UT Austin Villa 2003: A New RoboCup Four-Legged Team

Peter Stone, Kurt Dresner, Selim T. Erdoğan, Peggy Fidelman,
Nicholas K. Jong, Nate Kohl, Gregory Kuhlmann, Ellie Lin,
Mohan Sridharan, Daniel Stronger, Gurushyam Hariharan

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

Technical Report UT-AI-TR-03-304

October 6, 2003

Abstract

The UT Austin Villa RoboCup 2003 Four-Legged Team was a new entry in the ongoing series of RoboCup legged league competitions. The team development began in mid-January of 2003, at which time none of the team members had any familiarity with the Aibos. Without using any RoboCup-related code from other teams, we entered a team in the American Open competition at the end of April, and met with some success at the annual RoboCup competition that took place in Padova, Italy at the beginning of July. In this report, we describe both our development process and the technical details of its end result, the UT Austin Villa team. The main contributions of this paper are (i) a roadmap for new teams entering the competition who are starting from scratch, and (ii) full documentation of the algorithms behind our approach with the goal of making them fully replicable.

Contents

1 Introduction

RoboCup, or the Robot Soccer World Cup, is an international research initiative designed to advance the fields of robotics and artificial intelligence by using the game of soccer as a substrate challenge domain. The long-term goal of RoboCup is, by the year 2050, to build a full team of 11 humanoid robot soccer players that can beat the best human soccer team on a real soccer field [1].

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

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

For the purposes of this report, we assume familiarity with the specifications of the robots as well as 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. The main contributions of this report are

1. A roadmap for new teams entering the competition who are starting from scratch; and
2. 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, at which time none of the team members had any familiarity with the Aibos. Without using any RoboCup-related code from any other teams, we entered a team in the American Open competition at the end of April, and met with some success at the annual RoboCup competition that took place in Padova, Italy at the beginning of July. Although our team was not one of the top few at the competition, we view it as a great accomplishment that we were able to develop a competitive team in such a short time. A primary goal of this report is to document our development process as a guide for new teams in the future.

Our effort began as a graduate research seminar offered as a class during the Spring semester of 2003. The following section outlines the structure of the class. At the end of that section we outline the structure of the remainder of the paper.

2 The Class

The UT Austin Villa 2003 legged robot team began as a focused class effort during the Spring semester of 2003 at the University of Texas at Austin. Nineteen graduate students and one undergraduate were enrolled in the course CS395T: *Multi-Robot Systems: Robotic Soccer with Legged Robots.*[4] All of the authors on this paper participated in the class.

Students in the class studied past approaches, both as described in the literature and as reflected in publicly available source code. However, we developed the entire code base *from scratch* with the goals of learning about all aspects of robot control and of introducing a completely new code base to the community.

Class sessions were devoted to students educating each other about their findings and progress, as well as coordinating the integration of everybody's code. Just nine weeks after their initial introduction to the robots, the students already had preliminary working solutions to vision, localization, fast walking, kicking, and communication.

The concrete goal of the course was to have a completely new working solution by the end of April so that we could participate in the American Open competition, which happened to fall during the last week of the class. After that point, a subset of the students continued working towards RoboCup 2003 in Padova.

[1] `http://www.openr.org/robocup/index.html`
[2] `http://openr.aibo.com/`
[3] `http://www.cs.utexas.edu/~AustinVilla`
[4] `http://www.cs.utexas.edu/~pstone/Courses/395Tspring03`

The class was organized into three phases. Initially, the students created simple behaviors with the sole aim of becoming familiar with Open-R.

Then, about two weeks into the class we shifted to phase two by identifying key subtasks that were important for creating a complete team. Those subtasks were:

- Vision;
- Movement;
- Fall Detection;
- Kicking;
- Localization;
- Communication;
- General Architecture; and
- Coordination.

During this phase, students chose one or more of these subtasks and worked in subgroups on generating initial solutions to these tasks in isolation.

By about the middle of March, we were ready to switch to phase three, during which we emphasized "closing the loop," or creating a single unified code-base that was capable of playing a full game of soccer. We completed this integration process in time to enter a team in the RoboCup American Open competition at the end of April.

The following sections chronicle our progress towards our RoboCup 2003 entry. All of the subtopics addressed in phase two of the class continued to be improved throughout our development process. For clarity of presentation, we present our eventual solutions in the same sections in which we introduce our initial approaches. In so doing, we make an effort to document the evolution of ideas that led to our final solutions, though in general we give full details only for our final solutions. Subsequent sections address our final integration efforts as well as our experiences at the competition.

The remainder of the report is organized as follows. In Section 3 we document some of the initial behaviors that were generated during phase one of the class. Next we document the output of some of the subgroups that were formed in phase two of the class: vision in Section 4; movement in Section 5; fall detection in Section 6; kicking in Section 7; localization in Section 8; and communication in Section 9. In each of these sections we fully document our solutions to the subtasks as of RoboCup 2003 in July. Next, we document the tasks that occupied phase three of the class, namely those that allowed us to put together the above modules into a cohesive code base. In Section 10 we describe our general architecture that combines sensing, decision-making, and acting. In Section 11 we introduce *global maps*, our main state representation. Section 12 describes our soccer-playing behaviors such as goal-scoring and goaltending. Then in Section 13 we document our methods for coordinating the behaviors of the robots as a team. Section 14 introduces our debugging and development tool. Then in Section 15 we summarize our experiences at the American Open and RoboCup 2003 competitions, and Section 16 concludes.

3 Initial Behaviors

The first task for the students in the class was to learn enough about the Aibo to be able to compile and run any simple program on the Aibo.

The open source release of Open-R came with several sample programs that could be compiled and loaded onto the Aibo right away. These programs could do simple tasks such as:

L-Master-R-Slave: Cause the right legs to mirror manual movements of the left legs.

Ball-Tracking-Head: Cause the head to turn such that the pink ball is always in the center of the visual image (if possible).

PIDcontrol: Move a joint to a position specified by the user by typing in a telnet window.

The students were to pick any program and modify it, or combine two programs in any way. The main objective was to make sure that everyone was familiar with the process for compiling and running programs on the Aibos. Some of the resulting programs included:

- Variations on L-Master-R-Slave in which different joints were used to control each other. For example, one student used the tail as the master to control all 4 legs, which resulted in a swimming type motion. Doing so required scaling the range of the tail joints to those of the leg joints appropriately.
- Variations on Ball-Tracking-Head in which a different color was tracked. Two students teamed up to cause the robot to play different sounds when it found or lost the ball.
- Variations on PIDcontrol such that more than one joint could be controlled by the same input string.

After becoming familiar with the compiling and uploading process, the next task for the students was to become more familiar with the Aibo's operating system and the Open-R interface. To that end, they were required to create a program that added at least one new subject-observer connection to the code.[5] The students were encouraged to create a new Open-R object from scratch. Pattern-matching from the sample code was encouraged, but creating an object as different as possible from the sample code was preferred.

Some of the responses to this assignment included:

- The ability to turn on and off LEDs by pressing one of the robots' sensors.
- A primitive walking program that walks forward when it sees the ball.
- A program that alternates blinking the LEDs and flapping the ears.

After this assignment, which was due after just the second week of the class, the students were familiar enough with the robots and the coding environment to move on to their more directed tasks with the aim of creating useful functionality.

4 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 located on the Aibo. The module identifies colors in order to recognize objects, which are then used to localize the robot and to plan its operation.

Our visual processing is done using the established procedure of color segmentation followed by object recognition. Color segmentation is the process of classifying each pixel in an input image as belonging to one of a number of predefined color classes based on the knowledge of the ground truth on a few training images. Though the fundamental methods employed in this module have been applied previously (both in RoboCup and in other domains), it has been built from scratch like all the other modules in our team. Hence, the implementation details provided are our own solutions to the problems we faced along the way. We have drawn some of the ideas from the previous technical reports of CMU [2] and UNSW [4]. This module can be broadly divided into two stages: (i) low-level vision, where the color segmentation and region building operations are performed, and (ii) high-level vision, wherein object recognition is accomplished and the position and bearing of the various objects in the visual field are determined. The following sections present detailed descriptions of these processes. But first, we present a brief overview of the robot's CMOS color camera.

[5] A subject-observer connection is a pipe by which different Open-R objects can communicate and be made interdependent. For example, one Open-R object could send a message to a second object whenever the back sensor is pressed, causing the second object to, for example, suspend its current task or change to a new mode of operation.

4.1 Camera Settings

As mentioned previously, the robot comes equipped with a CMOS color camera that operates at a frame rate of $25 fps$. Some of its other preset features are:

- Horizontal viewing angle: 57.6°.
- Vertical viewing angle: 47.8°.
- Lens Aperture: 2.0.
- Focal length: 2.18mm.

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. $Indoor - mode$: 2800K.
  2. $FL - mode$: 4300K.
  3. $Outdoor - mode$: 7000K.

  This setting, as the name suggests, is basically a color correction system to accomodate 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 helps in better object recognition. The optimum setting depends on the 'light temperature' registered on the field (this 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 in distinguishing the orange ball from the other yellow objects on the field such as the goal and sections of the beacons.
- $ShutterSpeed$:
  1. Slow: $1/50 sec$.
  2. Mid: $1/100 sec$.
  3. Fast: $1/200 sec$.

  This setting denotes the time for which the shutter of the camera 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 would result in blurred images.
- *Gain:*
  1. Low: $0 dB$.
  2. Mid: $0 dB$.
  3. High: $6 dB$.

  This parameter sets the camera gain. In this case, we did not notice any major difference in performance among the three settings provided.

4.2 Color Segmentation

The image captured by the robot's camera, in the YCbCr format, is a set of numbers, ranging from 0 to 255 along each dimension, representing luminance (Y) and chrominance (Cb, Cr). To enable the robot to extract useful information from these images, the numbers have to be suitably mapped into an appropriate color space. We retain the YCbCr format and "train" the robot, using a Nearest Neighbor (NNr) scheme [8, 4], to recognize and distinguish between 10 different colors, numbered as follows:

- 0 = pink,
- 1 = yellow,
- 2 = blue,
- 3 = orange,
- 4 = marker green,
- 5 = red,
- 6 = dark (robot) blue,
- 7 = white,
- 8 = field green,
- 9 = black.

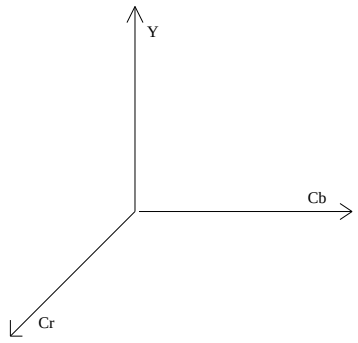
The motivation behind using the NNr approach is that the colors under consideration overlap in the YCbCr space (some, such as orange and yellow, do so by a significant amount). Unlike other common methods that try to divide the color space into cuboidal regions (or a collection of planes), the NNr scheme allows us to learn a color table where the individual blobs are defined more precisely.

The original color space has three dimensions, corresponding to the Y, Cb, and Cr channels of the input image. To build the color table (used for classification of the subsequent images on the robot), we maintain three different types of color cubes in the training phase: one Intermediate (IM) color cube corresponding to each color, a Nearest Neighbor cube, and a Master (M) cube (the names will make more sense after the description given below). To reduce storage requirements, we operate at half the resolution, i.e. all the cubes have their numerical values scaled to range from 0 to 127 along each dimension. The cells of the IM cubes are all initialized to zero, while those of the NNr cube and the M cube are initialized to 9 (the color black, also representing background).

Color segmentation begins by first training on a set of images using UT Assist, our Java-based interface/debugging tool (for more details see Section 14). A robot is placed at a few points on the field. Images are captured and then transmitted over the wireless network to a remote computer running the Java-based server application. The objects of interest (goals, beacons, robots, ball, etc.) in the images are manually "labeled" as belonging to one of the color classes previously defined, using the Image Segmenter (see Section 14 for some pictures showing the labeling process). For each pixel of the image that we label, the cell determined by the corresponding YCbCr values (after transforming to half-resolution), in the corresponding IM cube, is incremented by 3 and all cells a certain Manhattan distance away (within 2 units) from this cell are incremented by 1. For example, if we label a pixel on the ball orange in the image and this pixel corresponds to a cell $(115, 35, 60)$ based on the intensity values of that pixel in the image, then in the orange IM cube this cell is incremented by 3 while the cells such as $(115, 36, 61)$ and $(114, 34, 60)$ (among others) which are within a Manhattan distance of 2 units from this cell, in the orange IM cube alone, are incremented by 1. For another example, see Figure 1.

The training process is performed incrementally, so at any stage we can generate a single cube (the NNr cube is used for this purpose) that can be used for segmenting the subsequent images. This helps us see how "well-trained" the system is for each of the colors and serves as a feedback mechanism that lets us decide which colors need to be trained further. To generate the NNr cube, we traverse each cell in the NNr cube and compare the values in the corresponding cell in each of the IM cubes and assign to this cell the index of the IM cube that has the maximum value in this cell, i.e., $\forall (p, q, r) \in [0, 127]$,

$$NNrCube(y_p, cb_q, cr_r) = arg \max_{i \in [0,9]} IM_i(y_p, cb_q, cr_r) \tag{1}$$



0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

0	0	1	0	0
0	1	1	1	0
1	1	3	1	1
0	1	1	1	0
0	0	1	0	0

(a) (b) (c)

Figure 1: An example of the development of the color table, specifically the IM cube. Part(a) shows the general coordinate frame for the color cubes. Part(b) shows a planar subsection of one of the IM cubes before labeling. Part(c) depicts the same subsection after the labeling of a pixel that maps to the cell at the center of the subsection. Here only one plane is shown - the same operation occurs across all planes passing through the cell under consideration such that all cells a certain Manhattan distance away from this cell are incremented by 1.

When we use this color cube to segment subsequent images, we use the NNr scheme. For each pixel in the test image, the YCbCr values (transformed to half-resolution) are used to index into this NNr cube. Then we compute the weighted average of the value of this cell and those cells that are a certain Manhattan distance (we use 2-3 units) around it to arrive at a value that is set as the "numerical color" (i.e. the color class) of this pixel in the test image. The weights are proportional to the Manhattan distance from the central cell, i.e., the greater this distance the smaller the significance attached to the value in the corresponding cell (see Figure 2).

We do the training over several images (around 20-30) by placing the robot at suitable points on the field. The idea here is to train on images that capture the beacons, goals, ball and the robots from different distances (and also different angles for the ball) to account for the variations in lighting along different points on the field. This is especially important for the orange ball, whose color could vary from orange to yellow to brownish-red depending on the amount of lighting available at that point. We also train with several different balls to account for the fact that there is a marked variation in color among different balls. At the end of the training process, we have all the IM cubes with the corresponding cells suitably incremented. The NNr operation is computationally intensive to perform on the robot's processor. To overcome this, we precompute the result of performing this operation (the Master cube is used for this) from the corresponding cells in the NNr color cube, i.e. we traverse each cell of the M Cube and compute the "Nearest Neighbor" value from the corresponding cells in the NNr cube. In other words, $\forall (p,q,r) \in [0,127]$ with a predefined Manhattan distance $ManDist \in [3,7]$,

$$MCube(y_p, cb_q, cr_r) = arg \max_{i \in [0,9]} Score(i) \tag{2}$$

where $\forall (k_1,k_2,k_3) \in [0,127]$,

$$Score(i) = \left(\sum_{k_1,k_2,k_3} (ManDist - (\mid k_1 - p \mid + \mid k_2 - q \mid + \mid k_3 - r \mid) \right) \mid (\mid k_1 - p \mid + \mid k_2 - q \mid + \mid k_3 - r \mid) < ManDist \wedge \quad NNrCube(y_{k_1}, cb_{k_2}, cr_{k_3}) = i. \tag{3}$$

9	3	1	1	9
3	1	3	3	3
3	3	1	3	3
1	1	3	3	3
9	1	3	3	9

(a)

9	3	1	1	9
3	1	3	3	3
3	3	3	3	3
1	1	3	3	3
9	1	3	3	9

(b)

Figure 2: An example of the weighted average applied to the NNr cube (a 2-dimensional representative example). Part (a) shows the values along a plane of the NNr cube before the NNr scheme is applied to the central cell. Part (b) shows the same plane after the NNr update for its central cell. We are considering cells within a Manhattan distance of 2 units along the plane. For this central cell, color label 1 gets a vote of $3+1+1+1=6$ while label 3 gets a vote of $2+2+2+2+1+1+1+1+1=13$ which makes the central cell's label = 3. This is the value that is set as the classification result. This is also the value that is stored in the cell in the M cube that corresponds to the central cell.

This cube is loaded onto the robot's memory stick. This then makes color segmentation on the robot a simple process of table lookup, thereby making it a lot faster. (For an example of the color segmentation process and the Master Cube generated at the end of it, see Figure 17).

One important point about our color segmentation scheme is that we do not (at present) make an effort to normalize the cubes based on the number of pixels (of each color) that we train on. So, if we labeled a number of yellow pixels and a relatively smaller number of orange pixels, then we would be biased towards yellow in the NNr cube. This is not a problem if we are careful during the training process and label regions such that all colors get (roughly) equal representation. We leave a principled treatment of the problem of normalization to future research.

4.3 Region Building and Merging

The Master cube is loaded onto the robot's memory stick and this is used to segment the images that the robot's camera captures (in real-time). The next step in low-level processing involves the formation of rectangular bounding boxes around connected regions of the same color. This in turn consists of run-length encoding (RLE) and region merging [7], which are standard image processing approaches used previously in the RoboCup domain [2].

As each image is segmented (during the first scan of the image), left to right and top to bottom, it is encoded in the form of run-lengths along each horizontal scan line i.e. along each line we store the (x, y) position (the root node) where a sequence of a particular color starts and the number of pixels until a sequence of another color begins. The data corresponding to each run-length is stored in a separate data structure (called RunRegion) and the run-lengths are all stored as a linked list. Each RunRegion data structure also stores the corresponding color. Further, there is a bounding box corresponding to each RunRegion/run-length, which during the first pass is just the run-length itself, but has additional properties such as the number of run-lengths enclosed, the number of actual pixels enclosed, the upper left (UL) and lower right (LR) corners of the box etc. Each run-length has a pointer to the next run-length of the same color (null if none exists) and an index corresponding to the bounding box that it belongs to, while each bounding box has

a pointer to the list of run-lengths that it encloses. This facilitates the easy merging of two run-lengths (or a bounding box containing several run-lengths with a single run-length or two bounding boxes each having more that one run-length). The RunRegion data structure and the BoundingBox data structure are given in Table 1.

```
// The Runregion data structure definition.

struct RunRegion {
     int color;   //color associated with the run region.
     RunRegion*  root;   //the root node of the runregion.
    RunRegion*  listNext;   //pointer to the next runregion in the current run length.
    RunRegion*  nextRun;
    int  xLoc;   //x location of the root node.
    int  yLoc;   //y location of the root node.
    int  runLength;   // number of run lengths with this region.
    int  boundingBox;   //the bounding box that this region belongs to.
};


// The  BoundingBox  data structure definition.

struct BoundingBox {
     BoundingBox*  prevBox;  //pointer to the previous bounding box.
     BoundingBox*  nextBox;   // pointer to the next bounding box.
    int  ULx;  //upper left corner x coordinate.
    int  ULy;   //upper left corner y coordinate.
    int  LRx;
    int  LRy;
    bool  lastBox;
    int  valid;
    int  numRunLengths;  //number of runlengths associated with this bounding box.
    int  numPixels;   //number of pixels in this bounding box.
    int rrcount;
    int color;  //color cooresponding to this bounding box.
    RunRegion*  listRR;
    RunRegion*  eoList;
    float  prob;   //probability corresponding to this bounding box.
 };
```

Table 1: This table shows the basic run region and bounding box data structures with which we operate.

Next, we need to merge the run-lengths/bounding boxes corresponding to the same object together under the assumption that an object in the image will be represented by connected run-lengths. In the second pass, we proceed along the run-lengths (in the order in which they are present in the linked list) and check for pixels of the same color immediately below each pixel over which the run-length extends, merging run-lengths of the same color that have significant overlap (the threshold number of pixel overlap is decided based on experimentation: see Appendix A.1). When two run-lengths are to be merged, one of the bounding boxes is deleted while the other's properties (root node, number of run-lengths, size etc) are suitably modified to include both the bounding boxes. This is accomplished by moving the corresponding pointers around appropriately. By incorporating suitable heuristics, we remove bounding boxes that are not significantly large or dense enough to represent an object of interest in the image, and at the end of this pass, we end up with a number of candidate bounding boxes, each representing a blob of one of the nine colors under consideration. The bounding boxes corresponding to each color are linked together in a separate linked list, which (if required) is sorted in descending order of size for ease of further processing. Details of the heuristics used here can be found in Appendix A.1.

4.4 Object Recognition with Bounding Boxes

Once we have bounding boxes of the various colors arranged in separate lists, we can proceed to high-level vision, i.e., the detection of objects of interest in the robot's image frame. The objects that we primarily need to identify in the visual field are the ball, the two goals, the field markers (other than the goals) and the opponents. This stage takes as input the lists of bounding boxes and provides as output a collection of objects (structures called the *VisionObjects*), one for each detected object, which are then used for determining the position and bearing of these objects with respect to the robot. This information is in turn used in the localization module (see Section 8) to calculate the robot's position in the field coordinates. To identify these objects we introduce some constraints and heuristics, some of which are based on the known geometry of the environment while others are parameters that we identified by experimentation. We first document the basic process used to search for the various objects, and at the end of the section we provide a description of the constraints and heuristics used.

We start with the goals because they are generally the largest blobs of the corresponding colors and once found they can be used to eliminate spurious blobs during beacon and ball detection. We search through the lists of bounding boxes for colors corresponding to the goals (blue and yellow) on the field, given constraints on size, aspect ratio and density Furthermore, checks are included to ensure that spurious blobs (noisy estimates on the field, blobs floating in the air, etc.) are not taken into consideration. On the basis of these constraints we compare the blob found in the image (and identified as a goal) with the known geometry of the goal. This provides some sort of likelihood measure, and a *VisionObject* is created to store this and the information of the corresponding bounding box. (Table 2 displays the data structures used for this purpose)

```
struct VisionObjects{
    int NumberOfObjects;  //number of vision obejcts in curretn frame.
    BBox*  ObjectInfo;    //array of objects in view.
}

struct BBox {
   int ObjID; //object ID.
   Point ul;  //upper left point of the bounding box.
   Point lr;  //lower right point of the bounding box.
   double  prob; //likelihood corresponding to this bounding box/object.
}

struct Point {
    double x;  //x coordinate.
    double  y;  //y coordinate.
}
```

Table 2: This table shows the basic VisionObject and associated data structures with which we operate.

After searching for the goals, we search for the orange ball, probably the most important object in the field. We sort the orange bounding boxes in descending order of size and search through the list (not considering very small ones), once again based on heuristics on size, aspect ratio, density, etc. To deal with cases with partial occlusions, which is quite common with the ball on the field, we use the "circle method" to estimate the equation of the circle that best describes the ball (see Appendix A.3 for details). Basically this involves finding three points on the edge of the ball and finding the equation of the circle passing through the three points. This method seems to give us an accurate estimate of the ball size (and hence the ball distance) in most cases. In the case of the ball, in addition to the check that helps eliminate spurious blobs (floating in the air), checks have to be incorporated to ensure that minor misclassification in the segmentation stage (explained below) do not lead to detection of the ball in places where it does not exist.

Next, we tackle the problem of finding the beacons (six field markers, excluding the goals). The identification of beacons is important in that the accuracy of localization of the robot depends on the determination of the position and bearing of the beacons (with respect to the robots) which in turn depends on the proper determination of the bounding boxes associated with the beacons. Since the color pink appears in all beacons, we use that as the focus of our search. Using suitable heuristics to account for size, aspect ratio, density, etc. we match each pink blob with blue, green, or yellow blobs to determine the beacons. We ensure that only one instance of each beacon (the most likely one) is retained. Additional tests are incorporated to remove spurious beacons: those that appear to be on the field or in the opponents, floating in the air, inappropriately huge or tiny, etc. For details, see Appendix A.4.

After this first pass, if the goals have not been found, we search through the candidate blobs of the appropriate colors with a set of reduced constraints to determine the occurrence of the goals (which results in a reduced likelihood estimate as we will see below). This is useful when we need to identify the goals at a distance, which helps us localize better, as each edge of the goal serves as an additional marker for the purpose of localization.

We found that the goal edges were much more reliable as inputs to the localization module than were the goal centers. So, once the goals are detected, we determine the edges of the goal based on the edges of the corresponding bounding boxes. Of course, we include proper buffers at the extremities of the image to ensure that we detect the actual goal edges and not the 'artificial edges' generated when the robot is able to see only a section of the goal (as a result of its view angle) and the sides of the truncated goal's bounding box are mistaken to be actual edges.

Next, we present a brief description of some of the heuristics employed in the detection of ball, goals, beacons and opponents. We begin by listing the heuristics that are common to all objects and then also list those that are specific to goals, ball and/or beacons. For more detailed explanations on some methods and parameters for individual test see the corresponding appendices.

- *Spurious blob elimination:* A simple calculation using the tilt angle of the robot's head is used to determine and hence eliminate spurious (beacon, ball and/or goal) blobs that are too far down or too high up in the image plane. See Appendix A.2 for the actual thresholds and calculations.

- *Likelihood Calculation:* For each object of interest in the robot's visual field, we associate a measure which describes how sure we are of our estimation of the presence of that object in the current image frame. The easiest way to accomplish this would be to compare the aspect ratio (the ratio of the height to the width) of the bounding boxes that identify these objects, to the actual known aspect ratio of the objects in the field. For example, the goal has an aspect ratio of $1:2$ in the field, and we can compare the aspect ratio of the bounding box that has been detected as the goal with this expected ratio. We can claim that the closer these two values are, the more sure we are of our estimate and hence higher is the *likelihood*.

- *Beacon specific calculations:*

  1. To remove spurious beacons, we ensure that the two sections that form the beacon are of comparable size, i.e. that each section is at least half as large and half as dense as the other section.
  2. We ensure that the separation between the two sections is within a small threshold, which is usually $2-3$ pixels.
  3. We compare the aspect ratio of bounding box corresponding to the beacon in the image to the actual aspect ratio ($2:1 :: height : width$), which helps remove candidate beacons that are too small or disproportionately large.
  4. Aspect ratio, as mentioned above is further used to determine an estimate of the likelihood of each candidate beacon that also helps choose the "most probable" candidate when there are multiple occurrences of the same beacon. Only beacons with a likelihood above a threshold are retained and used for localization calculations. This helps ensure that false positives, generated by lighting variations and/or shadows, do not cause major problems in localization.

Note: for sample threshold values, see Appendix A.4.

- *Goal specific calculations:*

  1. We use the 'tilt-angle test' (described in detail in Appendix A.2)
  2. We use a similar aspect ratio test for the goals, too. In the case of the goals we also look for sufficiently high density (the ratio of the number of pixels of the expected color to the area of the blob), the number of run-lengths enclosed, and a minimum number of pixels. All these thresholds were determined experimentally, and changing these thresholds changes the distance from which the goal can be detected and the accuracy of detection. For example, lowering these thresholds can lead to false positives.
  3. The aspect ratio is used to determine the likelihood, and the candidate is accepted iff it has a likelihood measure above a predefined minimum.
  4. When doing a second pass for the goal search, we relax the constraints slightly but proportionately a lower likelihood measure gets assigned to the goal, if detected.

  Note: for sample threshold values, see Appendix A.5.

- *Ball specific calculations:*

  1. We use the 'tilt-angle test' to eliminate spurious blobs from consideration.
  2. In most cases, the ball is severely occluded, precluding the use of the aspect ratio test. Nonetheless, we first search for an orange object with a high density and an aspect ratio (1:1) that would detect the ball if it is seen completely and not occluded.
  3. If the ball is not found with these tight constraints, we relax the aspect ratio constraint and include additional heuristics (e.g. if the ball is close, even if it is partially occluded, it should have a large number of run-lengths and pixels) that help detect a bounding box around the partially occluded ball. These heuristics and associated thresholds were determined experimentally.
  4. If the yellow goal is found, we ensure that the candidate orange ball does not occur within it and above the ground (which can happen since yellow and orange are close in color space).
  5. We check to make sure that the orange ball is found lower than the lower-most beacon in the current frame. Also, the ball cannot occur above the ground, or within or slightly below the beacon. The latter can occur if the white and/or yellow portions of the beacon are misclassified as orange.
  6. We use the "circle method" to detect the actual ball size. But we also include checks to ensure that in cases where this method fails and we end up with disproportionately huge or very small ball estimates (thresholds determined experimentally), we either keep the estimates we had before employing the circle method (and extend the bounding box along the shorter side to form a square to get the closest approximation to the ball) or reject the ball estimate in the current frame. The choice depends on the extent to which the estimated "ball" satisfies experimental thresholds.

  Note: for sample threshold values, see Appendix A.6.

Finally, we check for opponents in the current image frame. As in the previous cases, suitable heuristics are employed both to weed out the spurious cases and to determine the likelihood of the estimate. To identify the opponents, we first sort the blobs of the corresponding color in descending order of size, with a minimum threshold on number of pixels and run-lengths. We include a relaxed version of the aspect ratio test and strict tilt angle tests (an "opponent" blob cannot occur much lower or much higher than the horizon when the robot's head has very little tilt and roll) to further remove spurious blobs (see Appendix A.2 and Appendix A.7). Each time an opponent blob (that satisfies these thresholds) is detected, the robot tries to merge it with one of its previous estimates based on thresholds. If it does not succeed and it has less than

4 valid (previous) estimates it adds this estimate to the list of opponents. At the end of this process, each robot has a list that stores the four largest bounding boxes (that satisfy all these tests) of the color of the opponent with suitable likelihood estimates that are determined based on the size of the bounding boxes (see Appendix A.8). Further processing of opponent estimates using the estimates from other teammates etc is described in detail in the section on visual opponent modeling (Section 4.6). Once processing of the current visual frame is completed, the detected objects, each stored as a VisionObject is passed through the Brain to the GlobalMap module wherein the VisionObjects are operated upon using Localization routines.

4.5 Position and Bearing of Objects

The object recognition module returns a set of data structures, one for each "legal" object in the visual frame. Each object also has an estimate of its likelihood, which represents the uncertainty in our perception of the object. The next step (the final step in high-level vision) is to determine the distance to each such object from the robot and the bearing of the object with respect to the robot. In our implementation, this estimation of distance and bearing of all objects in the image, with respect to the robot, is done as a preprocessing step when the localization module kicks into action during the development of the global maps. Since this is basically a vision-based process we describe it here rather than in the section (Section 8) on localization. As each frame of visual input is processed, the robot has access to the tilt, pan, and roll angles of its camera from the appropriate sensors and these values give us a simple transform that takes us from the 3D world to the 2D image frame. Using the known projection of the object in the image plane and the geometry of the environment (the expected sizes of the objects in the robot's environment) we can arrive at estimates for the distance and bearing of the object relative to the robot. The known geometry is used to arrive at an estimate for the variances corresponding to the distance and the bearing. Suppose the distance and angle estimates for a beacon are d and θ. Then the variances in the distance and bearing estimates are estimated as:

$$variance_d = \left(\frac{1}{b_p}\right) \cdot (0.1d) \tag{4}$$

where $\left(\frac{1}{b_p}\right)$ is the likelihood of the object returned by vision.

$$variance_\theta = \tan^{-1}\left(\frac{beacon_r}{d}\right) \tag{5}$$

where $beacon_r$ is the actual radius of the beacon in the environment.

By similar calculations, we can determine the distance and bearing (and the corresponding variances) of the various objects in the robot's field of view.

4.6 Visual Opponent Modeling

Another important task accomplished using the image data is that of opponent modeling. As described in Section 4.4, each robot provides a maximum of four best estimates of the opponent blobs based on the current image frame. To arrive at an efficient estimate of the opponents (location of the opponents relative to the robot and hence with respect to the global frame), each robot needs to merge its own estimates with those communicated by its teammates. As such this process is accomplished during the development of the global maps (Section 11) but since the operation interfaces directly with the output from the vision module, it is described here.

When opponent blobs are identified in the image frame, the vision module returns the bounding boxes corresponding to these blobs. We noticed that though the shape of the blob and hence the size of the bounding box can vary depending on the angle at which the opponent robot is viewed (and its relative orientation), the height of the bounding box is mostly within a certain range. We use this information to arrive at an estimate of the distance of the opponent and use the centroid of the bounding box to estimate the bearing of the candidate opponent with respect to the robot (see Section 4.5 for details on estimation of

distance and bearing of objects). These values are used to find the opponent's (x, y) position relative to the robot and hence determine the opponent's global (x, y) position (see Appendix A.9 for details on transforms from local to global coordinates and vice versa). Variance estimates for both the x and the y positions are obtained based on the calculated distance and the likelihood associated with that particular opponent blob. For example, let d and θ be the distance and bearing of the opponent relative to the robot. Then, in the robot's local coordinate frame (determined by the robot's position and orientation), we have the relative positions as:

$rel_x = d \cdot \cos(\theta), \quad rel_y = d \cdot \sin(\theta)$

From these we obtain the global positions as:

$$\begin{pmatrix} glob_x \\ glob_y \end{pmatrix} = T_{local}^{global} \cdot \begin{pmatrix} rel_x \\ rel_y \end{pmatrix} \tag{6}$$

where T_{local}^{global} is the 2D-transformation matrix from local to global coordinates.

For the variances in the positions, we use a simple approach:

$$var_x = var_y = \frac{1}{Opp_{prob}} \cdot (0.1d) \tag{7}$$

where the likelihood of the opponent blob, Opp_{prob} is determined by heuristics (see Appendix A.8).

If we do not have any previous estimates of opponents from this or any previous frame, we accept this estimate and store it in the list of known opponent positions. If any previous estimates exist, we try to merge them with the present estimate by checking if they are close enough (based on heuristics). All merging is performed assuming Gaussian distributions. The basic idea is to consider the x and y position as independent Gaussians (with the positions as the means and the associated variances) and merge them (for more details see Section 8.3.3 and [10]). If merging is not possible and we have fewer than four opponent estimates, we treat this as a new opponent estimate and store it as such in the opponents list. But if four opponent estimates already exist, we try to replace one of the previous estimates (the one with the maximum variance in the list of opponent estimates and with a variance higher than the new estimate) with the new estimate. Once we have traversed through the entire list of opponent bounding boxes presented by the vision module, we go through our current list of opponent estimates and degrade all those estimates that were not updated, i.e. not involved in merging with any of the estimates from the current frame (for more details on the degradation of estimates, see the initial portions of Section 11 on global maps). When each robot shares its Global Map (see Section 11) with its teammates, this data gets communicated.

When the robot receives data from its teammates, a similar process is incorporated. The robot takes each current estimate (i.e. one that was updated in the current cycle) that is communicated by a teammate and tries to merge it with one of its own estimates. If it fails to do so and it has fewer than four opponent estimates, it accepts the communicated estimate as such and adds it to its own list of opponent estimates. But if it already has four opponent estimates, it replaces its oldest estimate (the one with the largest variance which is larger than the variance of the communicated estimate too) with the communicated estimate. If this is not possible, the communicated estimate is ignored.

This procedure, though simple, gives reliable results in nearly all situations once the degradation and merging thresholds are properly tuned. It was used both during games and in one of the challenge tasks (see Section 15.3) during RoboCup and the performance was good enough to walk from one goal to the other avoiding all seven robots placed in its path.

5 Movement

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

The Aibo comes with a stable but slow walk. From watching the videos of past RoboCups, and from reading the available technical reports, it became clear that a fast walk is an essential part of any RoboCup team. The walk is perhaps the most feasible component to borrow from another team's code base, since it can be separated out into its own module. Nonetheless, we decided to create our own walk in the hopes of ending up with something at least as good, if not better, than that of other teams, while retaining the ability to fine tune it on our own.

The movement component of our team can be separated into two parts. First, the walking motion itself, and second, an interface module between the low level control of the joints (including both walking and kicking) and the decision-making components.

5.1 Walking

This section details our approach to enabling the Aibos to walk.

5.1.1 Basics

At the lowest level, walking is effected on the Aibo by controlling the joint angles of the legs. Each of the four legs has three joints known as the rotator, abductor, and knee. The rotator is a shoulder joint that rotates the entire leg (including the other two joints) around an axis that runs horizontally from left to right. The abductor is the shoulder joint responsible for rotating the leg out from the body. Finally, the knee allows the lower link of the leg to bend forwards or backwards, although the knees on the front legs primarily bend the feet forwards while the ones on the back legs bend primarily backwards. These rotations will be described more precisely in the section on forward kinematics.

Each joint is controlled by a PID mechanism. This mechanism takes as its inputs P, I, and D gain settings for that joint and a desired angle for it. An online tutorial about PID control can be found at [11]. The robot architecture can process a request for each of the joints at a rate of at most once every eight milliseconds. We have always requested joint values at this maximum allowed frequency. Also, the Aibo model information lists recommended settings for the P, I, and D gains for each joint. We have not thoroughly experimented with any settings aside from the recommended ones and use only the recommended ones for everything that is reported here.

The problem of compelling the robot to walk is greatly simplified by a technique called inverse kinematics. This technique allows the trajectory of a leg to be specified in terms of a three-dimensional trajectory for the foot. The inverse kinematics converts the location of the foot into the corresponding settings for the three joint angles. A precursor to deriving inverse kinematics formulas is having a model of the forward kinematics, the function that takes the three joint angles to a three-dimensional foot position. This is effectively our mathematical model of the leg.

5.1.2 Forward Kinematics

For each leg, we define a three-dimensional coordinate system whose origin is that leg's shoulder. In these coordinate systems, positive x is to the robot's right, positive y is the forward direction, and positive z is up. Thus, when a positive angle is requested from a certain type of joint, the direction of the resulting rotation may vary from leg to leg. For example, a positive angle for the abductor of a right leg rotates the leg out from the body to the right, while a positive angle for a left leg rotates the leg out to the left. We will describe the forward and inverse kinematics for the front right leg, but because of the symmetry of the Aibo, the inverse kinematics formulas for the other legs can be attained simply by first negating x or y as necessary.

The unit of distance in our coordinate system is the length of one link of any leg, i.e. from the shoulder to the knee, or from the knee to the foot. This may seem a strange statement, given that, physically speaking, the different links of the robot's legs are not exactly the same length. However, in our mathematical model of the robot, the links are all the same length. This serves to simplify our calculations, although it is admittedly an inaccuracy in our model. We argue that this inaccuracy is overshadowed by the fact that we are not

modeling the leg's foot, a cumbersome unactuated aesthetic appendage. As far as we know, no team has yet tried to model the foot.

We call the rotator, abductor, and knee angles J_1, J_2, and J_3 respectively. The goal of the forward kinematics is to define the function from $J = (J_1, J_2, J_3)$ to $p = (x, y, z)$, where p is the location of the foot according to our model. We call this function $K_F(J)$. We start with the fact that when $J = (0, 0, 0)$ $K_F(J) = (0, 0, -2)$, which we call p_0. This corresponds to the situation where the leg is extended straight down. In this base position for the leg, the knee is at the point $(0, 0, -1)$. We will describe the final location of the foot as the result of a series of three rotations being applied to this base position, one for each joint.

First, we associate each joint with the rotation it performs when the leg is in the base position. The rotation associated with the knee, $K(q, \Theta)$, where q is any point in space, is a rotation around the line $y = 0$, $z = -1$, counterclockwise through an angle of Θ with the x-axis pointing towards you. The abductor's rotation, $A(q, \Theta)$, goes clockwise around the y-axis. Finally, the rotator is $R(q, \Theta)$, and it rotates counterclockwise around the x-axis. In general (i.e. when J_1 and $J2$ are not 0), changes in J_2 or J_3 do not affect p by performing the corresponding rotation A or K on it. However, these rotations are very useful because the forward kinematics function can be defined as

$$K_F(J) = R(A(K(p_0, J_3), J_2), J_1). \tag{8}$$

This formulation is based on the idea that for any set of angles J, the foot can be moved from p_0 to its final position by rotating the knee, abductor, and rotator by J_3, J_2, and J_1 respectively, *in that order*. This formulation works because when the rotations are done in that order they are always the rotations K, A, and R. A schematic diagram of the Aibo after each of the first two rotations is shown in Figure 3.

It is never necessary for the robot to calculate x, y, and z from the joint angles, so the above equation need not be implemented on the Aibo. However, it is the starting point for the derivation of the Inverse Kinematics, which are constantly being computed while the Aibo is walking.

5.1.3 Inverse Kinematics

Inverse kinematics is the problem of finding the inverse of the forward kinematics function K_F, $K_I(q)$. With our model of the leg as described above, the derivation of K_I can be done by a relatively simple combination of geometric analysis and variable elimination.

The angle J_3 can be determined as follows. First we calculate d, the distance from the shoulder to the foot, which is given by

$$d = \sqrt{x^2 + y^2 + z^2}. \tag{9}$$

Next, note that the shoulder, knee, and foot are the vertices of an isosceles triangle with sides of length 1, 1, and d with central angle $180 - J3$. This yields the formula

$$J_3 = 2\cos^{-1}\left(\frac{d}{2}\right). \tag{10}$$

The inverse cosine here may have two possible values within the range for J_3. In this case we always choose the positive one. While there are some points in three-dimensional space that this excludes (because of the joint ranges for the other joints), those points are not needed for walking. Furthermore, if we allowed J_3 to sometimes be negative, it would be very difficult for our function K_I to be continuous over its entire domain.

To compute J_2, we must first write out an expression for $K(p_0, J_3)$. It is $(0, \sin J_3, 1 + \cos J_3)$. This is the position of the foot in Figure 3a. Then we can isolate the effect of J_2 as follows. Since the rotation R is with respect to the x-axis, it does not affect the x-coordinate. Thus we can make use of the fact that the $K_F(J)$, which is defined to be $R(A(K(p_0, J_3), J_2), J_1)$ (Equation 8), has the same x-coordinate as $A(K(p_0, J_3), J_2)$. Plugging in our expression for $K(p_0, J_3)$, we get that

$$A(K(p_0, J_3), J_2) = A((0, \sin J_3, 1 + \cos J_3), J_2). \tag{11}$$

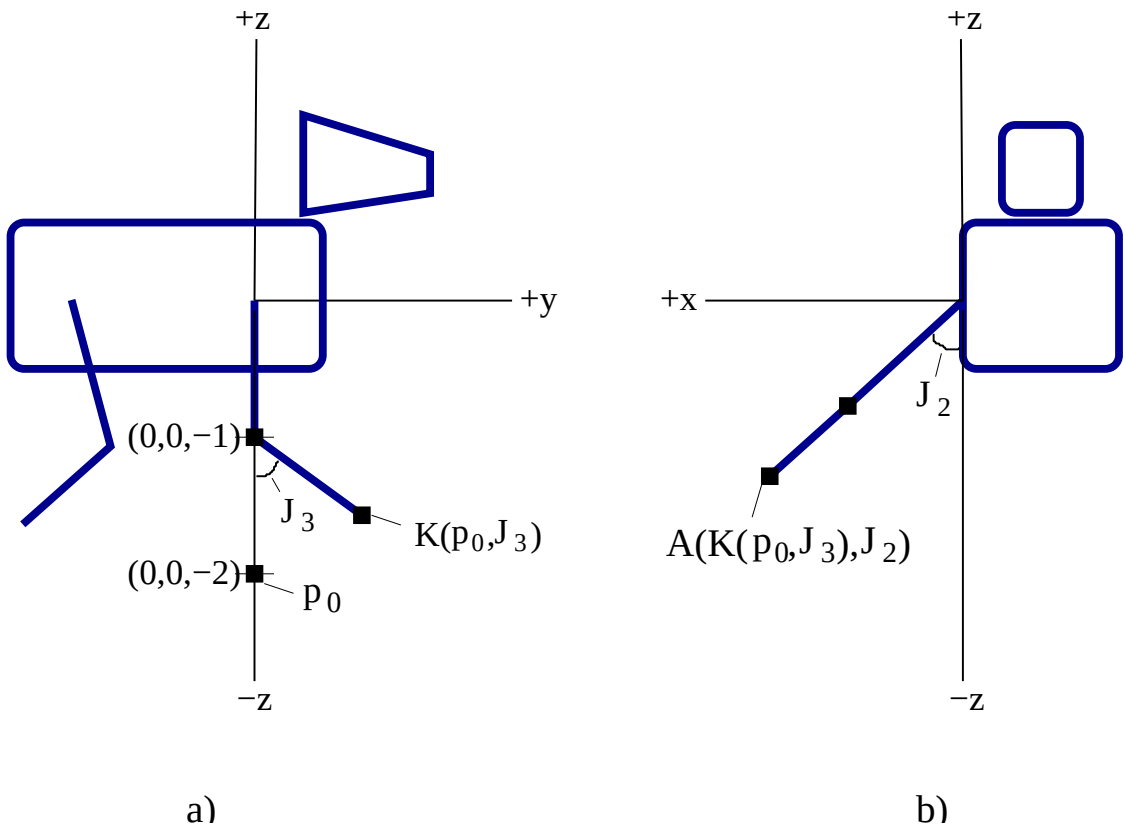


Figure 3: Schematic drawings of the Aibo according to our kinematics model. a) This is a side view of the Aibo after rotation K has been performed on the foot. b) In this front view, rotation A has also been performed.

Since A is a rotation around the y-axis,

$$A(K(p_0, J_3), J_2) = (\sin J_2(1 + \cos J_3), \sin J_3, \cos J_2(1 + \cos J_3)). \tag{12}$$

Setting x (which is defined to be the x-coordinate of $K_F(J)$) equal to the x-coordinate here and solving for J_2 gives us

$$J_2 = \sin^{-1}\left(\frac{x}{1 + \cos J_3}\right). \tag{13}$$

Note that this is only possible if $x \leq 1 + \cos(J_3)$. Otherwise, there is no J_2 that satisfies our constraint for it, and, in turn, no J such that $F_K(J) = q$. This is the *impossible sphere* problem, which we discuss in more detail below. The position of the foot after rotations K and A is depicted in Figure 3b.

Finally, we can calculate J_1. Since we know y and z before and after the rotation R, we can use the difference between the angles in the y-z plane of the two (y, z)'s. The C++ function atan2(z, y) gives us the angle of the point (y, z), so we can compute

$$J_1 = \text{atan2}(z, y) - \text{atan2}(\cos J_2(1 + \cos J_3), \sin J_3). \tag{14}$$

The result of this subtraction is normalized to be within the range for J_1. This concludes the derivation of J_1 through J_3 from x, y, and z. The computation itself consists simply of the calculations in the four equations (9), (10), (13), and (14).

It is worth noting that expressions for J_1, J_2, and J_3 are never given explicitly in terms of x, y, and z. Such expressions would be very convoluted, and they are unnecessary because the serial computation given here can be used instead. Furthermore, we feel that this method yields some insight into the relationships between the legs joint angles and the foot's three-dimensional coordinates.

There are many points q, in three-dimensional space, for which there are no joint angles J such that $F_K(J) = q$. For these points, the inverse kinematics formulas are not applicable. One category of such points is intuitively clear: the points whose distance from the origin is greater than two. These are impossible locations for the foot because the leg is not long enough to reach them from the shoulder. There are also many regions of space that are excluded by the angle ranges of the three joints. However, there is one unintuitive, but important, unreachable region, which we call the impossible sphere. The impossible sphere has a radius of 1 and is centered at the point $(1, 0, 0)$. The following analysis explains why it is impossible for the foot to be in the interior of this sphere.

Consider a point (x, y, z) in the interior of the illegal sphere. This means that

$$\begin{aligned} (x-1)^2 + y^2 + z^2 &< 1 \\ x^2 - 2x + 1 + y^2 + z^2 &< 1 \\ x^2 + y^2 + z^2 &< 2x. \end{aligned}$$

Substituting d for $\sqrt{x^2 + y^2 + z^2}$ and dividing by two gives us

$$\frac{d^2}{2} < x. \tag{15}$$

Since $J_3 = 2\cos^{-1}\left(\frac{d}{2}\right)$ (Equation (10)), $\cos\frac{J_3}{2} = \frac{d}{2}$, so by the double angle formula $\cos J_3 = \frac{d^2}{2} - 1$, or $\frac{d^2}{2} = 1 + \cos J_3$. Substituting for $\frac{d^2}{2}$, we get

$$x > 1 + \cos J_3. \tag{16}$$

This is precisely the condition, as discussed above, under which the calculation of J_2 breaks down. This shows that points in the illegal sphere are not in the range of F_K.

Occasionally, our parameterized walking algorithm requests a position for the foot that is inside the impossible sphere. When this happens, we project the point outward from the center of the sphere onto its

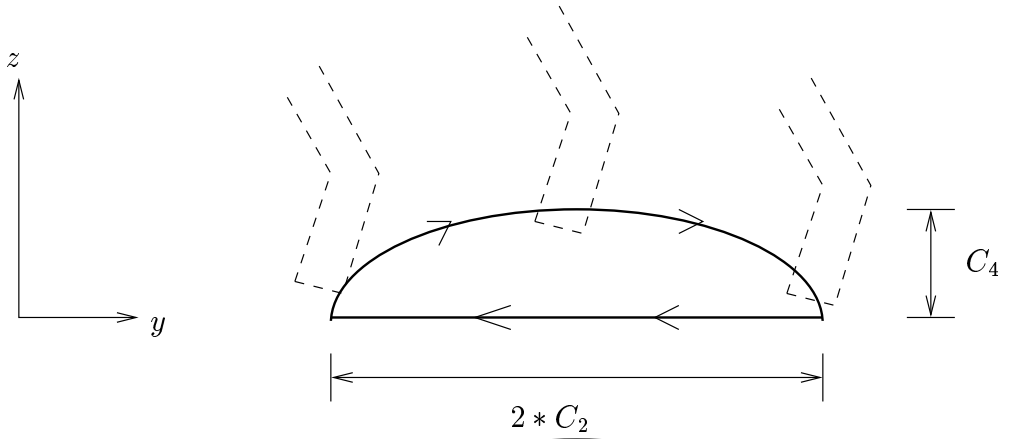


Figure 4: The foot traces a half ellipse as the robot walks forward.

surface. The new point on the surface of the sphere is attainable, so the inverse kinematics formulas are applied to this point.

5.1.4 General Walking Structure

Our walk uses a trot-like gait in which diagonally opposite legs step together. That is, first one pair of diagonally opposite legs steps forward while the other pair is stationary on the ground. Then the pairs reverse roles so that the first pair of legs is planted while the other one steps forward. As the Aibo walks forward, the two pairs of diagonally opposite legs continue to alternate between being on the ground and being in the air. For a brief period of time at the start of our developmental process, we explored the possibility of other gait patterns, such as a walking gait where the legs step one at a time. We settled on the trot gait after watching video of RoboCup teams from previous years.

While the Aibo is walking forwards, if two feet are to be stationary on the ground, that means that they have to move backwards with respect to the Aibo. In order for the Aibo's body to move forwards in a straight line, each foot should move backwards in a straight line for this portion of its trajectory. For the remainder of its trajectory, the foot must move forward in a curve through the air. We opted to use a half ellipse for the shape of this curve (Figure 4).

A foot's half-elliptical path through the air is governed by two functions, $y(t)$ and $z(t)$, where t is the amount of time that the foot has been in the air so far divided by the total time the foot spends in the air (so that t runs from 0 to 1). While the Aibo is walking forwards, the value of x for any given leg is always constant. The values of y and z are given by

$$y(t) = C_1 - C_2 \cos(\pi t) \tag{17}$$

and

$$z(t) = C_3 - C_4 \sin(\pi t). \tag{18}$$

In these equations, C_1 through C_4 are four parameters that are fixed during the walk. C_1 determines how far forward the foot is and C_3 determines how close the shoulder is to the ground. The parameters C_2 and C_4 determine how big a step is and how high the foot is raised for each step (Figure 4). Our walk has many other free parameters, which are all described in Section 5.1.7.

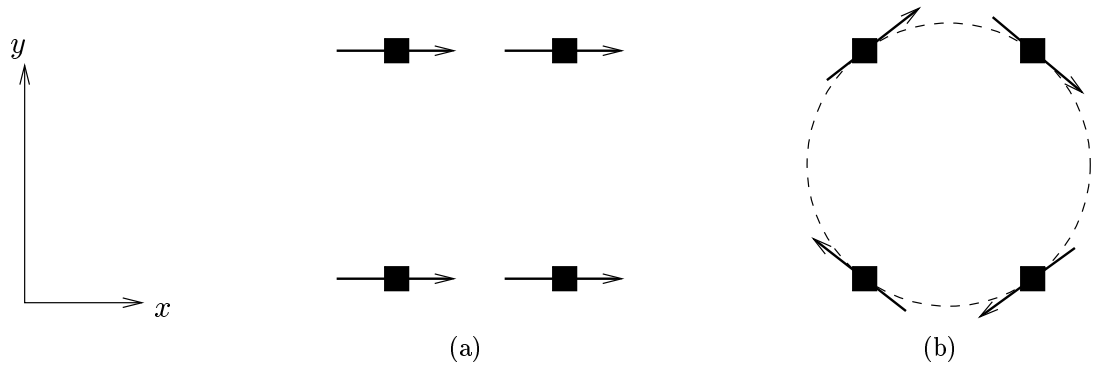


Figure 5: The main movement direction of the half ellipses changes for (a) walking sideways, (b) turning in place. (The dark squares indicate the positions of the four feet when standing still.)

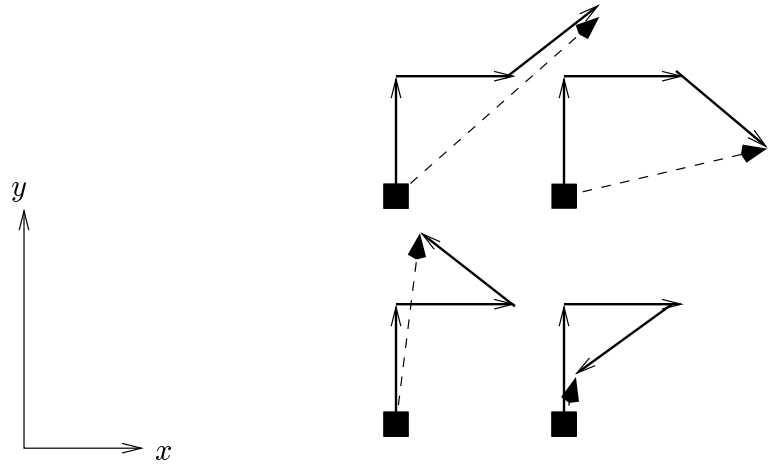


Figure 6: Combining forwards, sideways and turning motions. Each component contributes a vector to the combination. Dashed lines show the resulting vectors. (We show only half of the ellipse lengths, for clarity.) With the vectors shown, the robot will be turning towards its right as it moves diagonally forward and right.

5.1.5 Omnidirectional Control

After implementing the forward walk, we needed sideways, backwards, and turning motions. There is a nice description of how to obtain all these (and any combination of these types of walks) in [12]. We based our implementation on the ideas from that paper.

Sideways and backwards walks are just like the forward walk with the ellipse rotated around the z axis (Figure 5a). For walking sideways, the ellipse is rotated 90° to the side towards which the robot should walk. For walking backwards, the ellipse points in the negative y direction. Turning in place is a little more complicated. The four legs of the robot define a circle passing through them. The direction of the ellipse for each leg is tangent to this circle, pointing clockwise if the robot is to turn right and counterclockwise to turn left (Figure 5b).

Combinations of walking forwards, backwards, sideways, and turning are also possible by simply combining the different components for the ellipses through vector addition. For example, to walk forwards and to the right at the same time, at an angle of 45° to the y axis, we would make the ellipses point 45° to the right of the y axis. Any combination can be achieved as shown in Figure 6.

In practice, the method described here worked well for combinations of forwards and turning velocities, but we had difficulty also incorporating sideways velocities. The problem was that, after tuning the param-

eters (Section 5.1.8), we found that the parameters that worked well for going forwards and turning did not work well for walking sideways. It was not obvious how to find common parameters that would work for combinations of all three types of velocities.

In situations where we needed to walk with a non-zero sideways velocity, we frequently used a slower omnidirectional walk developed by a student in the Spring semester class.[6] That walk is called SPLINE_WALK, while the one being described here is called PARAM_WALK. Section 5.2.3 discusses when each of the walks was used.

5.1.6 Tilting the Body Forward

Up until the American Open, our walking module was restricted to having the Aibo's body be parallel to the ground. That is, it did not allow for the front and back shoulders to be different distances from the ground. This turned out to be a severe limitation. During this time, we were unable to achieve a forward speed of over 150 mm/s. After relaxing this constraint, only the slightest hand tuning was necessary to bring our speed over 200 mm/s. After a significant amount of hand tuning, we were able to achieve a forwards walking speed of 235 mm/s. (The parameters that achieve this speed are given in Section 5.1.8 and our procedure for measuring walking speed is described in Section 5.1.9.)

In many of the fastest and most stable walks the front legs touch the ground with their elbows when they step. Apparently, this is far more effective than just having the feet touch the ground. We enable the elbows to touch the ground by setting the height of the front shoulders to be lower than that of the back shoulders. However, this ability requires one more computation to be performed on the foot coordinates before the inverse kinematics equations are applied. That is, when the Aibo's body is tilted forward we still want the feet to move in half ellipses that run parallel to the ground. This means that the points given by equations 17 and 18 have to be rotated with respect to the x-axis before the inverse kinematics equations are applied.

The angle through which these points must be rotated is determined by the difference between the desired heights of the front and back shoulders and the distance between the front and back shoulders. The difference between the heights, d_h, is a function of the parameters being used (the heights of the front and back shoulders are two of our parameters), but the distance between the front and back shoulders is a fixed body length distance which we estimate at 1.64 in our units and call l_b. Then the angle of the body rotation is given by

$$\theta = \sin^{-1}\left(\frac{d_h}{l_b}\right). \tag{19}$$

5.1.7 Description of all the Parameters

This section lists and describes all twenty parameters of our Aibo walk. The units for most of the parameters are distances which are in terms of leg-link length, as discussed in Section 5.1.2. Exceptions are noted below.

- *Forward step distance*: How far forward the foot should move from its home position in one step.
- *Side step distance*: How far sideways the foot should move from its home position in one step.
- *Turn step distance*: How far each half step should be for turning.
- *Front shoulder height*: How high from the ground the robot's front legs' J1 and J2 joints should be.
- *Back shoulder height*: How high from the ground the robot's back legs' J1 and J2 joints should be.
- *Ground fraction*: What fraction of a step time the robot's foot is on the ground. (The rest of the time is spent with the foot in the air, making a half ellipse.) Between 0 and 1. Has no unit.

[6] Aniket Murarka

- *Front left y-offset*: How far out in the y-direction the robot's front left leg should be when it's in its home position.
- *Front right y-offset*: How far out in the y-direction the robot's front right leg should be when it's in its home position.
- *Back left y-offset*: How far out in the y-direction the robot's back left leg should be when it's in its home position.
- *Back right y-offset*: How far out in the y-direction the robot's back right leg should be when it's in its home position.
- *Front left x-offset*: How far out in the x-direction the robot's front left leg should be when it's in its home position.
- *Front right x-offset*: How far out in the x-direction the robot's front right leg should be when it's in its home position.
- *Back left x-offset*: How far out in the x-direction the robot's back left leg should be when it's in its home position.
- *Back right x-offset*: How far out in the x-direction the robot's back right leg should be when it's in its home position.
- *Front Clearance*: How far up the front legs should be lifted off the ground at the peak point of the half ellipse.
- *Back Clearance*: How far up the back legs should be lifted off the ground at the peak point of the half ellipse.
- *Direction_fwd*: Whether the robot should move forwards or backwards. Either 1 or -1. Has no unit.
- *Direction_side*: Whether the robot should move right or left. Either 1 or -1. Has no unit.
- *Direction_turn*: Whether the robot should turn towards its right or its left. Either 1 or -1. Has no unit.
- *Moving_max_counter*: Number of Open-R frames one step takes. Greater than 1. Has no unit.

5.1.8 Tuning the Parameters

Once the general framework of our walk was set up, we were faced with the problem of determining good values for all of the parameters of the walk. This process was greatly facilitated by the use of a tool we had written that allowed us to telnet into the Aibo and change walking parameters at run time. Thus we were able to go back and forth between altering parameters and watching (or timing) the Aibo to see how fast it was. This process enabled us to experiment with many different combinations of parameters.

We focused most of our tuning effort on finding as fast a straight forward walk as possible. Our tuning process consisted of a mixture of manual hill-climbing and using our observations of the walk and intuition about the effects of the parameters. For example, two parameters that were tuned by relatively blind hill-climbing were *Forward step distance* and *Moving_max_counter*. These parameters are very important and it is often difficult to know intuitively if they should be increased or decreased. So tuning proceeded slowly and with many trials. On the other hand, parameters such as the front and back clearances could frequently be tuned by noticing, for instance, that the front (or back) legs dragged along the ground (or went too high in the air). The fastest parameters we were able to find for our forward walk are given in the following table.

We found that these parameters worked well for combinations of forward and turning velocities (with the appropriate modifications to *Forward step distance* and *Turn step distance*). However, when we set the

Parameter	Value
Forward step distance	0.74
Side step distance	0.0
Turn step distance	0.0
Front shoulder height	1.1
Back shoulder height	1.6
Ground fraction	0.5
Front left y-offset	0.7
Front right y-offset	0.7
Back left y-offset	-0.4
Back right y-offset	-0.4
Front left x-offset	-0.25
Front right x-offset	0.25
Back left x-offset	0.0
Back right x-offset	0.0
Front clearance	0.9
Direction_fwd	1
Direction_side	1
Direction_turn	1
Moving_max_counter	92

Table 3: Fast Walking Parameter Values

forwards and turning components to zero and tried to walk straight sideways, the robot would curve quite sharply forwards. Thus to walk with a non-zero sideways velocity we used either a different set of parameters or SPLINE_WALK.

5.1.9 Odometry Calibration

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 8) and it is important for them to be as accurate as possible. The Movement Module takes a walking request in the form of a set of forward, horizontal, and angular velocities. These velocities are then converted to walking parameters. The Brain assumes that the velocities being requested are the ones that are actually attained, so the accuracy of the odometry relies on that of those conversions.

Since the step distance parameters are proportional to the distance traveled each step and the time for each step is the same, the step distance parameters should theoretically be proportional to the corresponding velocities. This turned out to be true to a fair degree of accuracy for combinations of forward and turning velocities. As mentioned above, we needed to use a different set of parameters for walking with a non-zero sideways velocity. These parameters did not allow for a fast forward walk, but with them the velocities were roughly proportional to the step distances for combinations of forward, turning, and sideways velocities.

The proportionality constants are determined by a direct measurement of the relevant velocities. To measure forward velocity, we use a stopwatch to time the robot walking from one goal line to the other with its forward walking parameters. The time taken is divided into the length of the field, 4200 mm, to yield the forward velocity. The same process is used to measure sideways velocity. To measure angular velocity, we execute the walk with turning parameters. Then we measure how much time it takes to make a certain number of complete revolutions. This yields a velocity in degrees per second. Finally, the proportionality constants were calculated by dividing the measured velocities by the corresponding step distance parameters that gave rise to them.

Since the odometry estimates are used by localization (Section 8), the odometry calibration constants

could be tuned more precisely by running localization with a given set of odometry constants and observing the effects of the odometry on the localization estimates. Then we could adjust the odometry constants in the appropriate direction to make localization more accurate. We feel that we were able to achieve quite accurate odometry estimates by a repetition of this process.

5.2 General Movement

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 10) 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 in the behavior module itself (Section 12), where the decisions to initiate or continue entire types of movement are made.

5.2.1 Movement Module

The movement module shares three connections ("services") with other Open-R objects: one with the OVirtualRobotComm object mentioned above, and two with the *Brain*, the Open-R object which includes most of our code (see Section 10 for a description of our general architecture), including the C++ object corresponding to the movement interface described in Section 5.2.2. It uses one connection with the Brain to take requests from the Brain for types of high-level movement, such as walking in a particular direction or kicking. It then converts them to joint values, and uses its connection with OVirtualRobotComm to request that joint positions be set accordingly. These requests are sent as often as is allowed – every 8 milliseconds. The second connection with the Brain allows the movement module to send updates to the Brain about what movement it is currently performing. Among other things, this lets the Brain know when a movement it requested has finished (such as a kick). The flow of control is illustrated by the arrows in Figure 7 (the functions identified in the figure are defined below). Thick arrows represent a message containing information (from Subject to Observer); thin arrows indicate a message without further information (from Observer to Subject). An arrow ending in a null marker indicates that the message does nothing but enable the service to send another message.

Because the movement module must send an Open-R message to OVirtualRobotComm every time it wants to change a joint position, it is necessary for the movement module to keep an internal state so that it can resume where it left off when OVirtualRobotComm returns control to the movement module. Whenever this happens, the movement module begins execution with the function `ReadyEffector`, which is called automatically every time OVirtualRobotComm is ready for a new command. `ReadyEffector` calls the particular function corresponding to the current movement module state, a variable that indicates which type of movement is currently in progress. Many movements (for example, walking and kicking) require that a sequence of sets of joint positions be carried out, so the functions responsible for these movements must be executed multiple times (for multiple messages to OVirtualRobotComm). The states of the movement module are summarized in Table 4.

Whereas kicking and getting up require the Aibo's head to be doing something specific, neither the idle state nor the two walks require anything in particular from the head joints. Furthermore, it is useful to allow the head to move independently from the legs whenever possible (this allows the Aibo to "keep its eye on the ball" while walking, for instance). Thus the movement module also maintains a separate internal state for the head. If the movement module's state is KICK_MOTION or GETUP_MOTION when `ReadyEffector` begins execution, the new joint angles for the head will be specified by the function corresponding to the

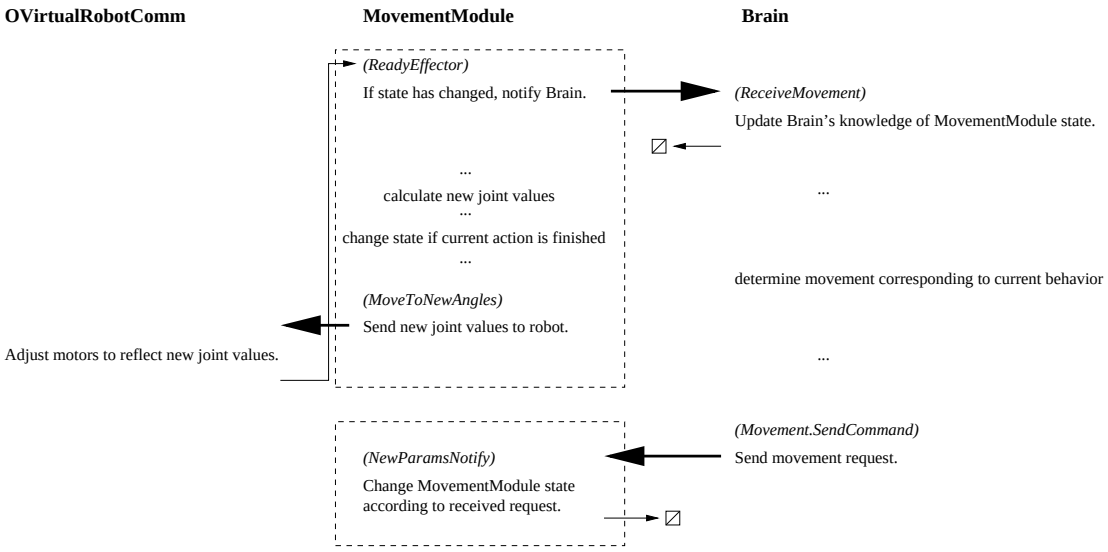


Figure 7: Inter-object communication involving the movement module. Thick arrows represent a message containing information (from Subject to Observer); thin arrows indicate a message without further information (from Observer to Subject). An arrow ending in a null marker indicates that the message does nothing but enable the service to send another message.

State	Description
INIT	Initial state
IDLE	No leg motion, but joint gains are set (robot is standing)[7]
PARAM_WALK	Fastest walk
SPLINE_WALK	Omnidirectional slower walk
KICK_MOTION	Kicking
GETUP_MOTION	No joint position requests being sent to OVirtualRobotComm, thus allowing built-in Sony getup routines control over all motors

Table 4: Movement module states

movement module state. Otherwise, `ReadyEffector` calls a function corresponding to the current head state, which determines the new joint angles for the head, and the rest of the joint angles are determined by the function for the current movement module state. A summary of the head states appears in Table 5.

The movement module listens for commands with a function called `NewParamsNotify`. When the Brain sends a movement request, `NewParamsNotify` accepts it and sets the movement module state and/or head state accordingly. When the internal state is next examined – this occurs in the next call to `ReadyEffector` (that is, after the next time the joint positions are set by OVirtualRobotComm) – the movement module begins executing the requested movement. See Table 6 for a summary of the possible requests to the movement module. Note that both a head movement and a body movement may be requested simultaneously, with the same message. However, if the body movement that is requested needs control of the head joints, the head request is ignored.

[7] In practice, this is implemented by executing a "walk" with forward velocity, side velocity, turn velocity, and leg height all equal to 0.

State	Description
IDLE	Head is still (but joint gains are set)
MOVE	Moving head to a specific position
SCAN	Moving head at a constant speed in one direction
KICK	Executing a sequence of head positions

Table 5: Head states

Type of request	Explanation	Associated parameters
MOVE_NOOP	don't change body movement	
MOVE_STOP	stop leg movement	
MOVE_PARAM_WALK	start walking using ParamWalk	x-velocity, y-velocity, angular velocity
MOVE_SPLINE_WALK	start walking using SplineWalk	x-destination, y-destination, angular destination
MOVE_KICK	execute a kick	type of kick
MOVE_GETUP	get up from a fall	
DONE_GETUP	robot is now upright, resume motions	
HEAD_NOOP	don't change head movement	
HEAD_MOVE	move head to a specific angle	
HEAD_SCAN	scan head at constant velocity	scan speed, direction
HEAD_KICK	kick with the head	type of kick
HEAD_STOP	stop head movement	

Table 6: Possible requests to the movement module

5.2.2 Movement Interface

The movement interface is part of the Brain Open-R object. Its main function is to translate high-level movement commands into movement module requests, so that the Brain can simply specify high-level movement behaviors (such as "turn toward this angle and kick with this kick") and let the movement interface take care of the rest.

During each Brain cycle, the behavior modules specify movements by calling movement interface functions, which compute the combination of movement module requests necessary to carry out the specified movement. If the requested types of movement do not interfere with each other (for example, if both a head scan and a forward walk are requested in the same Brain cycle), then all requested movements are combined in the message that is eventually sent to the movement module. Finally, at the end of each Brain cycle, the function `Movement.SendCommand` is called. This function takes care of sending the message to the movement module containing the request, and ensuring that redundant messages are not sent.

The movement interface provides functions for basic movements such as walking forward, turning, moving the head to a position, stopping the legs or head, and getting up from a fall. It also provides several functions for more complex movements, which are described here.

Head Scan When searching for the ball, it is helpful to move the head around in some fashion so that more of the field can be seen. On the one hand, the more quickly the field can be covered by the scan, the more quickly the ball can be found. On the other hand, if the head moves too quickly, the vision will not be able to recognize the ball, because it will not be in sight for the required number of frames. Therefore it makes sense to try to cover as much of the field with as little head movement as possible. At first we believed that it was not possible to cover the entire height of the field with fewer than three horizontal scans, so we used a three-layer head scan at the American Open. However, by watching other teams, we became convinced that it must be possible to cover the entire relevant portion of the field with two head scans. After some experimentation, we managed to eliminate the persistent blind spot in the middle of a two-layer head

scan that we created. Thus, the movement interface now provides a function that takes care of executing the two-layer head scan. It also allows the behaviors to specify which corner the scan starts from. This is because the two-layer head scan typically occurs immediately after losing the ball, and often the brain knows which direction the ball is most likely to be in given where it was last seen. Thus allowing the starting corner to be specified allows this information to be used.

Follow Object Once the robot sees the ball, walking towards it is achieved by two simultaneous control laws. The first keeps the head pointed directly at the ball as the ball moves in the image. This is achieved by taking the horizontal and vertical distances between the location of the ball in the image and the center of the image and converting them into changes in the head pan and tilt angles.

Second, the Aibo walks towards the direction that its head is pointing. It does this by walking with a combination of forward and turning velocities. As the head's pan angle changes from the straight ahead position towards a sidewise-facing position, the forward velocity decreases linearly (from its maximum) and the turning velocity increases linearly (from zero). In combination, these policies bring the Aibo towards the ball.

While we were able to use the above methods to have the Aibo walk in the general direction of the ball, it proved quite difficult to have the Aibo reliably attain control of the ball. One problem was that the robot would knock the ball away with its legs as it approached the ball. We found that if we increased the proportionality constant of the turning velocity, it would allow the robot to face the ball more precisely as it went up to the ball. Then the ball would end up between the Aibo's front legs instead of getting knocked away by one of them. Another problem that arose was that the Aibo occasionally bumped the ball out of the way with its head. We dealt with this by having the robot keep its head pointed 10° above the ball. Both of these solutions required some experimentation and tuning of parameters.

Track Object This function follows a ball with the head, and turns the body in place when necessary so as not to lose sight of the ball. It is used chiefly for the goalie.

Strafe Before we had localization in place, we needed a way to turn the robot around the ball so that it could kick it towards the goal. The problem was that we needed to keep its head pointing down the field so it could see the goal, which made turning with the ball pinched underneath the chin (see below) unfeasible. Strafing consisted of walking with a sideways velocity and a turning velocity, but no forward velocity. This caused the Aibo to walk sideways in a circle around the ball. During this time, it was able to keep its head pointed straight ahead so that it could stop when it saw the goal.

Chin Pinch Turn This is a motion which lowers the head (to a tilt angle of -55°) to trap the ball below the chin, and then turns some number of degrees while the ball is trapped there. Once we had localization in place, this replaced the strafe function just described, because it is both faster and more reliable at not losing the ball.

Tuck Ball Under This function walks forward slowly while pulling the head down. It helps the Aibo attain control of the ball, and is typically used for the transition between follow object and chin pinch turn.

5.2.3 High-Level Control

For the most part, it is the task of the behaviors to simply choose which combinations of the movement interface functions just described should be executed. However, there are exceptions; sometimes there is a reason to handle some details of movement at the level of the behavior. One such exception is establishing the duration of the chin pinch turn. Because localization is used to determine when to stop the chin pinch turn, it makes more sense to deal with this in the behavior than in the movement interface, which does not otherwise need to get localization information.

If the behavior chooses to do a chin pinch turn (see Section 12.1.2 for details on when this happens), it will specify an Aibo-relative angle that it wishes to turn toward as well as which way to turn (by the sign of the angle). This angle is then converted to an angle relative to the robot's heading to the offensive goal.[8] The robot continues to turn[9] until the robot's heading to the opponent goal is as desired, and then the behavior transitions to the kicking state.

While we use PARAM_WALK for the vast majority of our walking, we use SPLINE_WALK in most cases where we need to walk with a non-zero sideways velocity. An important example of this is in the supporter role (Section 13.2.1), where we need to walk to a point while facing a certain direction. SPLINE_WALK was also used for part of the obstacle avoidance challenge task. In general, we decided which walk to use in any particular situation by trying both and seeing which one was more effective.

6 Fall Detection

Sony provides routines that enable the robot to detect when it's fallen and that enable it to get up. Our initial approach was to simply use these routines. However, as our walk evolved, the angle of the Aibo's trunk while walking became steeper. This, combined with variations between robots, caused several of our robots to think they were falling over every few steps and to try repeatedly to get up. To remedy this, we implemented a simple fall detection system of our own.

The fall detection system functions by noting the robot's x- and y-accelerometer sensor values each Brain cycle. If the absolute value of an accelerometer reading is greater than some constant (we used $6,800,000$) for a number (5) of consecutive cycles, a fall is registered.

It is also possible to turn fall detection off for some period of time. Many of our kicks require the Aibo to pass through a state which would normally register as a fall, so fall detection is disabled while the Aibo is kicking. If the Aibo falls during a kick, the fall detection system registers the fall when the kick is finished, and the Aibo then gets up.

7 Kicking

The robot's kick is specified by a sequence of poses. A $Pose = (j_1, \ldots, j_n)$, $j_i \in \Re$, where j represents the positions of the n joints of the robot. The robot uses its PID mechanism to move joints 1 through n from one $Pose$ to another over a time interval t. We specify each kick as a series of moves $\{Move_1, \ldots, Move_m\}$ where a $Move = (Pose_i, Pose_f, \Delta t)$ and $Move_{jPose_f} = Move_{(j+1)Pose_i}$, $\forall j \in [1, m-1]$. All of our kicks only used 16 of the robot's joints (leg, head, and mouth). Table 7 depicts the used joints and joint descriptions.

7.1 The Initial Kick

In the beginning stages of our team development, our main focus was on creating modules (Movement, Vision, Localization, etc.) and incorporating them with one another. Development of kicks did not become a high priority until after the other modules had been incorporated. Thus, we created a "first kick" early on to address the needs of the other modules as they developed and created other kicks much later to expand our strategic capabilities.

We decided to model our first kick after what seemed to be the predominant goal-scoring kick from previous RoboCup competitions. During the kick, the robot raises its two front legs up and drops them onto the sides of the ball. The force of the falling legs propels the ball forward. Our first kick, called the "front power kick" tried to achieve this effect.

[8] The choice of heading to the offensive goal as the landmark for determining when the chin pinch turn should stop is due to the fact that the chin pinch turn's destination is often facing the opponent goal, as well as the fact that there was already a convenient GlobalMap interface function that provided heading to the offensive goal. In theory, anything else would work equally well.

[9] That is, the behavior repeatedly sends requests to the movement interface to execute the chin pinch turn.

joint	*joint description*
j_1	front right rotator
j_2	front right abductor
j_3	front right knee
j_4	front left rotator
j_5	front left abductor
j_6	front left knee
j_7	back right rotator
j_8	back right abductor
j_9	back right knee
j_{10}	back left rotator
j_{11}	back left abductor
j_{12}	back left knee
j_{13}	head tilt joint
j_{14}	head pan joint
j_{15}	head roll joint
j_{16}	mouth joint

Table 7: Joints used in kicks

We wanted our front power kick to transition from any walk without prematurely tapping the ball out of the way. Thus, we started the kick in a "broadbase" position in which the robot's torso is on the ground with its legs spread out to the side. If the robot were to transition into the front power kick from a standing position, the robot would drop to the ground while pulling its legs away from the ball. From this broadbase position, the robot then moves its front legs together to center the ball. After the ball has been centered, the robot moves its front legs up above its head and then quickly drops the front legs onto the sides of the ball, kicking the ball forward.

We found that the kick moves the ball relatively straight ahead for a distance of up to 3 meters. However, we noticed that the robot's front legs would miss the ball if the ball were within 3cm of the robot's chest. We resolved this issue by using the robot's mouth to push the ball slightly forward before dropping its legs on the ball.

7.2 A General Kick Framework

We soon realized that we would need to create several different kicks for different purposes. To that end, we started thinking of the kick-generation process in more general terms. In this section we formalize that process.

The kick is an example of a fine-motor control motion where small errors matter. Creation of a kick requires special attention to each *Pose*. A few angles' difference could affect whether the robot makes contact with the ball. Even a small difference in Δt in a *Move* could affect the success of a kick. To make matters more complicated, our team needed the kick to transition from and to a walk. More consideration had to be taken to ensure that neither the walk nor the kick disrupted the operation of the other.

We devised a two-step technique for kick-generation:

1. Creating the kick in isolation from the walk.
2. Integrating the kick into the walk.

7.2.1 Creating the Critical Action

We first created the kick in isolation from the walk. The *Moves* that comprise the kick in isolation constitute the *critical action* of the kick. To obtain the joint angle values for each *Pose*, we used a tool that captured

all the joint angle values of the robot after physically positioning the robot in its desired pose. We first positioned the robot in the *Pose* in which the robot contacts the ball for the kick and recorded $j_1, \ldots, j_n$ for that *Pose*. We called this $Pose_b$.

We then physically positioned the robot in the *Pose* from which we wanted the robot to move to $Pose_b$. We called this $Pose_a$. We then created a *Move* $m = (Pose_a, Pose_b, \Delta t)$ and watched the robot execute m. At this point of kick creation, we were primarily concerned with the path the robot took from $Pose_a$ to $Pose_b$. Thus, we abstracted away the Δt of the *Move* by selecting a large Δt that enabled us to watch the path from $Pose_a$ to $Pose_b$. We typically selected Δt to be 64. Since movement module requests are sent every 8 milliseconds, this *Move* took 64 * 8 milliseconds to execute.

If the *Move* did not travel a path that allowed the robot to kick the ball successfully, we then added an intermediary $Pose_x$ between $Pose_a$ and $Pose_b$ and created a sequence of two *Moves* $\{(Pose_a, Pose_x, \Delta t_i), (Pose_x, Pose_b, \Delta t_{i+1})\}$ and watched the execution. Again, we abstracted away Δt_i and Δt_{i+1}, typically selecting 64. After watching the path for this sequence of *Moves*, we repeated the process if necessary.

After we were finally satisfied with the sequence of *Moves* in the *critical action*, we tuned the Δt for each *Move*. Our goal was to execute each *Move* of the *critical action* as quickly as possible. Thus, we reduced Δt for each *Move* individually, stopping if the next decrement disrupted the kick.

7.2.2 Integrating the Critical Action into the Walk

The second step in creating the finely controlled action involves integrating the *critical action* into the walk. There are two points of integration: (1) the transition from the walk to the *critical action*, (2) the transition from the *critical action* to the walk.

We first focus on the *Move* $i = (Pose_y, Pose_a, \Delta t)$, where $Pose_y \in \{$*all possible poses of the walk*$\}$. Since i precedes the *critical action*, there may be cases in which i adds unwanted momentum to the *critical action* and disrupts it. If i had such cases, we found a $Pose_s$, in which $\{(Pose_y, Pose_s, \Delta t), (Pose_s, Pose_a, \Delta t)\}$ did not lend unwanted momentum to the *critical action*. We call this the *initial action*. The $Pose_s$ we used mirrored the *idle* position of the walk. The *idle* position of the walk is the *Pose* the robot assumes when walking with 0 velocity. We then added the *Move* $(Pose_s, Pose_a, \Delta t)$, abstracting away the Δt, to the moves of the *critical action* and watched the path of execution.

As with the creation of the *critical action*, we then added intermediary *Poses* until we were satisfied with the sequence of *Moves* from $Pose_y$ to $Pose_a$. We then fine-tuned the Δt for the added *Moves*.

Finally, at the end of every kick during game play the robot assumes the *idle* position of the walk, which we call $Pose_z$, before continuing the walk. This transition to $Pose_z$ takes 1 movement cycle. Thus we consider the last *Move* of the kick, f, to be $(Pose_b, Pose_z, 1)$. Since f follows the *critical action*, there may be cases in which f hinders the robot's ability to resume walking.

In such cases, as with the creation of the *critical action* and the *initial action*, we then added intermediary *Poses* until we were satisfied with the sequence of *Moves* from $Pose_b$ to $Pose_z$. We call the *Moves* between the intermediary *Poses* the *final action*. We then fine-tuned the values of Δt used in the *final action*.

The sequence of *Moves* constituting the *initial action*, *critical action*, and *final action* make up the kick.

7.3 Head Kick

After many of our modules had been integrated, the need arose for a kick in a non-forward direction. Inspired by previous RoboCup teams, decided that the head could be used to kick the ball to the left or to the right. During the head kick, the robot first leans in the direction opposite of the direction it intends to kick the ball. The robot then moves its front leg (left leg when kicking left, right leg when kicking right) out of the way. Finally, the robot leans in the direction of the kick as the head turns to kick the ball.

The head kick moves the ball almost due left (or right) a distance of up to 0.5 meters. We discovered that the head kick was especially useful when the ball was close to the edge of the field. The robot could walk to the ball, head kick the ball along the wall, and almost immediately continue walking, whereas the front power kick frequently kicked the ball against the wall, effectively moving the ball very little, if at all.

7.4 Chest Push Kick

The creation of the head kick informed us that the robot could enter and exit a kick much faster when the kick occurred with the robot in a standing position. We thus created the chest push kick in hopes that its execution would be much faster than that of the front power kick. During the chest push kick, the robot quickly leans its chest into the ball. This occurs while the robot remains in a standing position.

To create the kick, we first isolated the kick from the walk. The following table shows the critical action for the chest push kick. In these tables each value of Δt is listed in the row of the *Pose* that ends the corresponding *Move*.

	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	Δt
$Pose_1$	-12	30	91	-12	30	91	-70	45	104	-70	45	104	0	0	0	0	64
$Pose_2$	-120	90	145	-120	90	145	120	25	125	120	25	125	0	0	0	0	1
$Pose_3$	-12	30	91	-12	30	91	-30	6	104	-30	6	104	0	0	0	0	64

Table 8: Chest push kick critical action

We then integrated the walk with the kick. Testing revealed that the robot successfully kicked the ball 55% of the time and fell over after 55% of the successful kicks. Since $(Pose_y, Pose_1, \Delta t)$ added unwanted momentum to the critical action, we created an initial action to precede the critical action. $\{(Pose_y, Pose_s, 64), (Pose_s, Pose_1, 64)\}$ does not lend unwanted momentum to the critical action. Testing revealed that the robot now successfully kicked the ball 100% of the time. The following table shows the initial action with the critical action.

	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	Δt
$Pose_s$	-12	30	91	-12	30	91	-30	6	104	-30	6	104	0	0	0	0	64
$Pose_1$	-12	30	91	-12	30	91	-70	45	104	-70	45	104	0	0	0	0	64
$Pose_2$	-120	90	145	-120	90	145	120	25	125	120	25	125	0	0	0	0	1
$Pose_3$	-12	30	91	-12	30	91	-30	6	104	-30	6	104	0	0	0	0	64

Table 9: Chest push kick initial action and critical action

Since the critical action did not add unwanted momentum that hindered the robot's ability to resume its baseline motion, there was no need to create a final action.

We found that the chest push kick moves the ball relatively straight ahead. It is also very fast. However, the distance the ball travels after the chest push kick is significantly smaller than the distance the ball travels after the front power kick. Thus, we decided against using the chest push kick instead of the front power kick during game play.

7.5 Arms Together Kick

After creating kicks geared toward scoring goals, we realized that we needed a kick for the goalie to block the ball from entering its goal. Deciding that speed and coverage area were more important than the direction of the kick, we created the arms together kick. During the arms together kick, the robot first drops into broadbase position mentioned in Section 7.1. The robot then swings its front left leg inward. After that, the robot swings its front right leg inward as it swings its front left leg back out. The arms together kick proved successful at quickly propelling the ball away from the goal.

7.6 Fall Forward Kick

After attending the American Open, we saw a need for a forward direction kick more powerful than the front power kick. Inspired by a kick used by the CMPack team from Carnegie Mellon, we created the fall

forward kick. The fall forward kick makes use of the forward momentum of the robot as it falls from standing position to lying position. Since the kick begins in a standing position, the robot can quickly transition from the walk to the kick. However, since the kick ends in a lying position, the robot does not transition from the kick back to the walk as quickly.

We first isolated the kick from the walk. The following table shows the critical action.

	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	Δt
$Pose_1$	-5	0	20	-5	0	20	-35	6	75	-35	6	75	45	-90	0	0	32
$Pose_2$	-100	23	0	-100	23	0	100	6	75	100	6	75	45	-90	0	0	32

Table 10: Fall forward kick critical action

We then integrated the walk with the kick. There was no need to create an initial action because any momentum resulting from $(Pose_y, Pose_1, 32)$ was in the forward direction (the same direction we wanted the robot to fall). However, testing revealed that $(Pose_2, Pose_z, \Delta t)$ caused the robot to fall forward on its face every time. Although the robot successfully kicked the ball, the robot could not immediately resume walking. In this situation, the robot had to wait for its fall detection to trigger and tell it to get up before resuming the walk. The get up routine triggered by fall detection was very slow. Thus, we found a $Pose_g$ such that $\{(Pose_2, Pose_g, 32), (Pose_g, Pose_z, \Delta t)\}$ does not hinder the robot's ability to resume walking. The following table shows the critical action with $Move(Pose_2, Pose_g, 32)$.

	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	Δt
$Pose_1$	-5	0	20	-5	0	20	-35	6	75	-35	6	75	45	-90	0	0	32
$Pose_2$	-100	23	0	-100	23	0	100	6	75	100	6	75	45	-90	0	0	32
$Pose_g$	90	90	0	90	90	0	100	6	75	100	6	75	45	-90	0	0	32

Table 11: Fall forward kick critical action and $\{(Pose_2, Pose_g, 32)\}$

From observation, it is noted that transitioning from $Pose_2$ directly to $Pose_g$ is not ideal. The robot would fall over 25% of the time during $(Pose_2, Pose_g, 32)$. Thus, we added $Pose_w$ to precede $Pose_g$ in the final action. Afterwards, the robot no longer fell over when transitioning from the kick to the walk. The following table shows the entire finely controlled action, consisting of the critical action and the final action.

	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}	j_{13}	j_{14}	j_{15}	j_{16}	Δt
$Pose_1$	-5	0	20	-5	0	20	-35	6	75	-35	6	75	45	-90	0	0	32
$Pose_2$	-100	23	0	-100	23	0	100	6	75	100	6	75	45	-90	0	0	32
$Pose_w$	-100	90	0	-100	90	0	100	6	75	100	6	75	45	-90	0	0	32
$Pose_g$	90	90	0	90	90	0	100	6	75	100	6	75	45	-90	0	0	32

Table 12: Fall forward kick critical action and final action

The fall forward kick executed quickly and potentially moved the ball the entire distance of the field (4.2 meters). Unfortunately, the fall forward kick did not reliably propel the ball directly forward. Thus, in game play, we used the fall forward kick in the defensive half of the field and used the front power kick for more reliable goal scoring in the offensive half of the field.

One unexpected side-effect of adding $Pose_g$ to the end of the fall forward kick was that the outstretched legs in $Pose_g$ added additional ball coverage. A ball that the fall forward action missed because it was not located around the robot's chest would actually be propelled forward if the ball was just in front of one of the front legs. Thus, the fall forward kick, which moves the ball away much farther than the arms together kick, also became our primary goalie block.

7.7 Yoshi Kick

Games at the American Open also inspired us to create the yoshi kick. During the yoshi kick, the robot launches its body over the ball and kicks the ball out from behind it. The yoshi kick ideally works well in situations when the robots are crowded together around the ball. However, because the yoshi kick is still somewhat unreliable, the behavior used for RoboCup games only executes a yoshi kick in very specific circumstances, which in practice occur rarely. (See Section 12.1.2 for details.)

8 Localization

Since it requires at least vision and preferably locomotion to already be in place, localization was a relatively late emphasis in our efforts. In fact, it did not truly come into place until after the American Open Competition at the end of April. Before that time, we had been working on a preliminary approach that was eventually discarded and replaced by the current one.

For self-localization, the Austin Villa team implemented a Monte-Carlo localization approach similar to the one used by the German Team [5]. This approach uses a collection of particles to estimate the global position and orientation of the robot. These estimates are updated by visual percepts of fixed landmarks and odometry data from the robot's movement module (see Section 5.1.9). The particles are averaged to find a best guess of the robot's pose.

We have extended the approach of the German Team to improve the accuracy of the observation updates. Rather than using only the most current landmark observations, our approach maintains a history of recent observations that are averaged according to their estimated accuracy. Because it is rare for the robot to gather sufficient information in a single camera frame to triangulate its position, it is important to incorporate visual information from the recent past. At the same time, if visual data is inaccurate, reusing it again and again can aggravate the problem. Our approach is able to leverage past data while, in most situations, robustly tolerating occasional bad inputs.

8.1 Basic Particle Filtering Approach

The goal of the localization module is to calculate a probability distribution over the possible locations and orientations of the robot. Rather than modeling this distribution parametrically, Monte-Carlo localization uses a finite set of samples called *particles*. Each particle can be seen as a hypothesis for the current pose of the robot: $< x, y, \theta >$ where $< x, y >$ is the position of the robot and θ is its orientation in the global coordinate system. Along with a pose hypothesis, each particle is assigned a probability, p, representing the likelihood that the estimate is correct. In our implementation, we used a set of 100 of these particles, which we found experimentally to provide a sufficient level of accuracy without substantially lowering the rate of our main execution cycle.

During each execution cycle of the robot, the localization module updates the set of particles in three steps. The first step is the *motion update* in which the particles are moved based on the physical movement of the robot. The next step is the *observation update* in which the particle probabilities are adjusted for the latest visual information. Finally, *resampling* is done to stochastically move the particles closer to the most likely pose estimate. The following sections describe these updates in detail.

8.2 Motion Update

Based on the currently executing walk or kick, the movement module returns an estimate of the robot's change in position and orientation since the last localization update: $< \delta x, \delta y, \delta\theta >$. This change is added to each particle's pose according to the following equation:

$$pose_{new} = pose_{old} + < \delta x', \delta y', \delta\theta > \tag{20}$$

where $\delta x'$ and $\delta y'$ are δx and δy translated from the coordinate system of the robot into the coordinate system of the particle (see Appendix A.9). Because the odometry information is noisy, we assume that

motion updates decrease the certainty in our pose estimate. For this reason, after each motion update, the probability of each particle is decayed according to the following equation:

$$p_{new} = p_{old}(1 - \epsilon) \tag{21}$$

In our implementation, we chose the value 0.02 for ϵ. This value was chosen, without experimentation, so that the probability would drop by half every couple of seconds.

8.3 Observation Update

After the current frame has been processed by the vision module, the localization module receives a list of landmark observations. For our purposes, the field consists of 10 identifiable fixed landmarks: 6 beacons and 4 goal edges.[10] Each observation consists of a landmark identity (e.g. yellow goal's left edge), a distance estimate, d, a bearing estimate, α, and a probability, $\hat{p}$, representing the certainty that the observed landmark was identified correctly. These observations are used to update the *landmark memory* structure, which is described in the next section.

8.3.1 Landmark Memory

The landmark memory data structure stores a history of recent observations in order to make accurate estimates of the robot's relative position to landmarks. For each of the 10 landmarks, the landmark memory maintains a list containing past observations. Along with the observation itself, each list entry includes the following data:

- σ_d^2: variance of distance estimate
- σ_α^2: variance of bearing estimate
- T: absolute time the observation was made
- Δd: distance robot has moved since this observation
- $\Delta\theta$: total angle the robot has rotated since this observation

An observation is modeled as a 2-d Gaussian distribution with mean $< d, \alpha >$ and variance $< \sigma_d^2, \sigma_\alpha^2 >$. The initial variances are calculated from d and $\hat{p}$ using the following equations:

$$\sigma_d^2 = \frac{d}{\hat{p} * 10} \tag{22}$$

$$\sigma_\alpha^2 = \tan^{-1}\left(\frac{W_b}{d}\right) \tag{23}$$

where W_b is the actual width of a beacon. Because the distance of a beacon is more difficult to estimate as it gets farther away, we made the distance error proportional to d. Also, we made the error inversely proportional to our certainty in landmark identity so that false landmark sightings would generate estimates with high variances. The bearing to a beacon is actually easier to estimate as it gets farther away. For this reason, we made the bearing variance inversely proportional to distance. These error estimates are crude, but we found them to be satisfactory in practice.

When an observation is added to the list, the timestamp T is set to the current time and Δd and $\Delta\theta$ are initialized to 0.

[10] We chose to use the left and right edges of the goals as landmarks, instead of the goals themselves, because the goal edges had more precise locations and were more numerous.

8.3.2 Removing Obsolete Observations

During every motion update (see Section 8.2), the entries in the landmark memory are modified to reflect passing time and robot movement. For each observation entry, the Δd and $\Delta\theta$ values are increased according to the odometry data returned by the movement module. The observation estimates are updated to correspond to the robot's new position and orientation. Also, the variances are increased proportionally to the robot's movement. These updates are summarized by the following equations:

$$\Delta d' = \Delta d + \sqrt{(\delta x)^2 + (\delta y)^2} \tag{24}$$

$$\Delta\theta' = \Delta\theta + |\delta\theta| \tag{25}$$

$$d' = \sqrt{(d\cos(\theta) - \delta x)^2 + (d\sin(\theta) - \delta y)^2} \tag{26}$$

$$\theta' = \text{atan2}\,(d\sin(\theta) - \delta y, d\cos(\theta) - \delta x) - \delta\theta \tag{27}$$

$$\sigma_d^{2\prime} = \sigma_d^2 + \sqrt{(\delta x)^2 + (\delta y)^2} \tag{28}$$

$$\sigma_\alpha^{2\prime} = \sigma_\alpha^2 + \frac{|\delta\theta|}{2} \tag{29}$$

After the observation entries have been updated, we decide if the observation should remain in the list. If the observation has a high variance ($\sigma_d^2 > 500$mm or $\sigma_\alpha^2 > 22^o$), then it is removed from the landmark memory. Additionally, if the robot has traveled too far ($\Delta d > 150$mm) or turned too much ($\Delta\theta > 10^o$) since the observation was made, then the observation is thrown out. Finally, if the observation is too old (time $- T > 3$s) then the entry is deleted. This way, even if the robot is standing still, old observations do not stay around forever. All thresholds were chosen without experimentation.

8.3.3 Merging Past Observations

For each of the 10 fixed landmarks, the landmark memory contains a list of 0 or more relative position estimates. To use this data for updating the particles, we must merge the entries within each list to find a single set of landmark observations.

As stated previously, observations in the landmark memory are modeled as 2-d Gaussians. We chose this distribution because the theory behind merging Gaussian distributions is well-understood. Here, we treat the distance and bearing estimates as independent distributions. Therefore, we can perform a two-dimensional merge by doing two independent one-dimensional merges. To merge two 1-d Gaussian with means and variances (μ_a, σ_a^2) and (μ_b, σ_b^2), respectively, into a new distribution $(\mu_{merged}, \sigma_{merged}^2)$, we use the following equations:

$$\mu_{merged} = \frac{\sigma_a^2\mu_b + \sigma_b^2\mu_a}{\sigma_a^2 + \sigma_b^2} \tag{30}$$

$$\sigma_{merged}^2 = \frac{\sigma_a^2\sigma_b^2}{\sigma_a^2 + \sigma_b^2} \tag{31}$$

These operations are both commutative and associative, so we are free to merge the observations in any order. For each landmark with at least one observation entry, we compute a merged position estimate to be used for updating the particle probabilities.

8.3.4 Updating Probabilities

Using the set of merged estimates from the landmark memory, we update the probability, p_i, of each particle, i, based on the posterior probability of making these observations assuming that the particle is the correct pose hypothesis. Here, we use only the bearing measurement of the observation. The distance information is used at a different stage (see Section 8.3.6).

Given the particle's position and orientation along with information about the positions of all fixed landmarks from an internal map, we can calculate the expected bearing, $\alpha_{expected}$ for each observed landmark.

If the difference between the measured and expected bearings is small, then the particle is likely to be a good estimate of our current position and orientation. If the difference is large, the particle is probably a bad estimate.

The posterior probability for a single observation is estimated by the following equation:

$$s(\alpha_{measured}, \alpha_{expected}) = \begin{cases} e^{-50\omega^2} & \text{if } \omega < 1 \\ e^{-50(2-\omega)^2} & \text{otherwise} \end{cases} \tag{32}$$

where $\omega = \frac{|\alpha_{measured} - \alpha_{expected}|}{\pi}$. The probability, p, of a particle is just the product of these probabilities:

$$p = \prod_{\alpha_{measured}} s(\alpha_{measured}, \alpha_{expected}) \tag{33}$$

However, the particle probability is not simply set to this new value. To prevent occasionally poor observations from changing the estimate too dramatically, we place a threshold on how much a probability estimate can change in a single cycle. Therefore, the new probability of a particle is calculated by the following equation:

$$p_{new} = \begin{cases} p_{old} + 0.1 & \text{if } p > p_{old} + 0.1 \\ p_{old} - 0.05 & \text{if } p < p_{old} - 0.05 \\ p & \text{otherwise} \end{cases} \tag{34}$$

8.3.5 Resampling

Once the particle probabilities have been updated, the particles are resampled to move a higher density of particles closer to the most likely pose estimates. To do this, we copy particles from an old particle list to a new particle list in proportion to their probabilities. Higher probability particles are duplicated and lower probability particles are thrown out. The resampling is performed such that the new particles list will contain about 90 particles. For a given particle, i, in the old list, the number of times that it will appear in the new list is given by the following equation:

$$\#_i = \left\lfloor \frac{1}{\sum_j p_j} 90 p_i \right\rfloor \tag{35}$$

After copying over the old particles, triangulation estimates made from combinations of two or three beacons are added until the list contains 100 particles. Each of these particles are given a probability based on the uncertainty of the observations used in the calculation. Our methods for triangulating the robot's position are discussed in the following two sections.

8.3.6 Two Beacon Triangulation

In this approach, we use both the distance and angle estimates of the beacons, provided by high level vision, to determine the position and orientation of the robot in the global reference frame. The inclusion of beacon distance estimates (in addition to the angle that the beacon is estimated to make with the robot) in localization does produce robot position estimates that are more error prone than the estimates obtained using the angle information alone (i.e. three beacon triangulation, see Section 8.3.7). But we found that when the robot position estimates obtained using this technique are used as seed values in particle filtering (with an appropriate probability value) in addition to the estimates obtained using three beacon triangulation, the results obtained are more accurate than those with just the three beacon estimates as the seed values.

Given two beacon distances and bearings with respect to the robot's local coordinate frame, we can draw two circles, one around each of the beacons with radius equal to the distance (estimated) of the beacon from the robot. The circles intersect at two points (or none when the estimates are bad in which case they are not used in calculations), one of them being the correct estimate of the robot's position (see Figure 8).

We first use the estimated distances from the robot to the beacons to determine the robot's position with respect to a local frame with the x-axis along the line joining the two beacons. This is then converted to

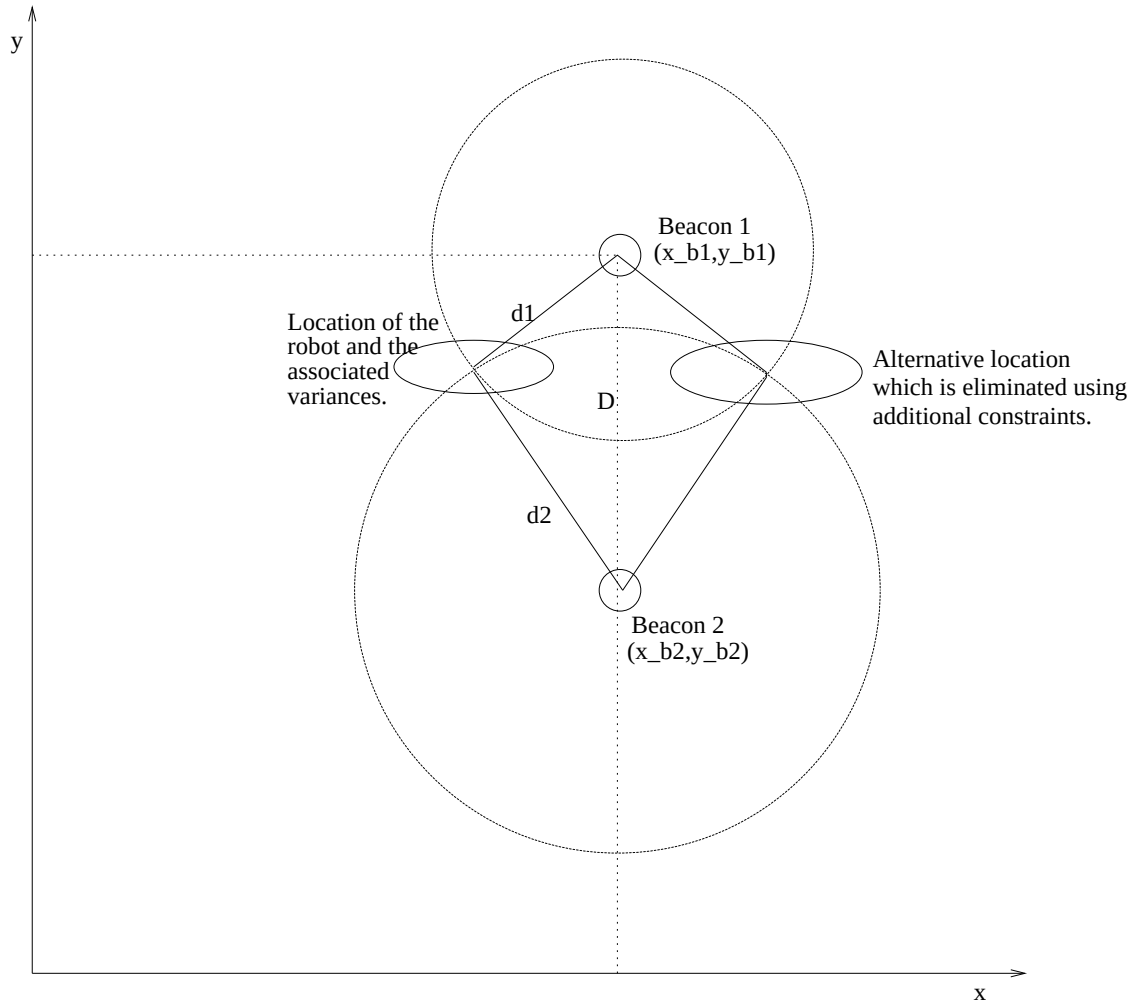


Figure 8: Localization using two beacons.

the global reference frame using the known geometry of the field. The global orientation of the robot is then determined using the robot's calculated position and estimated distances and angles to the beacons. We actually calculate both possible robot poses (position and orientation) but then eliminate one of them using constraints (for example, we check if the position is on the field). Then, we need to determine the variances in the estimated pose. To do so we basically find the partial derivatives of the expressions for pose with respect to the variables in the system. We do this starting from the final expression and move backwards to the initial expressions so that we have the "change" in pose expressed in terms of the change in the distance and bearing estimates of the beacons/markers (known values), thereby obtaining the variance estimates.

8.3.7 Three Beacon Triangulation

The image of a distant landmark can be quite small with respect to the size of an image pixel. This can result in a lack of accuracy in the distance estimates, but it does not detract from the angle estimates. Because of this, it is especially desirable to be able to estimate the Aibo's location using only angle information from the landmarks. This method eliminates the inaccuracy in the distance estimates, but it has the disadvantage of requiring knowledge about three landmarks to be applicable.

The difference between the horizontal angles of any two landmarks, combined with the actual positions

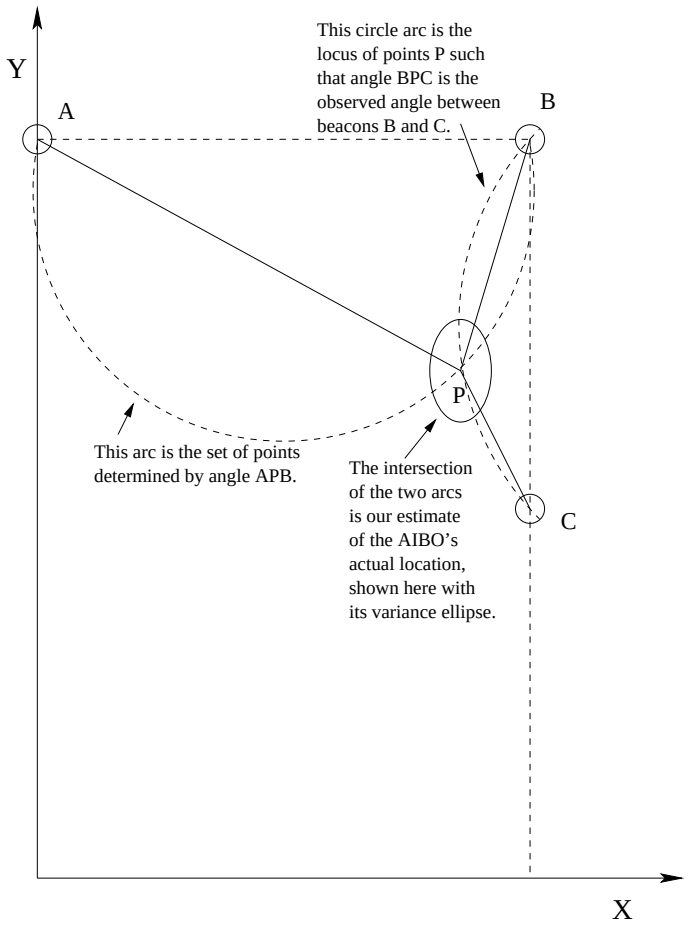


Figure 9: Three Beacon Localization. The horizontal angles between beacons A, B, and C are used to construct two circle arcs. Their intersection is the three-beacon estimate of the Aibo's location.

of those landmarks on the field, yields a circle arc of possible locations for the Aibo. Three landmarks yield two circle arcs (actually three, but the third is always redundant), whose intersection is our position estimate according to this method (Figure 9). The robot's orientation can then be determined from its position and the horizontal angle of any landmark.

8.3.8 Random Movement

In the final update step, the particles are moved locally in a random fashion. Particles with higher probabilities are moved less. This process performs a probabilistic search over nearby hypotheses. The update calculation is summarized as follows:

$$x' = x + 100mm \cdot (1 - p') \cdot rand[-1, 1] \tag{36}$$

$$y' = y + 100mm \cdot (1 - p') \cdot rand[-1, 1] \tag{37}$$

$$\theta' = \theta + 30^o \cdot (1 - p') \cdot rand[-1, 1] \tag{38}$$

Particles can be moved up to 100mm in x and y and rotated up to 30^o. These values were chosen without experimentation.

8.4 Pose Estimation

The final stage in the localization process is finding a pose estimate from the particle set. This estimate is computed in two steps: finding the largest cluster of particles and averaging the particles within that cluster.

To find the largest cluster, we divide the space of possible x,y,θ values into $10 \times 10 \times 10$ cells. We then search through all possible $2 \times 2 \times 2$ groups of adjacent cells to find the group with the most particles. The x,y,θ values for each particle in the group are then averaged according to the following equation:

$$pose = \left\langle \frac{1}{m}\sum_{i=1}^{m} x_i, \frac{1}{m}\sum_{i=1}^{m} y_i, \text{atan2}\left(\sum_{i=1}^{m}\sin(\theta_i), \sum_{i=1}^{m}\cos(\theta_i)\right)\right\rangle \tag{39}$$

where m is the number of particles in the group and $< x_i, y_i, \theta_i >$ is the pose of particle i. Notice that the θ values cannot simply be averaged because angle values wrap around.

Breaking the values into $10 \times 10 \times 10$ cells and searching the $2 \times 2 \times 2$ groups is an admittedly sup-optimal approach in that it risks missing concentrations of particles that span the boundaries of 3 adjacent cells. We leave more principled approaches to future work, but found that this method was straightforward to implement and it worked well in practice.

In addition to the pose estimate, the robot's behavior is also dependent upon its certainty in that estimate. We calculate our certainty as the density of particle probability in the largest cluster:

$$certainty = \frac{1}{n}\sum_{i=1}^{m} p_i \tag{40}$$

where n is the total number of particles and p_i is the probability of particle i. Based on this certainty value, the robot can decide whether to perform a localization-dependent skill (e.g. shot on goal) or take an information-gathering action (i.e. search for landmarks).

9 Communication

Collective decision making is an essential aspect of a multiagent domain such as robot soccer. The robots thus need the ability to share information among themselves. In this section we discuss the methodologies we adopted to enable communication and the various stages of development of the resulting communication module.

9.1 Initial Robot-to-Robot Communication

Our initial goal was to understand the capabilities and limitations of the wireless communication channel between the various robots. Although the rules required us to use TCPGateway for communication during the games, we wanted to examine other options that might be useful during non-game situations.

We created a simple server and a client that used the User Datagram Protocol (UDP). We chose UDP because it typically provides greater bandwidth than the alternative, TCP. Our intent was to determine how quickly we could transfer data between robots and to simply get used to writing applications that would allow the robots to communicate.

The first server that we created generated a few bytes of data and tried to broadcast it to a client. The client program simply gathered this data as it received it. We ran the server and the client on two different robots and monitored their actions by telnetting into them.

Once that worked, we extended our communication modules to interface with the robot's mechanical parts. The next server that we created captured the joint angles of the robot and broadcast them to the

client. The client gathered the data and set its own joint positions accordingly. Thus, when we moved the legs of the server robot, the client robot would move it's legs by the same amount, thus acting as a master-slave (puppet) interface.

As we became familiar with the networking interfaces of the robot, we continued to explore the various uses of communication. We streamed images from the robot's camera to a PC with both UDP and TCP, created a hierarchy of single-master, multiple-slaves that enabled one robot to "lead" a team of robots, and coded a remote-control program that we could use to control the Aibo from a PC. All of these experiments provided valuable feedback that we later used when creating both our official robot-to-robot communication module (described below) and UT Assist (Section 14).

9.2 TCP Gateway

Once we were familiar with the structures that the robots use to communicate, we began implementing a communication module. TCPGateway (the required interface for official robot-to-robot communication during games) abstracted away most of the low-level networking, providing a standard Open-R interface in its place. The most difficult part of getting TCPGateway working was understanding the organization of the configuration files.

The TCPGateway configuration files basically insert two network addresses and ports in the middle of an Open-R subject/observer relationship. This creates the following situation:

- Instead of sending data directly to the intended observer, the subject on the initiating robot sends data to a TCPGateway observer.
- The TCPGateway module on the initiating robot has a specific connection on a unique port for data flowing in that direction, and sends the data from the subject over that connection to the PC.
- The PC, which has been configured to map data from one incoming port to one outgoing port, sends the subject's data out to the receiving robot on a specific port.
- The TCPGateway module on the receiving robot processes the data that it receives on this port and sends the data to the intended observer.

All of the mappings described above were defined in two files on each robot (`CONNECT.CFG` and `ROBOTGW.CFG`) and in two files on the PC (`CONNECT.CFG` and `HOSTGW.CFG`).

9.3 Message Types

One of the challenges we faced regarding communication was the possibility that multiple types of messages would need to be sent. We could theoretically handle this with a stage in the brain loop that could read and distribute messages appropriately. As we proceeded, however, this option became more and more unwieldy. Variables and data that would be used in one part of a program would be read and set in another part, perhaps even in another file. What we needed was the ability to create an arbitrary number of different message types, such that anywhere in the program, we could request from the communication system the next message *of that type*.

Our first implementation kept the same communication stack, but when a request was made, the type of message was passed as a parameter. The communication system would then search through the stack for the next message of that type, remove it from the stack, and return it. This worked fine, but we quickly realized that if any one type of message ceased to be consumed, it could have serious ramifications in terms of the time needed to retrieve other types of messages.

To solve this, we implemented an array of communication stacks, one for each type of message. This gave us a constant-time fetch for the next message of any type. As messages arrived, they were processed by their type and placed into the correct stack. This way, messages related to global maps could be retrieved and used in the code that actually handles the operation of global maps, while messages relating to strategy changes could be handled in a different part of the code.

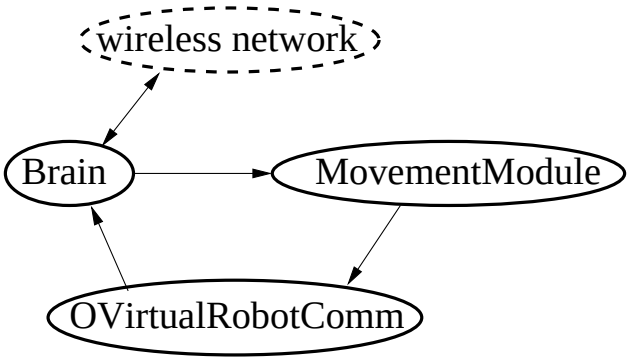


Figure 10: A high level view of the main Open-R objects in our agent. The robot sends visual data to the Brain object, which sends movement commands to the MovementModule object, which sends set points to the PID controllers in the robot. The Brain object also has network connections to teammates' Brain objects, the Robocup game controller, and our UT Assist client (Section 14). Note that this figure omits sensor readings obtained via direct Open-R API calls.

9.4 Queuing Messages

When we first tested our new multi-type communication system, we found that some of our messages were not being received. More specifically, the first message of any brain cycle was sent, but any other messages sent later in that brain cycle would be dropped. At first we thought it was just a connectivity issue. However, when we reversed the order of our messages, we found that all but the first were not delivered.

Further investigation found that the TCPGateway object was not able to process the messages we were sending quickly enough. We had enough bandwidth, and our robots were connected, but TCPGateway was just too slow to handle all the overhead for each message. The obvious solution to this was to queue our messages. Thus, when a request to send a message was made somewhere in the code, what would actually happen is that the message would be put into a queue, where it would sit until the end of the brain cycle. At the end of the brain cycle, the messages were stitched together into a long byte stream, and then sent off to the other robots. This meant that we could do all of our communication with only one TCPGateway communication per brain cycle, which cut back on the total overhead.

10 General Architecture

Due to our bottom-up approach, we did not address the issue of general architecture until some important modules had already taken shape. We had some code that cobbled together our vision and movement components to produce a rudimentary but functional goal-scoring behavior (see Section 12.1.1). Although this agent worked, we realized that we would need a more structured architecture to develop a more sophisticated agent, particularly with the number of programmers working concurrently on the project. The decision to adopt the architecture described below did not come easily, since we already had something that worked. Implementing a cleaner approach stopped our momentum in the short term and required some team members to rewrite their code, but we feel the effort proved worthwhile as we continued to combine more independently-developed modules.

We designed a framework for the modules with the aim of facilitating further development. We considered taking advantage of the operating system's inherent distributed nature and giving each module its own process. However, we decided that the task did not require such a high degree of concurrent computation, so we organized our code into just two separate concurrent objects (Figure 10).

We encapsulated all of the code implementing low-level movement (Section 5.2.1) 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 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 Section 5.2.1.

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, and Communication. 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. Generally the end result of this step is to update the agent's internal representation of its external environment: the global map (see Section 11).

Act: After the Brain has taken care of sensing, it invokes those modules that implement acting, described in Sections 12 and 13. These modules typically don't directly access the data gathered by the Brain. Instead they query the updated global map.

11 Global Map

Early in the development of our soccer playing agent, particularly before we had functioning localization and communication, we chose our actions using a simple finite state machine (see Section 12). Our sensory input and feedback from the Movement Module dictated state transitions, so sensations had a relatively direct influence on behavior. However, once we developed the capability to locate our agents and the ball on the field and to communicate this information, such a direct mapping became impossible. We created the global map to satisfy the need for an intermediate level of reasoning. The global map combines the outputs of Localization from Vision and from Communication into a coherent picture of what is happening in the game, and it provides methods that interpret this map in meaningful ways to the code that governs behavior.

11.1 Maintaining Location Data

When a robot computes new information about the location of any particular object on the field, it usually merges the new estimate of position with the current estimate of position that is stored in its global map (see Section 8.3.3).

As time passes, the error estimate for all of the information in the global map increases. This degradation of information is included to more accurately model the rapid rate of change in the state of the game. The

idea is to make the degradation smooth to reflect the maximum change that we are ready to allow (i.e. the change that we think could have happened) since the last update. The approach used here is to estimate a maximum 'velocity' by which we assume the object can move along the x and the y axes. We then use this velocity to calculate the maximum distance the object could have moved along the axes in the time since the last update. The estimated change, σ_{change}, is statistically added to the location's uncertainty in accordance with the formula:

$$\sigma_{updated} = \sqrt{\sigma^2_{previous} + \sigma^2_{change}} \tag{41}$$

For example, if we consider the modeling of the opponents, we want our estimates of the opponents to be as accurate as possible and we do not want new estimates to occur every frame. We would ideally want to be able to merge estimates from the current frame with those in the previous frame, wherever possible, so that we can actually map the motion of the opponents. At the same time, we may have spurious detections every once in a while and if they are not seen in successive frames, we want these estimates to disappear quickly. So for opponents we use an artificially high 'velocity' such as 1500 mm/s (determined by experimentation). On the other hand we want the estimates of the ball, robot position and those of the teammates to degrade depending on some 'velocity' that reflects their actual motion. So we choose the velocity for teammate motion as 300 mm/s (we do not think any team can move any faster than that as yet) and that for the ball as 1000 to 1500 mm/s because the ball can move about that fast after a single powerful kick. These values were all determined experimentally and seem to provide reasonable performance in terms of how we would like our estimates to be updated.

11.2 Information from Teammates

Each robot periodically sends information from its global map to each of its teammates. This transmitted information includes:

1. The location of the robot, along with an error estimate.
2. The locations of any opponents of which the robot currently is aware, along with error estimates.
3. The location of the ball, along with an error estimate.

When robot A receives teammate position information from robot B, robot A always assumes that B's estimate of B's position is better than A's estimate of B's position. Therefore, robot A simply replaces it's old position for B with the new position.

When a robot receives opponent information from another robot, it updates it's current estimate of opponent locations as described in Section 4.6.

If robot A has seen the ball recently when it receives a ball position update from robot B, robot A ignores B's estimate of ball position. If robot A hasn't seen the ball recently, then it merges its current estimate of the ball's position with the position estimate that it receives from robot B.

The basic idea behind having a global map is to make sharing of information possible so that the team consisting of individual agents with limited knowledge of their surroundings can pool the information to function better as a team. The aim is to have completely shared knowledge but the extent to which this succeeds is dependent upon the ability to communicate. Since the communication (see Section 9) is not fully reliable, we have to design a good strategy (Section 12 describes our strategy and behaviors) that uses the available information to the maximum extent possible. Other modules can access the information in the GlobalMaps using the accessor functions (predicates) described in the following section.

11.3 Providing a High Level Interface

From a high level perspective, the only data that the global map provides to other modules are the estimated positions of the ball and the robots on the field, along with degrees of uncertainty about these

estimates. However, the global map also houses an array of functions on these data, to prevent different portions of the behavior code from replicating commonly used predicates and high level queries. See Table 13 for a complete list of these functions, most of whose names are clear indicators of their functionality. Note that they range from relatively low level methods that return the position of an individual robot (`getTeamMembers`) to relatively high level methods such as `NumOpponentsWithinDistance`. They include tactical considerations, such as whether `IAmClosestToBall`, as well as methods relative to our strategic roles (see Section 13.2.1), such as `GetDistanceFromSupporter`. Finally, methods such as `AmIInDefensiveZone` and `IsDefenderWellLocalized` provide a more abstract interface to the position estimates.

`getID`	`GetDistanceFromDefender`	`InLeftThird`
`getTeamMembers`	`GetDistanceFromKeeper`	`InCentralThird`
`getOpponents`	`GetAttackerRelativePosition`	`InRightThird`
`getBall`	`GetSupporterRelativePosition`	`InTopQuarter`
`getMyPosition`	`GetDefenderRelativePosition`	`InOwnHalf`
`adjustRelativeBall`	`GetKeeperRelativePosition`	`IsLower`
`wellLocalized`	`GetAttackerAbsolutePosition`	`InOwnGoalBox`
`ballOnField`	`GetSupporterAbsolutePosition`	`AmILeftMost`
`getBallDistanceFromOurGoal`	`GetDefenderAbsolutePosition`	`AmIRightMost`
`getRelativeBall`	`GetKeeperAbsolutePosition`	`GetLeftPosAngle`
`getRelativeOrientation`	`IsAttackerWellLocalized`	`GetRightPosAngle`
`getRelativeOpponentGoal`	`IsSupporterWellLocalized`	`OpponentsOnLeft`
`getRelativeOwnGoal`	`IsDefenderWellLocalized`	`OpponentsOnRight`
`getRelativeOpponents`	`IsKeeperWellLocalized`	`NumOpponentsOnLeft`
`getRelativeTeamMembers`	`BallInOwnGoalBox`	`NumOpponentsOnRight`
`GetRelativePositionOf`	`BallInOppGoalBox`	`OnOurSideOfTheField`
`GetRelativePositionOfTeamRel`	`BallInOurHalf`	`OnLeftSideOfTheField`
`HeadingToOffPost`	`AmIInDefensiveZone`	`IAmClosestTo`
`HeadingToDefPost`	`NearOwnGoalBox`	`IAmClosestToBall`
`GetClosestCorner`	`NumberOfTeamMatesInOpponentHalf`	`NumOpponentsWithinDistance`
`DistanceToOffPost`	`NumberOfTeamMatesInOwnHalf`	`GetRelativePositionTo`
`DistanceToDefPost`	`HeadingToOppGoal`	`InZone`
`GetDefensivePost`	`HeadingToOwnGoal`	`ApproachingZone`
`GetDistanceFromAttacker`	`HeadingToOppLeftCorner`	
`GetDistanceFromSupporter`	`HeadingToOppRightCorner`	

Table 13: The predicates that GlobalMap provides.

12 Behaviors

In this section we describe the robot's soccer-playing behaviors. In our development, we had relatively little time to focus on behaviors, spending much more of our time building up the low-level modules such as walking, vision, and localization. As such, the behaviors we describe here are far from ideal. We anticipate overhauling this component of our code base should we participate in future competitions. Nonetheless, we present a detailed description for the sake of completeness, and to illustrate what was possible in the time we had to work.

12.1 Goal Scoring

One of the most important skills for a soccer-playing robot is the ability to score, at least on an empty goal. In this section we describe our initial solution that was devised before the localization module was developed,

followed by our eventual behavior that we used at RoboCup 2003.

12.1.1 Initial Solution

Once we had the initial movement and vision modules in place, we were in a position to "close the loop" by developing a very basic goal scoring behavior. The goal was to test the various modules as they interacted with each other. Since neither the localization module (Section 8) nor the general architecture (Section 10) had been implemented by this time, this behavior was entirely reactive.

This goal scoring behavior, implemented as a Finite State Machine (FSM), assumes that the robot is placed at a point on the field such that the distance between the orange ball and the robot is not more than one half the length of the field (i.e. the ball is at a distance where it can be seen by the robot). A point to note here is that this constraint could have been removed by incorporating a "random walk" sequence into the behavior. The robot first performs a three-layer head scan to determine if it can "see" the ball at its current position. If the ball is not in its visual field at this stage, the robot starts strafing (turning 360° about its current position) in search of the ball. In either case, the detection of a ball in a single visual frame causes the robot to stop and determine if the ball has actually been seen (noise in the image color segmentation can sometime cause false ball detections in high level vision). Once the ball is in sight, the robot walks towards it by tracking the centroid of the ball with its head and moving its body in whatever direction the head points to. This walking state continues until either the ball is lost from the visual frame (in which case the robot goes back to searching for the ball) or the robot reaches a point sufficiently close to the ball, as determined by its neck angles at that point. The thresholds in the neck angles are set such that the robot stops with the ball right under its head. Next, the robot strafes around the ball with its head held at 0° tilt), searching for the offensive goal (blue or yellow depending on whether the robot is on the red team or the blue team). Once the goal is found, the robot checks to ensure that the ball is still under its nose and then tries to kick the ball into the goal. If the robot finds that it has lost the ball (it sometimes pushed it away accidentally while strafing), it goes back to searching for the ball.

This behavior, despite being extremely rudimentary, helped us understand the issues involved in the interaction/communication between modules. It also served to illustrate the importance of a good architecture in implementing complex behaviors. At the time of the American Open, this was the only goal-scoring behavior that we had implemented.

12.1.2 Incorporating Localization

When localization came into place, we replaced the above behavior using strafing and a single kick with a more complex behavior involving the chin pinch turn. In the new behavior, the decision about which kick to use is made according to knowledge about where on the field the robot is and whether there are opponents nearby.

Figure 11 summarizes the kicking strategy used when no opponents are detected nearby. If the robot is on the offensive half of the field and is not near any walls, it follows the natural strategy of turning toward the goal and then kicking the ball. On the quarter of the field nearest the offensive goal, the front power kick is used rather than the fall forward kick. This is because we believe the front power kick to be more accurate than the fall forward kick, although less powerful.

When the robot is in the defensive half of the field, it kicks toward the far same-side corner (that is, if it is on the left half of the field, it kicks toward the offensive-half left corner). The reasoning behind this was that when the ball is in the robot's defensive half, the most important thing is to clear the ball to the other half of the field. Since other robots are generally more likely to be in the center of the field, a good strategy for accomplishing this is to kick toward the outside so that the ball will on average be allowed to travel farther before its path is obstructed.

When the robot is near the wall and facing it, the head kick is typically used.[11] This is chiefly because

[11] The exception to this is when the robot is close to and directly facing the back wall near its defensive goal, a situation which occurs relatively rarely. In this case, the yoshi kick is used, because under these circumstances it is likely to succeed at pushing the ball in the correct direction, and there is also a good chance that it will kick the ball farther than the head kick.

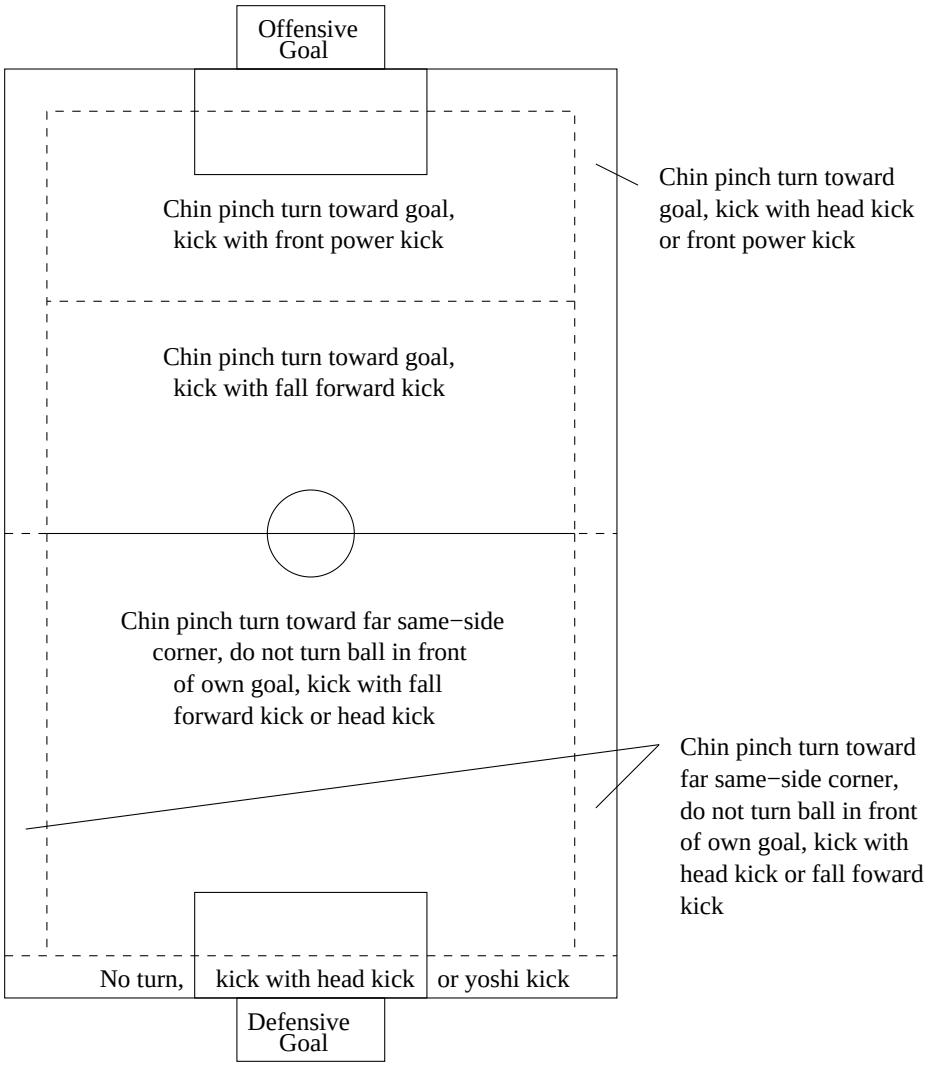


Figure 11: Kicking strategy when no nearby opponents are detected

we want to use the chin pinch turn as little as possible when we are along the wall. The more the robot runs into the wall while moving, the larger the discrepancy becomes between the actual distance traveled and the information that odometry gives to localization. Because the FSM uses localization to determine when to switch from chin pinch turning to kicking, the longer the robot uses a chin pinch turn along a wall, the less likely it is to stop turning at the right time. So, it is typically a better strategy when very near a wall and facing it to head kick the ball along the wall rather than trying to turn with the ball to an exact angle and then kick with a more powerful kick.

Another situation where the head kick is used is when we would otherwise need to turn more than 180 degrees with the ball. This situation typically arises when the robot is in the defensive half and needs to avoid turning in a way that will pass the ball between it and its own goal. A 360-degree chin pinch turn takes approximately 5 seconds. Thus, given that many of our kicks take a small amount of time to prepare before hitting the ball away, chin pinch turning for more than 180 degrees carries the danger of putting us in violation of the 3-second holding rule. Therefore, in situations where we need to turn through some angle $\theta > 180$ degrees, we instead turn through $\theta - 80$ (or 180, if $\theta - 80 > 180$) degrees and then head kick in the appropriate direction.

If opponents are detected nearby, the robot simply kicks with the head kick in the direction of the goal unless the goal is directly behind the robot, in which case it kicks with the yoshi kick. The reasoning behind this is the same as the reasoning just described underlying the choice of the head kick near walls.

12.1.3 A Finite State Machine

Our behaviors are implemented by a Finite State Machine (FSM), wherein at any time the Aibo is in one of a finite number of states. The states correspond roughly to primitive behaviors, and the transitions between states depend on input from vision, localization, the global map, and joint angles. This section describes the FSM underlying our main goal-scoring behavior. As we developed our strategy more fully, this became the behavior of the attacker (see Section 13.2.1). The behaviors of the other two roles are discussed in Section 13.2.1 as well.

The main goal scoring states are listed here. Note that the actions taken in these states are executed through the Movement Interface, and they are described in more detail in Section 5.2.2.

- Head Scan For Ball: This is the first of a few states designed to find the ball. While in this state, the robot stands in place scanning the field with its head. We use a two-layer head scan for this.
- Turning For Ball: This state corresponds to turning in place with the head in a fixed position (pointing ahead but tilted down slightly).
- Walking To Unseen Ball: This state is for when the robot does not see the ball itself, but one of its teammates communicates to it the ball's location. Then the robot tries to walk towards the ball. At the same time, it scans with its head to try to find the ball.
- Walking To Seen Ball: Here we see the ball and are walking towards it. During this state the robot keeps its head pointed towards the ball and walks in the direction that its head is pointing. As the robot approaches the ball, it captures the ball by lowering its head right before transferring into the Chin Pinch Turn state.
- Chin Pinch Turn: This state pinches the ball between the robot's chin and the ground. It then turns with the ball to face the direction it is trying to kick.
- Kicking: While in this state, the robot is kicking the ball.
- Recover From Kick: Here the robot updates its knowledge of where the ball is and branches into another state. Both of these processes are influenced by which kick has just been performed.

- Stopped To See Ball: In this state, the robot is looking for the ball and has seen it, but still does not have a high enough confidence level that it is actually the ball (as opposed to a false positive from vision). To verify that the ball is there, the robot momentarily freezes in place. When the robot sees the ball for enough consecutive frames, it moves on to Walking To Seen Ball. If the robot fails to see the ball, it goes back to the state it was in last (where it was looking for the ball).

In order to navigate between these states, the FSM relies on a number of helper functions and variables that help it make state transition decisions. The most important of these are:

- BallLost: This Boolean variable denotes whether or not we are confident that we see the ball. This is a sticky version of whether or not high level vision is reporting a ball, meaning that if BallLost is true, it will become false only if the robot sees the ball (according to vision) for a number of consecutive frames. Similarly, a few consecutive frames of not seeing the ball are required for BallLost to become true.

- NearBall: This function is used when we are walking to the ball. It indicates when we are close enough to it to begin capturing the ball with a chin pinch motion. It is determined by a threshold value for the head's tilt angle.

- DetermineAndSetKick: This function is used when transitioning out of Walking To Seen Ball upon reaching the ball. It determines whether or not a chin pinch turn is necessary, what angle the robot should turn to with the ball before kicking, and which kick should be executed.

Finally, an overview of the rules that govern how the states transition into one another is given in Figure 12.

12.2 Goalie

In this section we detail our initial (pre-localization) and final (RoboCup-2003) goalie behaviors.

12.2.1 Initial Solution

Like the rest of our behaviors, our goalie behavior used an FSM for control. The initial behavior was as follows: Once it started, the first thing would be to look around for the goal, go to it, turn around and stand there, in front of the goal, looking forward to see if it saw the ball. If it saw the ball, it would start to "track" it, i.e. keep its eye on the ball and turn in place if necessary. If the ball got too close, it would stretch its arms out, hoping to block the ball (Figure 13).

Closeness to the ball was based on the head tilt angle. Since we didn't have localization working properly at the time, this was the only way to reliably tell distance. The goalie would track the ball, which entails moving the head such that the ball is in the center of the field of vision (and turning the body in place if turning the head isn't enough). Therefore the head would always be pointed towards the ball and the closer the ball, the larger the tilt angle (Figure 14). The angles for being "close" were determined by trying various angles on the field.

This simple approach had many problems, some due to its simplicity and some due to inabilities of our Aibos at a lower behavioral level. For example, tracking the ball didn't work fast enough for the Aibo to react even to slow shots coming towards it. The ball would just roll by the goalie who would lose sight of it because its head wouldn't get moved in time to track the ball.

The most important problem was the passiveness of our goalie. Judging that we would be dead in the water if we just waited passively for the ball to slowly roll up to us, we decided to take a more active approach. Our revised goalie waited in its goalbox until the ball came within a safety distance and then it walked to the ball and attempted to clear it. This approach worked much better but it also brought along some new problems to solve:

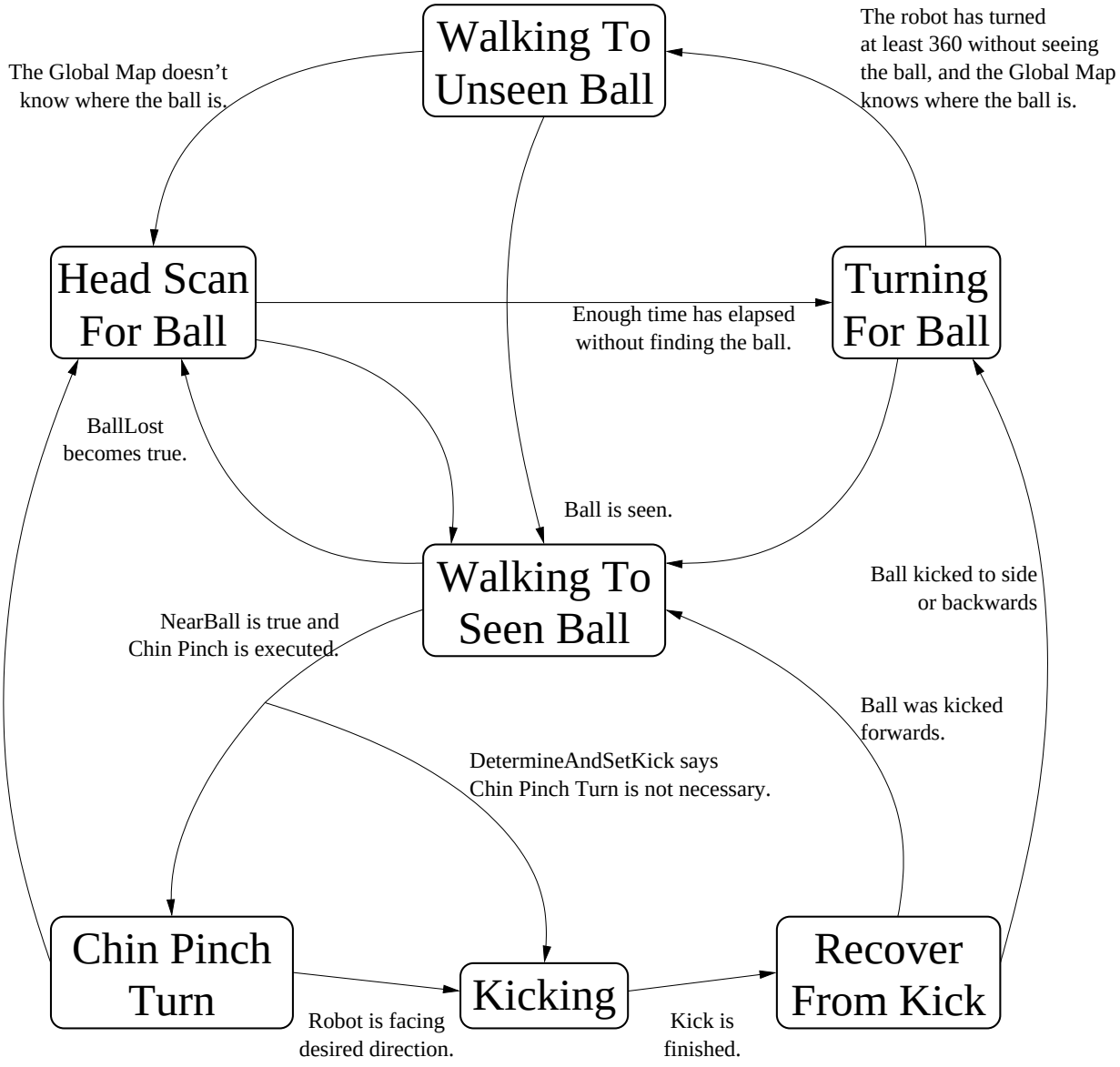


Figure 12: The finite state machine that governs the behavior of the attacking robot.

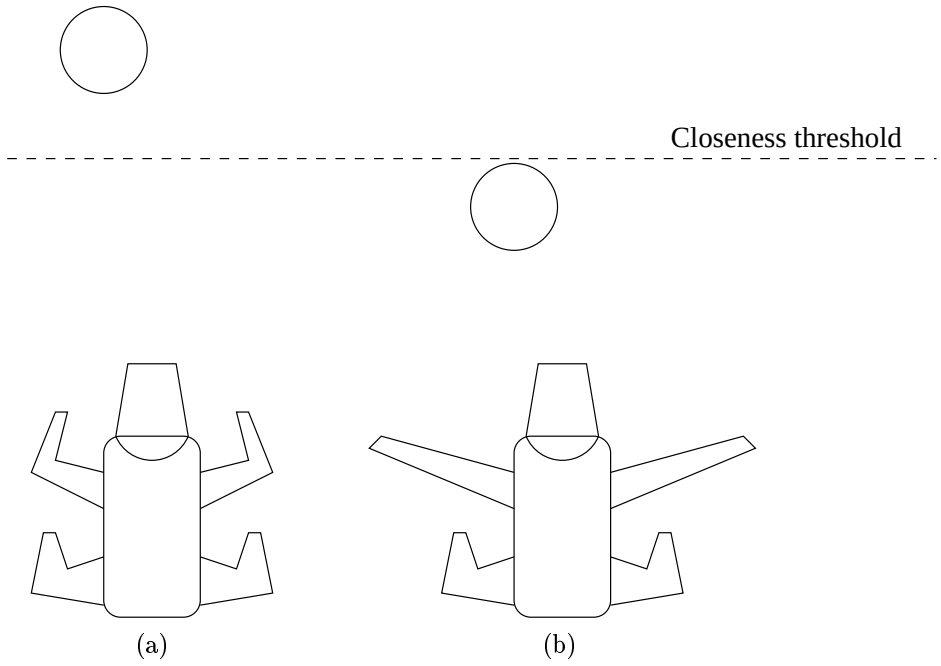


Figure 13: First attempt at a goalie: (a) it waits for the ball to get within a closeness threshold and then (b) stretches its arms out to block the ball. This approach didn't work well since ball tracking was slow. By the time the goalie stretched, the ball would be long gone.

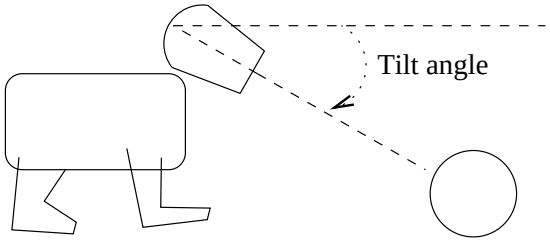


Figure 14: Closeness to ball was based on the how large the head tilt angle was.

- How close should the safety distance be? We don't want the goalie to leave the goal to clear a ball at midfield, but if it waits too long, the opponent with the ball will have too great a chance of shooting a fast shot into the goal.
- How should the robot clear the ball? If it just stretches as it did before, it won't be pushing the ball away from itself.
- When and how should it get back in position? There is often an opponent behind or next to the ball, so after attempting to clear it, the goalie needs to make sure the ball's not still right in front of it. Once it's cleared the ball, what is the quickest way to go back to position in the goalbox while minimizing the possibility of being caught unaware of the ball coming back to the goal?

The first method we used for clearing the ball was the simple stretching out to the sides, which worked sometimes but usually didn't clear the ball very far and just left it in place or pushed it to the side a little. This motivated us to experiment with different kicking styles. Some kicks we tried were the chest push kick, the arms together kick, the fall forward kick and the "right (or left) swerve kick." (See Section 7 for details of the first three kicks.) In the right (left) swerve kick the robot backs up to the right (left) side, raises its right (left) leg and then quickly brings it diagonally down and towards the inside, somewhat like a karate chop. This kick was one of our most powerful kicks early in our development process. However, after we developed the fall forward kick, we experimented with using different kicks in different situations (e.g. use the fall forward kick when the ball is right in front and the swerve kick when it is off to the side). Eventually we decided that the best approach was to always use the fall forward kick.

When the goalie tries to clear the ball, frequently the ball stays where it is or moves a very small amount. This can be due to not executing the kick perfectly or, more often, the fact that there is an opponent robot right behind the ball, keeping it from rolling away. To make sure the ball is cleared and it is safe to go back to the waiting position inside the goalbox, the goalie checked to see if the ball was right under it after kicking. If it saw the ball there it would try to kick again. This was repeated until the ball was successfully cleared.

Going back to the goal after clearing was one of the trickiest parts because we didn't have any localization initially. All the goalie knew was to recognize the ball and the goal. We didn't want the goalie to just turn back, look for the goal, walk back to it and turn around to face the field. That would mean spending a long time without looking at the ball, which might give our opponents a chance to score since the ball might not have been cleared very far away (even though we make sure its not right under our head). The solution to this was to walk backwards after successfully clearing the ball and at the same time keeping looking for the ball in case it appears close to the robot. This worked quite well and the goalie kept watching the ball when it wasn't cleared far away, but there was a new problem. When the goalie saw the ball while walking backwards, it would return to tracking and go out to clear the ball if it got close enough. After going out to clear and walking back a few times, the error in position would get large and the goalie would start drifting away from its home position. To counter this, we changed the behavior so the goalie would turn around and go back to its home position after walking out for a long time ("a long time" being chosen arbitrarily based on experiments on the field).

There are many ways in which the capabilities of the goalie can be improved. Adding localization was done after the American Open, and is described in the next subsection. Getting better ball tracking ability with faster reaction to fast-moving balls (such as shots) is definitely needed and would improve goalkeeping behavior substantially.

12.2.2 Incorporating Localization

Once our goalie had the ability to determine its position on the field, our primary strategy shifted to staying between the ball and the goal. Given the size of the goalie with respect to the goal, we adopted a fairly conservative strategy that kept the goalie in the goal most of the time.

Whenever the goalie saw the ball, it oriented itself such that it was pointed at the ball and situated between the ball and the goal. If the ball came within a certain distance of the goal, the goalie advanced

towards the ball and attempted to clear it. After attempting to clear the ball, the goalie retreated back into the goal, walking backwards and looking for the ball. Any time the goalie saw the ball in a non-threatening position, it oriented itself towards the ball and continued its current course of action.

Whenever the ball was in view, the goalie kept a history of ball positions and time estimates. This history allowed the goalie to approximate the velocity of the ball, which was useful in deciding when the goalie should "stretch out" to block a shot on the goal.

One interesting dilemma we encountered concerned the tradeoff between looking at the ball and looking around for landmarks. It seemed very possible that, given the goalie's size, if it could just stay between the ball and the goal it could to a fairly good job of preventing goals. However, this strategy depended on the goalie both being able to keep track of it's own position and the ball's position. When we programmed the goalie to fixate on the ball, it was not able to see enough landmarks to maintain an accurate estimate of its own position. On the other hand, when the goalie focussed on the beacons in order to stay localized, it would often miss seeing the approaching ball. It proved to be very difficult to strike a balance between these two opposing forces.

13 Coordination

In this section we describe our initial and eventual solutions to coordinating multiple soccer-playing robots.

13.1 Dibs

Our first efforts to make the players cooperate resulted directly from our attempts to play games with 8 players. Every game would wind up with six robots crowded around the ball, wrestling for control. At this point, we only had 2 weeks before our first competition, and thus needed a solution that did not depend on localization, which was not yet functional. Our solution was a process we called *Dibs*.

13.1.1 Relevant Data

In developing Dibs, we tried to focus on determining both what data were available to us, and of that data, which were relevant. Because we did not have a coherent set of global maps at this point, any information from other robots would have to come directly into the Dibs system. As we created the system, it became more and more clear that the only thing we cared about was how far from the ball each robot was. Our first attempt simply transmitted the ball distance to every other robot. Each robot would then only go to the ball if its distance estimate was lower than that of every other robot.

13.1.2 Thrashing

Unfortunately, this first attempt did not work so well. First of all, the robots' perception of their distance to the ball was very heavily dependent on how much of the ball they could see, how the lights were reflecting off the surface of the ball, and how much of the ball was actually classified as "orange." This means that estimates of the ball's distance varied wildly from brain cycle to brain cycle, often by orders of magnitude in each direction. Secondly, even when estimates were fairly stable, a robot could think that it was the closest to the ball, start to step, and in the process move slightly backward, which would signal another robot to go for the ball. The other robot would begin to step, moving slightly backward at first, and the cycle would continue *ad infinitum*.

13.1.3 Stabilization

To correct these problems, we decided that re-evaluating which robot should go to the ball in each brain cycle was too much. Evaluating that frequently didn't give a robot the chance to actually step forward (this was before our walk was fully developed as well), so that its estimate of ball distance could decrease. However, we couldn't just take measurements every n brain cycles and throw away all the other information — we

were strapped for information as it was, and we didn't want one noisy measurement to negatively affect the next n brain cycles of play. Our solution was to take an average of the measurements over a period of time, and instead only *transmit* them every n brain cycles.

13.1.4 Taking the Average

Because the vision is somewhat noisy (i.e. the robot sometimes sees the ball when it is not there, and sometimes doesn't see it when it is there), it didn't make sense just to take the raw mean of the estimates over the period of n brain cycles. We decided that unless the robot saw the ball for at least $\frac{n}{2}$ cycles in each period, it would report an essentially infinite distance to the ball. If it *did* see the ball enough, it would take all the non-infinite estimates in that "transmit cycle", discard some fixed number of highest and lowest values (an attempt to clean up some of the noise), and then transmit the mean of the remaining values.

13.1.5 Aging

To prevent deadlock we introduced an aging system into Dibs. Originally, if a robot had transmitted a very low estimate of distance to the ball, and then crashed or was removed from play, any other robots would just remain watching the ball, because they would still have the other robot's estimate in their memory. Thus, at the end of each transmit cycle, we incremented the age of each other robot's estimate. When the age reached a pre-determined cutoff (10 in our case), the estimate was discarded and set to the maximum value. In this way, other robots could then resume attacking the ball.

13.1.6 Calling the Ball

Another problem we ran into involved the "strafe" state. Once a robot had established "Dibs" on the ball, it would walk towards the ball while the other robots watched the ball closely. When the robot reached the ball, however, it would look up, in order to find the goal. While it was looking up, its ball estimates would all go to the maximum value, and other robots would resume attacking the ball. More often than not, this would result in a robot strafing to find the goal, while another robot of ours would come up and take the ball right out from under the nose of the first. Next, the second robot would start to strafe, and a large tangle of robots would result. To prevent this, we added functions called "callBall" and "relinquishBall." These functions merely set flags that made the robot start lying about its distance to the ball and stop lying, respectively. When lying about its distance to the ball, the robot would always report zero as its distance estimate. This way, whenever the robot entered the strafing state, it could effectively let the other robots know that even though it wasn't seeing the ball, they shouldn't go after it. The robot would then relinquish the ball at the beginning of most states, including when it had lost the ball and when it had just finished kicking the ball.

13.1.7 Support Distance

The system described so far worked pretty well in that it prevented more than one robot from going to the ball at once. However that was all it did. One robot might be going to the ball, but all the others would just stare at the ball, regardless of how far away they were. We determined that this was considerably sub-optimal, and that even if a robot is dribbling the ball down the field toward the enemy goal, if it were to lose the ball, it would be nice to have another robot nearby to recover, if possible. Thus we introduced the concept of a "support distance." Originally set at half a meter, and then tuned to approximately a meter, the support distance was how close the robot would have to be to the ball before its lack of Dibs would prevent it from advancing further. While we only enjoyed limited overall success using the support distance technique, it was a marked improvement over ordinary Dibs.

13.1.8 Phasing out Dibs

Once localization was brought online, the need for multiple types of transmissions (which Dibs did not respect) and the desire to use localization data dictated a phasing out of Dibs. Because Dibs was so carefully tuned to the robots' playing style, cooperation actually worsened for quite a while before it improved after phasing out Dibs. However, as with many things, it needed to get worse before it could get better.

13.2 Final Strategy

Here we describe the coordination strategy developed during the last week or so before RoboCup 2003. In particular, it takes advantage of both localization and global maps.

13.2.1 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 goalie does not participate in the role system. This section gives an overview of the ideas behind the roles. The following sections describe in more detail the supporter's and defender's behaviors and under what conditions the roles change.

The roles are dynamically assigned, in that at the start of each Brain cycle, a given robot reevaluates its role based on its current role, its global map information, and other strategic information communicated to it by its teammates. The default allocation of roles is for there to be one defender and two attackers. Under certain circumstances an attacker can become a supporter, but after some time it changes back into an attacker. It is also possible for the defender to switch roles with an attacker. There should always be exactly one defender and at least one attacker.

The differences between the roles manifest themselves in the robots' behaviors. Here is a summary of the differences between the behaviors effected by the different roles. The attacker's behavior is described in more detail in Section 12.1, and the supporter and defender behaviors are described more fully below.

- An attacker robot focuses exclusively on goal-scoring. That is, it tries to find the ball, move to it, and kick it towards the goal.
- The supporter's actions are based on a couple of goals. One is to stay out of the way of the attacker. This is based on the idea that one robot can score by itself more effectively than two robots both trying to score at the same time. Another goal is to be well placed so that if the attacker shoots the ball and it ricochets off the goalie or a wall the supporter can then become the attacker and continue the attack.
- Our defender stays on the defensive half of the field at all times. Its job is to wait for the ball to be on its half and then go to the ball and clear it back to the offensive side of the field.

13.2.2 Supporter Behavior

The supporter uses an omnidirectional walk to try to simultaneously face the ball and move to a *supporting post*. If it sees the ball, it keeps its head pointing towards the ball and tries to point its body in the same direction as its head. If it doesn't see the ball, it tries to turn towards its global map location of the ball and scans with its head to try to find it. It is very rare for there to be a supporter that has no idea where the ball is (i.e. while no robot sees the ball).

The location of the supporting post is a function of the position of the ball. For this we use a team-centric coordinate system where the edge of the field including the defensive goal line is the positive x-axis, the left edge of the field is the positive y-axis, and the units are millimeters. If the coordinates of the ball are (x, y), then the supporting post, (S_x, S_y), is given by

$$S_x = \begin{cases} 1150 & \text{if } x > 1450 \\ 1750 & \text{if } x \leq 1450 \end{cases} \tag{42}$$

and

$$S_y = \min\left(\frac{y + 4200}{2}, 3800\right). \tag{43}$$

13.2.3 Defender Behavior

The role of a defender in robot soccer is not much different from that in real soccer — to prevent the opponents from moving the ball anywhere near the goal it is defending and to try and kick the ball, when in its own half, towards a team member in the other half. We decided to go for a very conservative defender such that there is always one robot in our half defending the goal. At the same time we wanted to ensure that under conditions where the defender is in a better position to function as the attacker, there is smooth switching of roles between the robots.

When a robot is assigned the defender role, its first action is to walk within a certain distance (approx. $200mm$) of a predefined defensive post that is roughly the center of the defensive half of the field. Once it gets within this distance of the defensive post, it either turns such that it faces the ball which is within its field of vision or it turns to face the point where it thinks the ball is based on the result of merging the estimates from other teammates in its global map (see Section 11). If it cannot see the ball and also does not receive any communication regarding the ball from other teammates (a rare occurrence), it starts searching for the ball once it gets to the defensive post. Even while it is walking to this post, if it sees the ball and finds, on the basis of its current world knowledge, that it is the closest to the ball, it starts walking to the ball. Once it gets to the ball it tries to kick the ball away from the defensive zone (the bottom three-fourths of the half of the field that it is defending). For the defender, we use a combination of the chin-pinch turn and the fall forward kick (see Section 7), as it is the most powerful kick we have. While kicking, the defender always tries to angle the kick away from its own goal and towards one of the corners of the opposition. This strategy allows us to clear the ball in most instances and even takes it a long way into the other half thereby giving the attacker(s) (or attacker and supporter) a better chance of scoring a goal. According to the rules of the competition, none of the team members can enter the penalty box around their own goal. To accommodate this in the defender and in the other team members excluding the goalkeeper, we add a check that prevents the robot from entering the goal box and a "buffer" region around it. If the ball is within this region, the robot just tracks the ball and lets the goalkeeper take care of clearing the ball.

13.2.4 Dynamic Role Assignment

Our role assignment system has three main facets. One is a set of general rules that serve to maintain the status quo of there being exactly one defender and at least one attacker. Next are the rules that determine when one of the two attackers becomes a supporter and then when it switches back. The last set of rules orchestrates timely switches between the defender and an attacker.

General Rules We label the three robots R_1, R_2, and R_3. Then the following rules influence R_1's choice of role. (The rules are the same for each robot; the labels are to distinguish which robot's role is being determined presently.)

- The default is for each robot to keep its current role. It will only change roles if a specific rule applies.
- If R_1 finds that it is "alone" in that it has not been receiving communication from other teammates for some time, it automatically assumes the role of an attacker.
- In most cases, communication works fine, and if neither R_2 nor R_3 is a defender, then R_1 will automatically become (or stay) a defender. This ensures that (under normal conditions) there will always be at least one defender. Ensuring that there is not more than one defender is taken care of in the section on attacker and defender switching.

- If R_1 is a supporter and so is R_2 or R_3, then R_1 will automatically become an attacker. This could happen accidentally if two supporters simultaneously decide to become supporters without enough time in between for the second one to be aware of the first's decision. In this case this rule ensures that at least one of the supporters will immediately go back to being an attacker.

Attackers and Supporters A number of considerations influence our mechanism for switching between attacker and supporter. One such consideration is that we want to prevent a robot from changing roles twice with very little time in between. This is because a robot that keeps changing roles very frequently behaves in a scattered manner and is unable to accomplish anything. To enforce this, we made the roles somewhat *sticky*. That is, for an attacker or supporter, there is an amount of time such that once the robot enters that role, it is unable to leave it until that much time has passed. Presently, the amount of time for an attacker is 2.5 seconds, and for a supporter it is 2 seconds. Notably, stickiness can easily be in conflict with the general rules listed above. In these cases we give stickiness the highest priority. We also considered giving the general rules highest priority, and it is still not completely clear to us which system is better.

An important measure that we use to evaluate a robot's utility as an attacker is its *kick time*. This is an estimate of the amount of time it will take the robot in question to walk up to the ball, turn it towards the goal, and kick. Each robot calculates its own kick time and communicates it to the other robots as part of their communication of strategic information. The estimated amount of time to get to the ball is the estimated distance to the ball divided by the forward speed. The time to turn with the ball is determined by calculating the angle that the ball will have to be turned and dividing by the speed of the chin-pinch turn.

Consider the case where there are two attackers, A_1 and A_2. Once A_1's period of stickiness has expired, it will become a supporter precisely when *all* of the following conditions are met:

- A_1 and A_2 both see the ball. This helps to ensure the accuracy of the other information being used.
- The ball is in the offensive half, as well as both robots A_1 and A_2. Becoming a supporter is only useful when our team is on the attack.
- A_1 has a higher kick time than A_2. That is, A_2 is better suited to attack, so A_1 should become the supporter.

Once we have a supporter, S, and the role is no longer stuck, it will turn back into an attacker if *any* of the following conditions hold:

- S, the ball, or the attacker (A) go back into the defensive half.
- A and S both see the ball, and S's estimate of its distance from the ball is smaller than A's.
- A doesn't see the ball, and S's estimate of the ball's distance from it is less than some constant (presently 300 mm).
- S has been a supporter for longer than some constant amount of time (presently 12 seconds).

Attacker and Defender Switching The following set of rules is used to allow the defender and an attacker to switch roles under appropriate circumstances.

- If a defender receives the information that there is another defender, it checks, using the global map data on the robots' distances to the ball, if it is a "better" defender (the one farthest from the ball). If so, it stays a defender. If not, it becomes an attacker.
- If a defender finds that there is no other defender, it still checks to see if the conditions are suitable for it to become an attacker. Here we test to see if the robot is closest to the ball and is in the section of the field that is on the top half on its side of the field. If it is, it sends a request to the attacker, asking to switch roles with it. Then, instead of becoming an attacker immediately, it waits for the attacker to

receive the request. Once this happens, we end up with more than one defender in the team (see the rule mentioned below), and this is resolved using the condition mentioned above. More information on message types and communication can be found in Section 9.

- When an attacker receives a request from a defender to switch roles, it automatically accepts. It does not need to participate in the decision making process because the defender had access to the same information as it did (as a result of the global maps) when it decided to switch. The attacker communicates its acceptance by simply becoming a defender. This is sufficient because the robots always communicate their roles to all of their teammates.

As mentioned above, our role system was developed quite hastily in the last week or so before competition. However, we feel that the system performs quite appropriately during games. The attacker/defender switches normally occur where they seem intuitively reasonable. The two attackers (with one becoming a supporter periodically), trying to score a goal, frequently look like a well organized pair of teammates. Nonetheless, there are certainly some instances during the games where we can point to situations where a role change happened at an inopportune time, or where it seems like they should "know better" than to do what they just did. Finding viable solutions to problems like this can be strikingly difficult. We look forward to making further progress on these problems and to improving the cooperation between the robots.

14 UT Assist

During the course of our development, we developed 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.

14.1 General Architecture

UT Assist consists of two pieces: a client and a server. The function of the client software, which is programmed in C++ and runs on an Aibo, is to queue and send data to the server. The server, which is programmed in Java and runs on a remote computer, is primarily concerned with collecting, displaying, and saving the data that it receives. We chose Java for the server because it put us on a relatively quick development cycle and gave us access to a rich library of pre-existing code. In particular, the ease with which Java handles networking and graphics made it an obvious candidate for this project.

Multiple clients can connect to one server. It is possible for more than one server to be active at once, provided that it does not listen on a port that is already taken by another service. All client-server communication takes place via TCP. The client software uses the default Open-R TCP endpoint interface, and the server software uses TCP networking classes described in the Java 2 API specification.

14.1.1 Typical Usage

During each Brain cycle on the Aibo, many different pieces of code can attempt to send data messages to the server. If the client is not already sending data to the server, it will accept each request and place the specified data into a queue. If the client is busy sending data, it will reject the request to send data. At the end of each Brain cycle, if the client has some data in its queue, it will divide the data into fixed-length packets and start sending the data to the server. This method of processing data ensures that only data from the most recent Brain cycle will be sent to the server and avoids a "backlog" situation, in which the speed at which data is queued exceeds the speed at which it can be delivered to the server.

Each message that enters the queue in the client is uniquely identified by a one-byte ID field. From the perspective of the client, each message it receives is simply a group of bytes associated with a unique ID. None of the packet processing that the client performs upon the queue of messages depends on the actual data in the messages, which allows users to add new types of data messages without modifying the client.