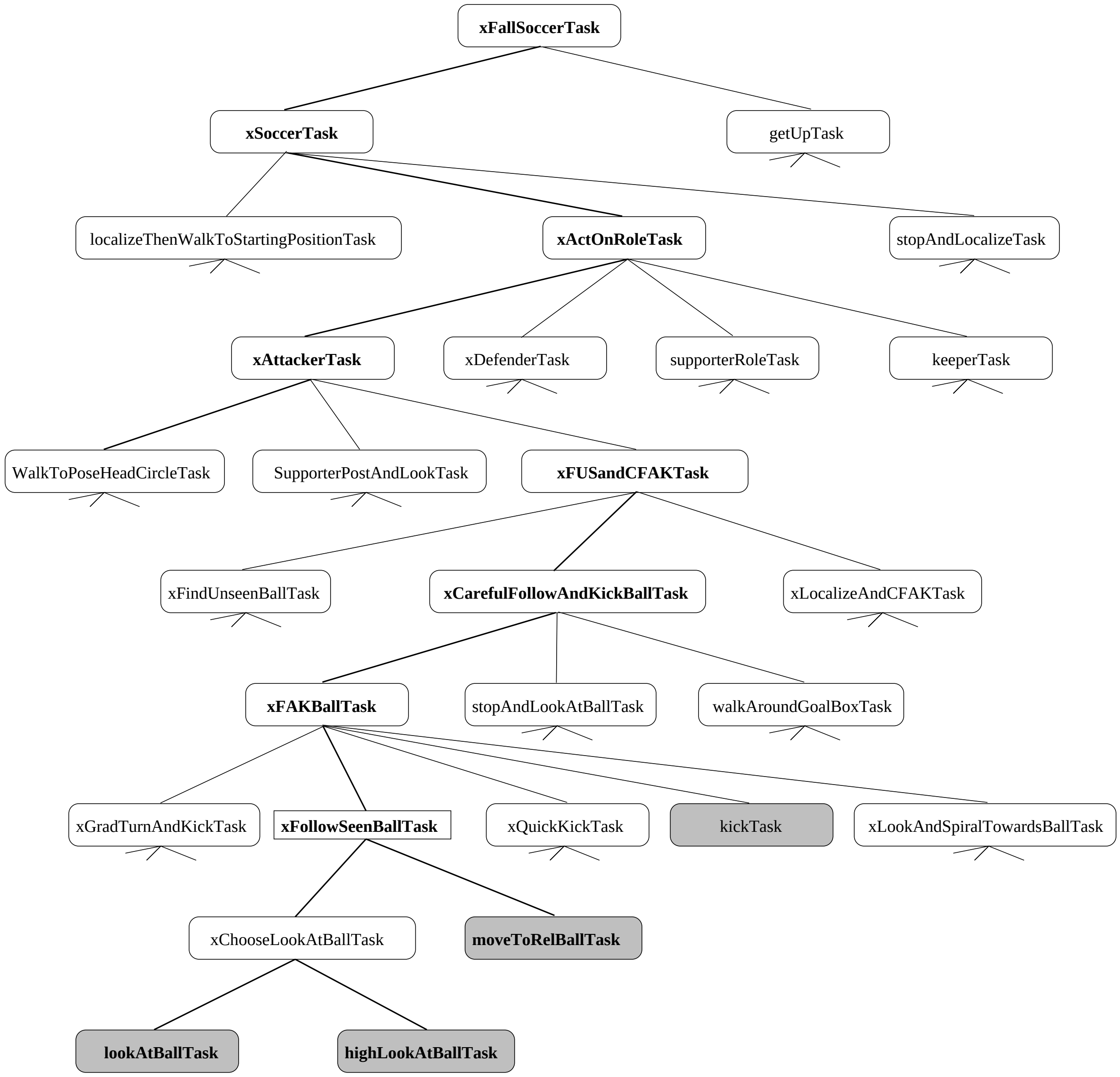

Figure 4: A portion of the task hierarchy, displayed in tree form. All tasks are serial tasks, except the shaded tasks, which are primitive tasks, and the thin rectangular tasks, which are parallel tasks.

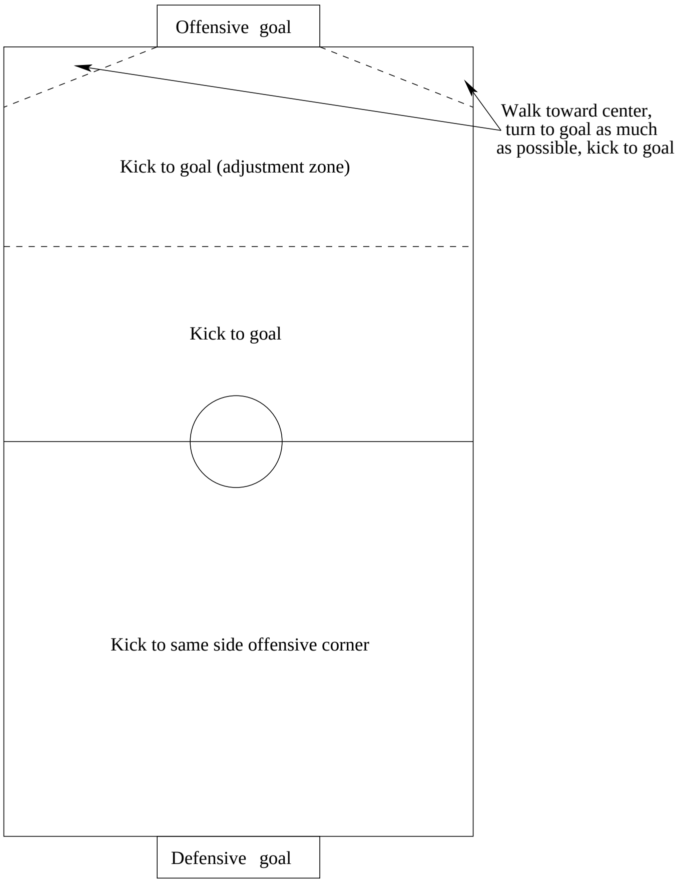

Figure 5: A summary of the kicking strategy.

the field. However, in the quarter of the field closest to the offensive goal,[10] the robot always turns so that it is facing the goal before kicking. This is so it may adjust based on what it sees at the end of its turn, which allows it to correct for small localization or odometry errors and possibly kick around the goalie. Our mechanism for adjustment is quite simple: if the robot sees any goal-colored patch during the last part of the turn, it does not immediately kick when finished turning, but rather uses a slower turn to adjust itself so it is facing the center of the largest patch of goal-color in the last frame when goal-color was seen.

The two offensive-half corners of the field, as delineated in Figure 5, constitute the "offensive wasteland," so called because the angle between the two offensive goalposts is so small that it is very unlikely that a direct kick on the goal will succeed. Therefore we have adopted the following behavior in this area: the robot first attempts to acquire the ball, next (while continuing to hold the ball) it walks toward the center of the field for 1.5 seconds, then it turns toward the goal until either it is facing the goal or 1.5 seconds have passed,[11] and finally it kicks the ball with whatever kick it estimates will shoot the ball most directly at the goal.

As described in Section 4.1.3, collisions during the chin pinch turn can be detected. It is particularly important to be able to detect collisions while turning with the ball, because if the robot's motion is inhibited, it may turn much less than it believes it is turning, and this can result in the robot kicking the ball in entirely the wrong direction. Therefore, if the robot is turning with the ball and it detects a collision, it immediately stops trying to turn. It then kicks the ball with the kick most likely to get the ball to the opponent goal assuming that the robot is still facing the direction it was facing when it *started* the chin pinch turn (because it is likely, if the robot is colliding, that it is has not turned very much if at all).

[10]That is, with the exception of the "offensive wasteland" described subsequently.

[11]This time threshold is intended to keep the robot from violating the 3-second holding rule.

At any time when the robot is approaching the ball and it believes the ball to be on the wall, it "spirals" out around the ball somewhat to avoid approaching parallel to the wall, which has a low chance of successfully capturing the ball (See Figure 6). The complete goal-scoring behavior is summarized in Figure 7.

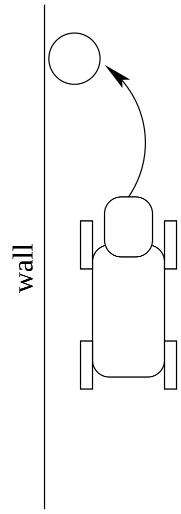

Figure 6: How the Aibo approaches a ball along the wall. At the end of its approach, it spirals its back legs towards the middle of the field to approach the ball more cleanly.

9.3 Goalie

The robot playing in the goalie role needs special-purpose behaviors in order to efficiently carry out its job of defending its goal. Our goalie behavior consists of three main components:

1. A procedure for finding the ball when the robot does not see it.
2. Dynamic positioning that causes the goalie to stand in a base location and orientation that varies depending on the location of the ball.
3. A mechanism that computes the velocity of the ball and uses that information to decide when to initiate a save motion.

The complete goalie behavior is summarized in Figure 8.

9.3.1 Finding the Ball

The goalie's highest priority is knowing where the ball is. If it can see the ball, and the ball is within a certain range it tries its best to keep its eye on the ball at all times. If the ball is sufficiently far away, then it is worth occasionally looking away from the ball to help localization. This trade-off is discussed in more detail in Section 9.5. If the robot has no idea where the ball is, it stands in the middle of the goal box and executes a head scan to look for the ball (and localize). The one exception to this is if the robot has seen the ball recently close by and just lost it. Then the robot turns its body once to each side while scanning its head to look for the ball.

9.3.2 Dynamic Positioning

If the goalie sees the ball, it tries to stand in a location that is as effective as possible for protecting the goal. To determine this location, first we compute the angle of the ball from the center of the goal line. The goalie's location and orientation are functions of this angle. The goalie's attempted orientation is in the direction of the ball, except with a maximum of 45 degrees away from facing straight ahead. For positioning,

```
if (far-from-ball) then
  if (ball-on-wall AND robot-along-wall) then
    SPIRAL-AROUND-BALL
  else
    GO-TO-BALL
  end if
else
  // the robot is at the ball
  SELECT-KICK-TARGET
  if (in-offensive-wasteland) then
    PINCH-BALL-UNDER-CHIN
    WALK-TOWARD-CENTER-OF-FIELD
    TURN-WITH-BALL-TO-ALIGN-FOR-KICK
  else
    if (ball-not-aligned-for-kick-to-target AND no-opponents-nearby) then
      PINCH-BALL-UNDER-CHIN
      TURN-WITH-BALL-TO-ALIGN-FOR-KICK
    end if
  end if
  KICK-TO-TARGET
end if
```

Figure 7: High-level pseudo-code for the robot's goal-scoring behavior. The semantics of the various routines are described in the text.

if the ball is all the way to the right, the goalie attempts to stand right on the goal line, as far right as possible while staying on the goal line. The goalie stands in the symmetrical position for the ball on the left. If the ball is in the middle of the field, the goalie stands in the middle horizontally, but about 10 centimeters further forward. For intermediate values of the ball angle, the robot interpolates between these positions. This corresponds to the DYNAMIC-POSITIONING routine in Figure 8

9.3.3 Velocity Blocking

While the goalie is watching the ball, it continually maintains an estimate of the ball's velocity. It uses this information to sidestep towards where the ball is coming, and to know when to initiate a save motion. The save motion is implemented via our kicking machinery (as described in Section 6). The save kick extends both front legs to the side, causing the robot to fall on its chest. The robot then pushes its legs forwards, often kicking the ball away, and then it moves back to the standing position.

To compute the ball's velocity, the robot maintains a queue of consecutive ball locations (relative to the robot). Every vision frame in which the ball is seen, the ball location and time-stamp are enqueued. If a ball location is more than a constant number of frames (Q_{max}) old, it is dequeued. Finally, in any frame where the ball is not seen, the queue is cleared. If the queue is longer than a minimum length (Q_{min}), then we can use it to compute a velocity for the ball. To do this, we split the queue in half and compare the center of mass of each half of the ball locations. The direction from the first half to the second half is taken as the ball's direction, and an amount of time is computed by subtracting the average time-stamp in the first half from that of the second half. This displacement vector and time yield a velocity vector. The constants Q_{min} and Q_{max} were tuned manually, and we use values of 2 and 6 respectively (with one frame coming in roughly 25 times a second). This is the procedure denoted as COMPUTE-BALL-VELOCITY in Figure 8.

The goalie's velocity-blocking mechanism is egocentric. That is, it tries to prevent the ball from going past the line through the robot perpendicular to the direction the robot is facing, independent of localization information. Thus the location information is stored in the goalie's frame of reference. To do this, the entire queue is translated and rotated as needed every cycle in accordance with the robot's odometry. The ball's

```
if (cannot-see-ball) then
   if (ball-seen-close-recently) then
      TURN-LEFT-AND-RIGHT
      SCAN-HEAD
   else
      STAND-IN-MIDDLE-OF-BOX
      SCAN-HEAD
   end if
else
   // the goalie sees the ball
   COMPUTE-BALL-VELOCITY
   if (ball-approaching-goal-and-in-saving-distance) then
      EXECUTE-SAVE-MOTION
   else
      DYNAMIC-POSITIONING
      LOOK-AT-BALL
   end if
end if
```

Figure 8: High-level pseudo-code for the robot's goal-tending behavior. The semantics of the various routines are described in the text.

location and velocity are extended to determine where and when the ball will cross the line of the goalie. These values determine whether the goalie steps to the right, or to the left, or executes a save motion.

9.4 Defensive Play

Here, we describe the set of actions that a robot performs when functioning as a defender. The behavior consists of a set of states that the defender transitions between based on certain events such as the position of the robot, relative and global position of the ball, positions of other teammates and opponents etc. The task is designed such that the robot is able to perform its primary duty: prevent any member of the opponent team from scoring on its goal. After we restructured our behavior architecture to make it more modular, certain behaviors that were common to different roles were defined separately and "reused" across several roles.

9.4.1 Defensive Post

The defender has a defensive post that is defined based on certain environmental factors, primarily the position of the ball. When the ball is in the opponent's half of the field, the robot positions itself at a point near the field center such that it suitably positioned to tackle a shot by one of the opponents. Our defensive task is conservative in that if the ball is in the top third of the field, the robot functioning as the defender does not go to it (if only one robot is on the field, it automatically becomes an attacker and then attacks the ball). At the same time, the defender also adjusts its defensive post based on whether the ball is in the left or right half of the field. This entire action corresponds to the predicate SELECT-DEFENSIVE-POST in the pseudo-code (Figure 9).

The behavior can be broadly split into two categories based on whether it can or cannot go to the ball. This is based on the command received from the commander (for more details on the commander and message passing between team members see Section 10). We shall describe both these situations below.

9.4.2 Defender cannot go to the ball

This occurs when the defender has been told explicitly by the commander not to approach the ball. This in general implies that some other robot from the same team is approaching the ball and it is therefore better for the other team members to position themselves at strategic locations. In such a situation the defender positions itself based on its estimate of the ball location (seen or known position of the ball). The seen position of the ball refers to the robot's estimate of the position of the ball when the robot actually sees the ball. The known position of the ball refers to the robot's estimate of the ball position when the robot does not actually see the ball but either it has seen the ball in the recent past and/or one of its team mates is seeing the ball and communicating the ball position to it. For a description of the ball knowledge and representation see Section 2.1. If the robot has a good enough estimate of the ball, it uses that to determine its defensive position (as described in Section 9.4.1 above) and attempts to walk backwards to this position while facing the ball (predicate WALK-TO-DEFENSIVE-POST-KEEPING-BALL-IN-VIEW). On the other hand, in the absence of the knowledge of the ball location (low confidence in ball location), it turns by the required angle (target orientation predefined based on team color; Blue = 0^o, Red = 180^o) and walks forward to the default defensive post — routine TURN-AND-WALK-TO-DEFENSIVE-POST-AT-FULL-SPEED in the pseudo-code given below. By turning to face the target point, it reaches the position faster, since our forward walk is significantly faster than our omni-directional walk. The assumption here is that in this situation none of the teammates has a good idea of the ball because if one of the teammate has a good idea of ball location it does get communicated to all the other teammates. The best a defender can then do is to try and go back to a post near its own goal box so that it can prevent a goal if one of the opponents has control of the ball. Once the robot has reached its position, its action is once more decided based on the ball location — remember that all this occurs only when the robot is specifically told not to approach the ball. If the robot has a good enough estimate of the ball location, it assumes an orientation so as to keep the ball in view, i.e. it tracks the ball movement on the field — routine TRACK-BALL-IN-PLACE. If not, the robot starts executing a search routine that turns in place to try and find the ball — routine SEARCH-IN-PLACE-FOR-BALL. To ensure that the robot (defender) does not get stuck in this state forever, it periodically tries to ensure that it is in its home position and if it finds that it has drifted, it corrects its position first before going back to its search routine. Further, since the estimate of the ball degrades rapidly unless the robot keeps seeing the ball or keeps receiving good estimates of the ball position from teammates, the robot does not get stuck in the state of staring in a particular direction.

9.4.3 Defender can go to the ball

When the defender can go to the ball (predicate *defender-can-go-to-ball* in Figure 9), it implies, in most cases, that it is the robot in its team that is the closest to the ball and hence has control over the ball (decided by a process similar to that from the previous year: see [12]). In this case its behavior is not much different from a robot functioning as an attacker (see Section 9.2). The overall behavior of the robot when working as a defender can be summarized as shown in Figure 9 below.

In either case, whether the robot can or cannot go to the ball, there is a check to see if the robot's intended path crosses its penalty zone and if so it attempts to walk around the box. We describe this procedure in the next section.

9.4.4 Avoiding the penalty zone

According to the rules of the game, any member of the team other than the keeper is not allowed to enter the penalty zone, the boxed region around its own goal. Figure 10 shows a pictorial representation of a possible situation where the direct path to the ball takes the robot through its own goal box.

This problem can be solved by imposing a strict constraint that just prevents the robot from walking into the box. But at the same time, we do not want the robot to not go to the ball just because the direct path to the ball goes through the penalty box; In this case, we want the robot to walk around the box until it has a direct path to the ball that does not move it into the box. We solved this problem by considering the following two cases:

```
if (defender-can-go-to-ball) then
  ATTACK
else
  //defender cannot go to ball
  SELECT-DEFENSIVE-POST
  if (not-at-defensive-post) then
    if (ball-seen) OR (ball-known) then
      WALK-TO-DEFENSIVE-POST-KEEPING-BALL-IN-VIEW
    else
      TURN-AND-WALK-TO-DEFENSIVE-POST-AT-FULL-SPEED
    end if
  else
    //defender already at the defensive post
    if (ball-seen) OR (ball-known) then
      TRACK-BALL-IN-PLACE
    else
      SEARCH-IN-PLACE-FOR-BALL
    end if
  end if
end if
```

Figure 9: High-level pseudo-code for the robot's defensive behavior. The semantics of the various routines are described in the text.

Figure 10: A situation where the robot is forced to walk around its goal box.

1. If the ball is within the goal box then any path that the robot takes towards the ball has to pass through the goal box. In this case, the robot does not go to the ball. It lets the keeper take care of clearing the ball in this situation and just positions itself outside the box while keeping the ball in view so that as soon as the ball leave the goal box, it can go to it.

2. If the ball is not within the goal box but the direct path to the ball takes the robot through the goal box, i.e., the straight line from the robot to the ball intersects the goal box lines at more than one location, the robot *plans* a path around the box. It basically attempts to get to the closest corner of the box (in its path to the ball) while keeping the ball in view and repeatedly performs this task until the line joining the robot and the ball does not intersect the goal box lines. It then goes to the ball directly.

This procedure helped us prevent occurrences of the *illegal defender* penalty and having all the robots on the field for most part of the game helps execute better strategies.

9.5 Active Localization

One important behavior that we wanted to be able to implement for 2004 was active localization, or the ability for the robot to execute a movement specifically designed to gather more localization information. Other teams have implemented active localization in fairly sophisticated ways, such as Kwok and Fox's use of reinforcement learning to decide when to look at landmarks instead of the ball [7]. We decided to start with a relatively simple active localization scheme, in which the Aibo periodically performed a head scan while moving towards the ball. It was important not to actively localize too frequently, or to localize when too close to the ball, lest the Aibo lose track of the ball.

Our initial soccer-playing behaviors were not designed with active localization in mind, since most of these behaviors focused on moving towards the ball when the ball was in view. Simply adding an active localization behavior to this scheme would have resulted in the Aibo losing the ball whenever it attempted to localize, which would have then caused the Aibo to go into a "find-the-ball" behavior. This discontinuity of behaviors would have severely impeded the ability of the Aibo to move quickly to the ball. In order to implement an effective active localization behavior that did not suffer from this problem, we found it necessary to create a set of behaviors that could move the Aibo towards an abstract representation of the ball without constantly seeing the ball. Often these new behaviors resembled the behaviors that they replaced, but reasoned instead about higher-level abstract objects instead of low-level data. For example, while we still had a behavior that was responsible for moving the robot towards the ball, the new behavior moved the robot towards a ball position defined by the ball particles (see Section 2.1) rather than a ball seen through the camera.

10 Coordination

Our approach to multi-robot coordination is essentially a dynamic role-based scheme. In this section, we first identify the full set of roles filled by the robots. Then we address how they are dynamically assigned such that each role is filled by one robot at any given time.

10.1 Roles

Our strategy uses a dynamic system of roles to coordinate the robots. In this system, each robot has one of three roles: *attacker*, *supporter*, and *defender* (the *goalkeeper* role is assigned to a single robot permanently). The roles are assigned based on the number of robots available (decided based on the number of robots that can communicate with the *commander* - see section 10.1.4) and their positions relative to each other and to the ball. Before we describe this process, let us take a look at some of the individual behaviors.

10.1.1 Attacker Behavior

The attacker's behavior is to try to score a goal, as discussed in Section 9.2.

10.1.2 Supporter Behavior

The supporter's behavior and the interplay between the attacker and supporter have not changed much from last year's team [12]. The switching decisions and supporter behavior have been rewritten in terms of the new world state and task hierarchy, but the functionality is effectively the same.

10.1.3 Defender Behavior

The defender's behavior is to prevent the opposition from scoring a goal. It is described in detail in Section 9.4.

10.1.4 Dynamic Role Assignment

Because of the continuously changing field, coordination amongst the four robots is very challenging. Originally, we wanted each robot to decide exclusively for itself what to do. Unfortunately, because each robot has its own frame of reference, latest information, and possible communication outages, this can't always be done. What we needed was a way to ensure that at any time we had the right number of players carrying out any given task.

In order to achieve this, we decided that the robots needed to agree on their role assignments. This left us with two options:

1. Solving a distributed consensus problem
2. Allowing one robot to make the role assignments

While the first is an interesting distributed systems problem, the second was much more feasible. We wanted to minimize overhead, and with only four robots, we did not really need anything fancy.

We decided that one robot would be the *commander*. The commander makes the decisions and informs each robot as to which role it should adopt. The only hard part is finding a way to ensure that exactly one commander exists at any time. Were we to solve this problem exactly, it would be the same as solving a distributed consensus problem. However, this problem is much easier to approximate than a distributed consensus problem.

In the face of frequent communications failures (as we often experienced in competition) and crashed robots, meeting this last goal turned out to be impossible. The robots cannot tell the difference between a robot they haven't heard from in a while and a robot that is off, crashed, or out of communication range. In fact, for all intents and purposes, they *should* see these as the same situation. Thus, we compromised on a weaker goal: amongst any clique of connected robots, there is at most one commander. Furthermore, if any two robots are connected to the same set of robots, they will listen to the same robot.

The trick to getting this to work was to break the symmetry among the robots. Because TCPGateway (the module provided by the league organizers for communication) imposes an ordering on the robots, we used this ordering to determine the commander. By using messages similar to the PING_MESSAGEs described in Section 8, a commander could *assert authority* over other robots by sending a COMMAND_PING_MESSAGE. If it hasn't issued a command to a robot in a while, it sends one of these messages just to remind the other robot who is boss. If a robot receives messages or COMMAND_PING_MESSAGEs from more than one robot, it ignores the messages from the robot with the higher ID (as determined by TCPGateway). For our normal setup, this means that the goalie (robot 1) is typically commander. This assignment was intentional: we did not want an attacker that was busy doing critical vision processing to spend its time calculating roles for the other players. In practice, our goalie had the least amount of contact with the ball - usually the ball was out in the other parts of the field.

This technique worked very well and was quite resilient to failure: when a robot crashes or is turned off, a new commander is selected almost immediately. When the old robot is returned to play, it takes over gracefully.

In the end, however, we found that it was beneficial to make the attacker the default commander. Instead of having the robot with the lowest ID the commander, we reversed all the inequalities and made the robot with the highest ID the commander. During the RoboCup 2004 competition, we experienced extremely high latencies in our network. Meanwhile, the most frequent role switching occurred between attacker and the supporter. Having the goalie as commander resulted in the in the following sequence of events happening frequently:

1. The attacker and supporter acquire new information with regard to their whereabouts relative to the ball.
2. The attacker and supporter send this information to the goalie (with latency).
3. The goalie decides the two should switch roles.
4. The goalie sends these new role assignments to the attacker and supporter (with latency).

5. The attacker and supporter switch roles.

This sequence was happening so frequently, with such high latency, that the extra processing time on the attacker was worth cutting out one leg of the trip. Instead, the attacker could make the decision on its own, begin to act in the appropriate way, and meanwhile send the message to the supporter. In practice, we noticed a significant increase in responsiveness next to which any deterioration in performance of the attacker was unnoticeable.

11 UT Assist

During the course of our development, we created a valuable tool to help us debug our robot behaviors and modules. This tool, which we called UT Assist, allowed us to experience the world from the perspective of our Aibos and monitor their internal states in real-time.

The basic structure of UT Assist did not change from the original implementation in 2003 [12]. The flexible structure of UT Assist did allow us to add new visualizations as they became useful, however. For example, when the vision module started to successfully process lines, a data type was added to UT Assist so that the Aibos could transmit the lines that they saw. When we started modeling the position of the ball with a particle filter, this flexible framework also allowed us to transmit and display the ball particles.

12 The Competitions

In the RoboCup initiative, periodic competitions create fixed deadlines that serve as important motivators. In 2004, we entered the Second U.S. Open competition as well as the Eighth International RoboCup Competition. This section describes our results and experiences at those events.

12.1 American Open

The Second U.S. Open RoboCup Competition was held in New Orleans, LA from April 24th to 27th, 2004.[12] Eight teams competed in the four-legged league, and were divided into two groups of four for a round robin competition to determine the top two teams which would advance to the semi-finals. The three other teams in our group were from Georgia Institute of Technology, Dortmund University, and Instituto Tecnologico Autonoma de Mexico (ITAM), from Mexico. After finishing in second place in the group, we advanced to the semifinals against a team from the University of Pennsylvania, and eventually the tournaments third place game against Dortmund. The results of our three games are shown in Table 1. Links to videos from these games are available at `http://www.cs.utexas.edu/~AustinVilla/?p=competitions/US_open_2004`.

Opponent	**Score (us-them)**	**Notes**
ITAM	8–0	
Dortmund	2–4	
Georgia Tech	7–0	
Penn	2–3	Semi-final
Dortmund	4–3	

Table 1: The scores of our 5 games at the U.S. Open.

Our first game against ITAM earned us our first official win in any RoboCup competition. The score was 3–0 after the first half and 8–0 in the end. The attacker's adjustment mechanism designed to shoot around the goalie (Section 9.2) was directly responsible for at least one of the goals (and several in later games).

Both ITAM and Georgia Tech were still using the smaller and slower ERS-210 robots, which put them at a considerable disadvantage. Dortmund, like us, was using the ERS-7 robots. Although leading the team

[12] `http://www.cs.uno.edu/~usopen04/`

from Dortmund 2–1 at halftime, we ended up losing 4–2. But by beating Georgia Tech, we still finished 2nd in the group and advanced to the semi-finals against Penn, who won the other group.

Our main weakness in the game against Dortmund was that our goalie often lost track of the ball when it was nearby. We focused our energy on improving the goalie, eventually converging on the behavior described in Section 9.3, which worked considerably better.

Nonetheless, the changes weren't quite enough to beat a good Penn team. Again we were winning in the first half and it was tied 2–2 at halftime, but Penn managed to score the only goal in the second half to advance to the finals.

Happily, our new and improved goalie made a difference in the third place game where we won the rematch against Dortmund by a score of 4–3. Thus, we won the 3rd place trophy at the competition!

We came away from the competition looking forward towards the RoboCup 2004 competition two months later. Our main priorities for improvement were related to improving localization including the ability to actively localize, adding more powerful and varied kicks, and more sophisticated coordination schemes.

12.2 RoboCup 2004

The Eighth International RoboCup Competition was held in Lisbon, Portugal from June 28th to July 5th, 2004.[13] 24 teams competed in the four-legged league and were divided into four groups of six for a round robin competition to determine the top two teams which would advance to the quarter-finals. The teams in our group were the ARAIBO from The University of Tokyo and Chuo University in Japan; UChile from the Universidad de Chile; Les 3 Mousquetaires from the Versailles Robotics Lab; and Penn. Wright Eagle from USTC in China was also scheduled to be in the group, but was unable to attend. After finishing 2nd in our group, we qualified for a quarter-final match-up against the NuBots the University of Newcastle in Australia. The results of our 5 games are shown in Table 2. Links to videos from these games are available at `http://www.cs.utexas.edu/~AustinVilla/?p=competitions/roboCup_2004`.

Opponent	**Score (us-them)**	**Notes**
Les 3 Mousquetaires	10–0	
ARAIBO	6–0	
Penn	3–3	
Chile	10–0	
NuBots	5–6	Quarter-final

Table 2: The scores of our five games at RoboCup.

In this pool, Les 3 Mousquetaires and Chile were both using ERS-210 robots, while the other teams were all using ERS-7s. In many of the first round games, the communication among robots and the game controller was not working very well (for all teams), and thus reduced performance on all sides. In particular, in the game against Penn, the robots had to be started manually and were unable to reliably switch roles. In that game, we were winning 2–0, but again saw Penn come back, this time to tie the game 3–3.

This result left us in a tie with Penn for first place in the group going into the final game. Since the tiebreaker was goal difference, we needed to beat Chile by 2 goals more than Penn beat ARAIBO in the respective last games in order to be the group's top seed. Penn proceeded to beat ARAIBO 9–0, leaving us in the unfortunate position of needing to score 11 goals against Chile to tie Penn, or 12 to pass them. The 10–0 victory left us in second place and playing the top seed of another group in the quarter-finals, the NuBots.

In the quarter-final, the network was working fine, so we got to see our robots at full speed. We scored first twice to go up 2–0. But by halftime, we were down 4–2. In the 2nd half, we came back to tie 4–4, then went down 5–4, then tied again. With 2 minutes left the NuBots scored again to make it 6–5, which is how it ended. It was an exciting match, and demonstrated that our team was competitive with some of the best

[13] `http://www.robocup2004.pt/`

teams in the competition (Penn and the NuBots both lost in the semi-finals, though). In the end, we were quite pleased with our team's performance.

13 Conclusions and Future Work

The experiences and algorithms reported in this technical report represent the coming of age of the UT Austin Villa legged-league robot team. Thanks to a concerted effort over the course of the few months leading up to the competition, we were able to port our 2003 code from the ERS-210A robots to the ERS-7s, and in the process greatly improve our robots' capabilities along many dimensions.

There are still many directions for future improvements to our team, as noted throughout this report. We plan to continue our development toward future RoboCup competitions. But more importantly, we continue to make use of this code base as a fully functional research platform and are using it for investigations in various directions including learned walking [6] and ball manipulation [3]; robot joint modeling [14]; color constancy in vision [11]; and automatic sensor and action model calibration [13]. Further details on these research projects, including links to several illustrative videos, as well as reports on our competitions and links to most of the games described in this report, are all available from our team webpage at `http://www.cs.utexas.edu/~AustinVilla`.

Overall, developing a competitive RoboCup soccer team has been a rewarding learning experience. We look forward to building from it in the future and continuing to contribute to the RoboCup initiative.

Acknowledgments

Thanks to Selim T. Erdoğan and Ellie Lin for their conceptual contributions to the team development process, and to Laurel Issen for her editing suggestions. The authors would also like to thank Sony for developing the robots and sponsoring the four-legged league, as well as the previous RoboCup legged-league teams for forging the way and providing their source code and technical reports as documentation. This research is supported in part by NSF CAREER award IIS-0237699 and ONR YIP award N00014-04-1-0545.

References

[1] The Matlab Curvefitting Reference. `http://www.mathworks.com/access/helpdesk/help/toolbox/curvefit/`.

[2] The Matlab Reference. `http://www.mathworks.com/`.

[3] Peggy Fidelman and Peter Stone. Learning ball acquisition on a physical robot. In *Proceedings of the Fourth International Symposium on Robotics and Automation*, pages 62–66, August 2004.

[4] Jeff Hyams, Mark W. Powell, and Robin R. Murphy. Cooperative navigation of micro-rovers using color segmentation. *In Journal of Autonomous Robots*, 9(1):7–16, 2000.

[5] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. Robocup: The robot world cup initiative. In *The First International Conference on Autonomous Agents*, pages 340–347, February 1997.

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

[7] C. Kwok and D. Fox. Reinforcement learning for sensing strategies. In *The IEEE International Conference on Intelligent Robots and Systems*, 2004.

[8] B. W. Minten, R. R. Murphy, J. Hyams, and M. Micire. Low-order-complexity vision-based docking. *IEEE Transactions on Robotics and Automation*, 17(6):922–930, 2001.

[9] Michael J. Quinlan, Craig L. Murch, Richard H. Middleton, and Stephan K. Chalup. Traction monitoring for collision detection with legged robots. In *RoboCup-2003: Robot Soccer World Cup VII.* Springer, Berlin, 2004.

[10] Robert J. Schilling. *Fundamentals of Robotics: Analysis and Control.* Prentice Hall, 2000.

[11] Mohan Sridharan and Peter Stone. Towards illumination invariance in the legged league. In Daniele Nardi, Martin Riedmiller, and Claude Sammut, editors, *RoboCup-2004: Robot Soccer World Cup VIII.* Springer Verlag, Berlin, 2005. To appear.

[12] Peter Stone, Kurt Dresner, Selim T. Erdoğan, Peggy Fidelman, Nicholas K. Jong, Nate Kohl, Gregory Kuhlmann, Ellie Lin, Mohan Sridharan, Daniel Stronger, and Gurushyam Hariharan. UT Austin Villa 2003: A new RoboCup four-legged team, AI Technical Report 03-304. Technical report, Department of Computer Sciences, University of Texas at Austin, October 2003.

[13] Daniel Stronger and Peter Stone. Simultaneous calibration of action and sensor models on a mobile robot. In *2004 International Symposium on Robotics and Automation (ISRA)*, August 2004.

[14] Daniel Stronger and Peter Stone. A model-based approach to robot joint control. In Daniele Nardi, Martin Riedmiller, and Claude Sammut, editors, *RoboCup-2004: Robot Soccer World Cup VIII.* Springer Verlag, Berlin, 2005. To appear.

A Coordinate Transforms

In this section, we document some of the coordinate transforms that we have incorporated in our code. Once the coordinate frames are appropriately set up, homogeneous transformation (DH transformation [10]) matrices provide an easy method to perform coordinate transforms. They are easy to implement and debug on the robots. In the following subsections, we shall elaborate separately on some of the coordinate transformations.

A.1 Camera Transform

In this section we describe the transform that, given a position (x, y, z) in the camera coordinate frame, provides the corresponding position in the frame centered at the base of the robot's neck. We also describe how to convert pixels in the image frame to positions in the camera coordinate frame. This is used extensively in the vision module (see Section 3.5.1 and Section 3.4) while dealing with the marker and line distance estimates.

Refer to Figure 11 to understand the coordinate frames used here. Basically we have aligned the frames so that all rotations work on the right-hand rule: Counterclockwise rotation is positive.

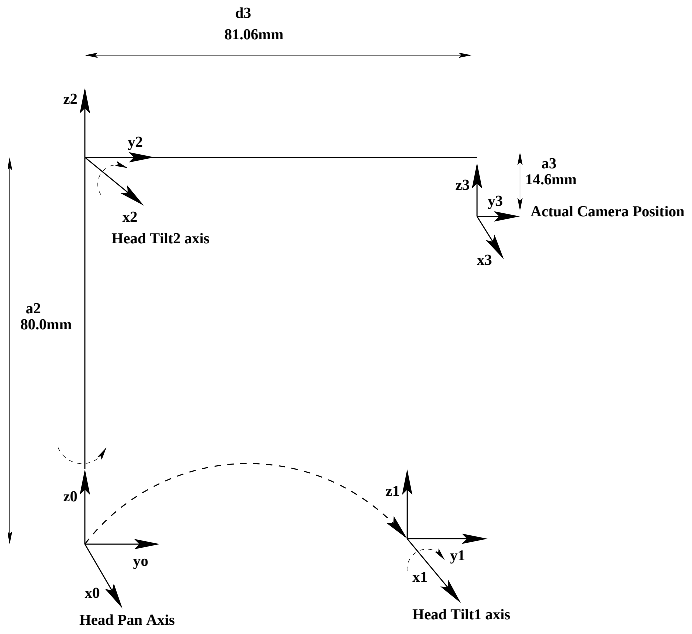

Figure 11: This figure shows the basic Camera coordinate systems.

Now we need to get a transform that moves values in the camera coordinate frame to the base-of-the-neck coordinate frame. To do so involves the following motions in the order mentioned here:

1. Rotation about the fixed frame z-axis (*pan*).
2. Rotation about the fixed frame x-axis (*tilt1*).
3. Translation of ($a_2 = 80mm$) along moving frame z-axis.
4. Rotation about the moving frame x-axis (*tilt2*).
5. Translations about the moving frame y and z axes by ($d_3 = 81.06mm$) and ($a_3 = -14.6mm$) respectively.

Now all we need to remember is that to get the transformation matrix from the final moving frame (here camera origin) to the fixed base frame (here base of neck), we pre-multiply rotations and translations about fixed frame axes while we post-multiply when the motions are about the moving frame. Then, if we know that:

$$Rot(x,\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{9}$$

$$Rot(y,\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{10}$$

$$Rot(z,\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{11}$$

$$Trans(a,b,c) = \begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{12}$$

We can easily write up the chain of matrices that make up the required transform from the camera origin to the base of the neck.

$$T_{neck}^{cam} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c_2 & -s_2 & 0 \\ 0 & s_2 & c_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & a_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c_3 & -s_3 & 0 \\ 0 & s_3 & c_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & d_3 \\ 0 & 0 & 1 & a_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This simplifies to the following matrix:

$$T_{neck}^{cam} = \begin{pmatrix} c_1 & -s_1c_3 & s_1s_3 & -s_1(d_3c_3 - a_3s_3) \\ s_1c_2 & c_1c_2c_3 - s_2s_3 & -c_1c_2s_3 - s_2c_3 & c_1c_2(d_3c_3 - a_3s_3) - s_2(a_2 + d_3s_3 + a_3c_3) \\ s_1s_2 & c_1s_2c_3 + c_2s_3 & -c_1s_2s_3 + c_2c_3 & c_1s_2(d_3c_3 - a_3s_3) + c_2(a_2 + d_3s_3 + a_3c_3) \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{13}$$

This is what we use for transforming a vector in the camera origin coordinate frame to the coordinate frame at the base of the neck. For example, given a vector $(x_{cam}, y_{cam}, z_{cam})^T$ in the camera coordinate frame, to get the value of the vector with respect to the coordinate frame at the base of the neck, we do:

$$\begin{pmatrix} x_{neck} \\ y_{neck} \\ z_{neck} \\ 1 \end{pmatrix} = T_{neck}^{cam} \begin{pmatrix} x_{cam} \\ y_{cam} \\ z_{cam} \\ 1 \end{pmatrix} \tag{14}$$

Not only does the transformation matrix give the position in the coordinate frame centered at the base of the robot's neck, it also provides unit vectors that represent the moving frame's axes with respect to the fixed frame's axes.

With reference to the camera, we also need to transform from the pixel/image coordinates (along the image) to the actual camera coordinate frame. This is performed by the following set of equations (based on the standard image coordinate frame and the camera coordinate frame system described above):

$$\begin{aligned} x_{cam}^{img} &= x_{img} - x_{imgc} \\ y_{cam}^{img} &= \text{focal length} = 3.27mm \\ z_{cam}^{img} &= y_{imgc} - y_{img} \end{aligned} \tag{15}$$

where, $x_{imgc} = 103.5$ and $y_{imgc} = 79.5$ represent the center of the image in pixel coordinates.

Next we shall look at similar transforms for body rotations and also for the back legs (required for finding the height of the robot's body center above the ground).

A.2 Body Transforms

In the case of the body several transforms have to be determined. Some of them are fixed and do not change from frame to frame while others need to be computed every frame of analysis. These transforms are used to determine the height of the center of the robot's body from the ground (used in line pixel projections - Section 3.4). They are also used in conjunction to the camera transformations for determining the distances of the markers with respect to the center of the robot's body (Section 3.5.1). Let us now look at some of these transforms.

A.2.1 Body Tilt and Roll

This matrix is computed for every visual frame. It provides the compensation for body tilt and roll while the robot is in motion. The body tilt and roll are determined from the robot's accelerometers. Once these values have been obtained, we can generate matrices that transform and hence compensate for the body tilt and roll with respect to the plane along the center of the robot's body. Using the suffix *r, t* for the roll and tilt respectively, we get:

$$T_{BodyC}^{TiltRoll} = \begin{pmatrix} c_r & 0 & s_r & 0 \\ 0 & 1 & 0 & 0 \\ -s_r & 0 & c_r & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c_t & -s_t & 0 \\ 0 & s_t & c_t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} c_r & s_r s_t & s_r c_t & 0 \\ 0 & c_t & -s_t & 0 \\ -s_r & c_r s_t & c_r c_t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{16}$$

Then, to obtain the transform from the coordinate frame at the base of the neck to the center-of-the-body coordinate frame (taking into account the body tilt and roll), we *post-multiply* the above matrix with the constant transformation matrix from the neck coordinate origin to the origin at the center of the robot's body:

$$T_{BodyC}^{neck} = T_{BodyC}^{TiltRoll} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & y_c \\ 0 & 0 & 1 & z_c \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{17}$$

where, $y_c = 67.5, z_c = 19.5$ are the y and z axis offsets between the base of the neck and the center of the body. These are constants that do not change from frame to frame.

By a similar process, we can produce the transforms from the rear leg shoulders to the center-of-the-body coordinate frames (refer to Figure 12 for more details on coordinate frames for the left rear leg):

$$T_{BodyC}^{left-should} = T_{BodyC}^{TiltRoll} \begin{pmatrix} 0 & -1 & 0 & x_{dl} \\ 1 & 0 & 0 & y_{dl} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{18}$$

where offsets $x_{dl} = -62.5mm$ and $y_{dl} = 65mm$.

$$T_{BodyC}^{right-should} = T_{BodyC}^{TiltRoll} \begin{pmatrix} -1 & 0 & 0 & x_{dr} \\ 0 & -1 & 0 & y_{dr} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{19}$$

where offsets $x_{dr} = 62.5mm$ and $y_{dr} = -65mm$.

Another easy extension at this stage would be:

$$T_{BodyC}^{cam} = T_{BodyC}^{neck} \cdot T_{neck}^{cam} \tag{20}$$

This provides the final transformation from the camera coordinate frame to the center-of-the-body coordinate frame with compensation for the body tilt and roll.

A.2.2 Transform for Back legs

Next, we shall present the transform that helps represent the points in the coordinate frame at the tip of the leg with respect to the coordinate frame at the shoulder joint of the leg. We need to consider only the rear legs in our code. This also makes sense considering that at least one of the rear legs of the robot is always on the ground while the robot is involved in the game. Since the transform for the left and right rear legs are quite similar, we shall elaborate on the left rear leg alone.

Figure 12 shows the coordinate frames corresponding to the left rear leg of the robot.

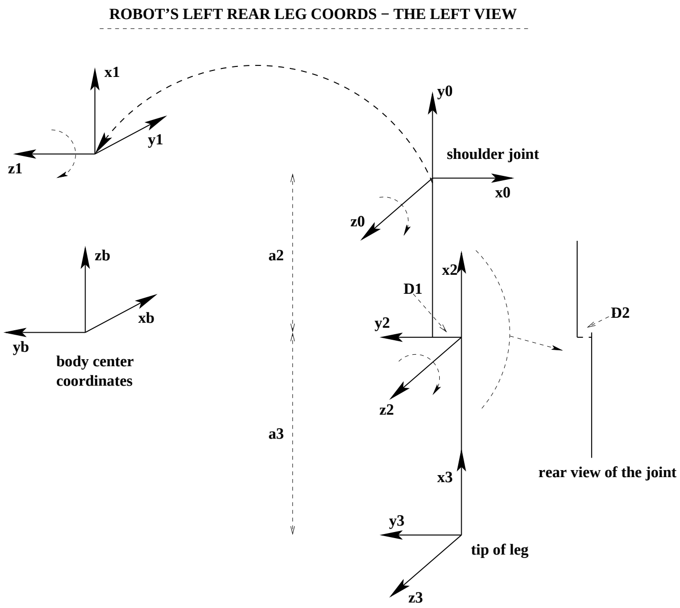

Figure 12: Coordinate system of the left rear leg - as seen from the left.

Using the previously described principles (for the camera transforms), we can then arrive at the following list of motions (in the order mentioned) for the transform from the tip of the leg to the shoulder joint:

1. Rotation about the fixed frame y-axis.
2. Rotation about the moving frame x-axis.
3. Translation of ($-a_2 = 69.5mm$) along moving frame z-axis.
4. Translations about the moving frame x and y axes by ($-D_1 = -9.0mm$) and ($D_2 = 4.7mm$) respectively.
5. Rotation about the moving frame y-axis.
6. Translation of ($-a_3 = 79.4mm$) about the moving frame z-axis.

This leads to the following order of matrix multiplications:

$$T_{left-should}^{left-tip} = \begin{pmatrix} c_1 & 0 & s_1 & 0 \\ 0 & 1 & 0 & 0 \\ -s_1 & 0 & c_1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c_2 & -s_2 & 0 \\ 0 & s_2 & c_2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -D_1 \\ 0 & 1 & 0 & D_2 \\ 0 & 0 & 1 & -a_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_3 & 0 & s_3 & 0 \\ 0 & 1 & 0 & 0 \\ -s_3 & 0 & c_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -a_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

which simplifies to yield:

$$T_{left-should}^{left-tip} = \begin{pmatrix} c_1c_3 - s_1c_2s_3 & s_1s_2 & c_1s_3 + s_1c_2c_3 & -c_1(a_3s_3 + D_1) + s_1(D_2s_2 - a_2c_2 - a_3c_2c_3) \\ s_2s_3 & c_2 & -s_2c_3 & D_2c_2 + a_2s_2 + a_3s_2c_3 \\ -s_1c_3 - c_1c_2s_3 & c_1s_2 & -s_1s_3 + c_1c_2c_3 & s_1(a_3s_3 + D_1) + c_1(D_2s_2 - a_2c_2 - a_3c_2c_3) \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{21}$$

The matrix $T_{left-should}^{left-tip}$ provides the transformation from the tip of the left rear leg to the shoulder of the leg of the robot. We can then use the transforms calculated earlier 18 to obtain the transformation matrix that transforms between the tip of the left rear leg to the center-of-the-body coordinate frame:

$$T_{BodyC}^{left-tip} = T_{BodyC}^{left-should} T_{left-should}^{left-tip} \tag{22}$$

By a similar process, we can calculate the corresponding transform for the right leg (first calculate $T_{right-should}^{right-tip}$) as:

$$T_{BodyC}^{right-tip} = T_{BodyC}^{right-should} T_{right-should}^{right-tip}$$

We therefore have all the necessary transformation matrices to perform the required operations in vision and localization modules.

B Pixel Projection - Image plane to ground plane

In this section we shall provide details on the process of projecting pixels from the image plane to the ground plane using domain knowledge (used in Section 3.4). We know that any object of interest (beacons, goals, ball) is in contact with the field and is at a fixed known height from the ground plane. This knowledge can be used, in the absence of stereo-vision, to determine the projection of the object on a plane corresponding to the height of the centroid of the object from the ground plane. This is most useful for projecting the line pixels because we then know the location of a candidate edge pixel on the ground relative to the robot. We shall use this as an example to illustrate the procedure. Given a candidate edge pixel on the image plane, in the image coordinate frame, the problem is that of finding the actual location of this point on the ground plane, relative to the robot. It follows the following steps:

1. First, we use the transformation matrices described above to determine the 3-dimensional location of the candidate pixel and the focal point of the camera with respect to the frame of reference at the center of the robot's body. The transform for the focal point needs to be done only once per image while for each candidate pixel, we first use equation 15 to transform from the image to the camera coordinate frame. Then, if img_{cam} represents the pixel location in camera coordinate frame and $focal_{cam}$ represents the focal point location in the camera frame of reference, the values relative to the center of the robot's body are:

$$\begin{aligned} focal_{BodyC} &= T^{cam}_{BodyC} \cdot focal_{cam} \\ img_{BodyC} &= T^{cam}_{BodyC} \cdot img_{cam} \end{aligned} \tag{23}$$

2. We then use the height of the back legs of the robot to determine the height of the center of the robot's body with respect to the ground plane, using the *body transforms* described in the previous appendix..

3. We use both these values to arrive at the relative projection of the pixel on the ground plane:

C Line and Line Intersection Thresholds

In this appendix we provide the thresholds that we actually used in the code for determining the lines and line intersections in the image (Section 3.4).

1. An image pixel is accepted as a candidate edge pixel *iff* it corresponds to a white-green transition and at least 75% of $20-25$ pixels below that pixel (in the image plane) are green.

2. We accept candidate line pixels only if the distance to its projection on the ground plane, relative to the robot, does not exceed the threshold of $1800mm$. Also, based on this and the horizontal field of view of the robot's camera (56.9^o), we set a limit on the actual offset allowed in the image plane. The pixel projection (on the ground plane) should lie within the robot's field of view.

3. The line type of an edge pixel is determined based on the color of the major proportion of pixels in a set of around 30 pixels above the pixel, in the image plane.

4. A pixel is added to the closest cluster of pixels corresponding to an existing line if it is within 15 pixels of the existing line.

5. A cluster of pixels belonging to the same line is utilized to generate the candidate lines *iff* the cluster consists of at least 7 pixels.

6. A maximum of 5 clusters of pixels are finally retained, i.e., we determine a maximum of five (the largest five) lines per input image.

7. Two candidate lines are considered for determining the intersection, if any, *iff* the absolute value of the acute angle between them is $\geq 15^o$. This helps filter out a lot of noise and helps eliminate calculation of intersection between fragmented line segments belonging to the same line.

8. To filter out noisy line intersections, we reject any candidate line intersection points that are more than $1500mm$ away from the robot.

Also, some of the predicates/functions used include:

- *Possible-Edge-Pixel*
- *Find-Pixel-Projection*
- *Closest Cluster*
- *Find-Line-Type*

D Simulator Message Grammar

This section contains a complete grammar for the simulator messages sent from the client and server in our simulator that we used for development of localization algorithms (see Section 7.6).

⟨CLIENT-MSG⟩→ ⟨INIT-MSG⟩ | ⟨PARAM-WALK-MSG⟩ | ⟨MOVE-HEAD-MSG⟩ | ⟨INFO-MSG⟩
⟨INIT-MSG⟩→ `( init )`
⟨PARAM-WALK-MSG⟩→ `( param_walk` ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ `)`
⟨MOVE-HEAD-MSG⟩→ `( move_head` ⟨INTEGER⟩ ⟨INTEGER⟩ `)`
⟨INFO-MSG⟩→ `( info` ⟨INFO⟩* `)`
⟨INFO⟩→ ⟨ESTIMATE-INFO⟩ | ⟨PARTICLE-INFO⟩ | ⟨BALL-PARTICLE-INFO⟩
⟨ESTIMATE-INFO⟩→ `( e` ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ `)`
⟨PARTICLE-INFO⟩→ `( p` ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ `)`
⟨BALL-PARTICLE-INFO⟩→ `( b` ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ `)`

⟨SERVER-MSG⟩→ ⟨CONNECT-MSG⟩ | ⟨SENSE-MSG⟩ | ⟨SEE-MSG⟩ | ⟨ERROR-MSG⟩
⟨CONNECT-MSG⟩→ `( connect )`
⟨SENSE-MSG⟩→ `( sense` ⟨INTEGER⟩ ⟨SENSATION⟩* `)`
⟨SENSATION⟩→ ⟨HEAD-SENSE⟩
⟨HEAD-SENSE⟩→ `( h` ⟨INTEGER⟩ ⟨INTEGER⟩ `)`
⟨SEE-MSG⟩→ `( see` ⟨INTEGER⟩ ⟨OBSERVATION⟩* `)`
⟨OBSERVATION⟩→ ⟨BALL-OBS⟩ | ⟨LANDMARK-OBS⟩
⟨BALL-OBS⟩→ `( b` ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ `)`
⟨LANDMARK-OBS⟩→ `( l` ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ ⟨INTEGER⟩ `)`
⟨ERROR-MSG⟩→ `( error` ⟨STRING⟩ `)`

⟨INTEGER⟩→ `0` | `[\-]? [1-9] [0-9]*`
⟨STRING⟩→ `[a-zA-Z\-]`$^+$

D.1 Client Action Messages

The following messages are sent by the client to change its action.

- **(param_walk** ∂x ∂y $\partial\theta$**)**

  Set the robot's translational velocity to $\langle \partial x, \partial y\rangle$ in mm/s and its rotational velocity to $\partial\theta$ in degrees per second.

- **(move_head** *pan tilt***)**

  Move robot's head to angles *pan* and *tilt* in degrees.

D.2 Client Info Messages

The following strings are sent by the client in **info** messages to supply internal state information to the server.

- **(e** x y θ *certainty pan tilt***)**

  The current pose estimate produced by localization is $\langle x, y, \theta\rangle$ in mm and degrees. Its pose estimate confidence is *certainty*%. Its estimate of its head joint angles are *pan* and *tilt* in degrees.

- **(p** x y θ p**)**

  One of the robot's localization particles has pose $\langle x, y, \theta\rangle$ in mm and degrees and probability p%.

- **(b** x y p**)**

  One of the robot's ball particles has position $\langle x, y\rangle$ in mm and degrees and probability p%.

D.3 Simulated Sensation Messages

The simulator's **sense** messages, which act to emulate the robot's sensors and joint feedback are formatted as follows.

- (**sense** *cycle sensations* ...)

  The *sensations* were sensed at time step *cycle*.

The following sensation strings are sent as part of **sense** messages. Currently, only one type of sensation is supported.

- (**h** *pan tilt*)

  The robot's head joint feedback returns angles *pan* and *tilt* in degrees.

D.4 Simulated Observation Messages

The simulator's **see** messages, which act to emulate the robot's vision module are formatted as follows.

- (**see** *cycle observations* ...)

  The *observations* were made at time step *cycle*.

The following observation strings are sent as part of **see** messages.

- (**b** d α *certainty*)

  The robot observes a ball at distance d in mm at heading α in degrees with confidence *certainty*%.

- (**l** *id* d α *certainty*)

  The robot observes a landmark #*id* at distance d in mm at heading α in degrees with confidence *certainty*%.